

Seminarausarbeitungen

Computational Intelligence bei Computerspielen

Prof. Dr. Günter Rudolph
Dipl.-Inform. Oliver Kramer
(Hrsg.)

Fakultät für Informatik
Technische Universität Dortmund

Seminar
Computational Intelligence bei Computerspielen
(Sommersemester 2007)

Inhalt

1. Klassische KI bei Computerspielen (Teil 1)
Tim-Hendrik Müller
2. Klassische KI bei Computerspielen (Teil 2)
Tobias Hein
3. Evolutionäre Algorithmen in Computerspielen
Frank Behler
4. Fuzzy-Logik in Computerspielen
Sebastian Schnelker
5. Neuronale Netze und selbstorganisierte Karten
Andreas Thom
6. Schwarmintelligenz und Computerspiele
André Barthelmes
7. KI in Egoshootern
Jan Quadflieg
8. Computational Intelligence in Echtzeitstrategiespielen
Simon Wessing
9. Computational Intelligence in Rollenspielen
Dominik Opolony
10. KI und CI in Pokergames
Holger Danielsiek
11. CI und KI bei Browserspielen
Daniel Haus
12. CI beim Roboterfußball
Patrick Szcypior
13. Computerspiele im Fokus der Sozialpsychologie
Moritz Hofmann
14. KI in Sportspielen
Armin Büscher

Klassische KI in Computerspielen

Wahrscheinlichkeit, Bayesnetze, stochastische Automaten

Tim-Hendrik Müller

Universität Dortmund
tim.mueller@uni-dortmund.de

Zusammenfassung Klassische künstliche Intelligenz findet Verwendung in Computerspielen um diese attraktiver zu gestalten. Hierbei werden Methoden angewandt, mithilfe derer ein Computer Entscheidungen treffen kann. Durch maschinelles Lernen können jene Entscheidungen im Verlauf eines Spiels weiter präzisiert werden. Die wesentlichen, hier betrachteten Teilgebiete sind Wahrscheinlichkeitsrechnung sowie die darauf aufbauenden Bayesnetze und stochastischen Automaten.

Diese Ausarbeitung führt in die jeweiligen Grundlagen ein und illustriert ihre Anwendung in Computerspielen anhand von realen und erdachten Beispielen.

1 Einleitung

Computerspiele dienen der Unterhaltung. Sie können so realisiert sein, dass ein menschlicher Spieler alleine spielt, mehrere menschliche Spieler gegeneinander spielen oder noch so genannte NPC (Non-Player-Characters zu deutsch: Nicht-Spieler-Figuren) hinzukommen. Beispiele für die erste Variante wären Mahjongg oder Sudoku, für die zweite Brettspielumsetzungen wie Schach oder Monopoly, aber auch Kampf-, Geschicklichkeits- oder Sportspiele. Jeder menschliche Gegner lässt sich durch NPCs ersetzen. Hier lässt sich dann künstliche Intelligenz in Spielen einsetzen.

Die künstliche Intelligenz kann auf verschiedene faire und unfaire Weisen umgesetzt werden. So gibt es Spiele, in denen Wissen über den Gegner unfairerweise zum Sieg verhilft. Wenn man zum Beispiel bei einem Quartett die aktuelle Karte oder bei Schwarzer-Peter die Reihenfolge der Karten kennt, ist es ein leichtes, den Spielzug für sich zu entscheiden.

Bei Spielen mit vollständiger Information gibt es Algorithmen mit Hilfe derer ein NPC gewinnen bzw. nicht verlieren wird. So gewinnt bei „Vier gewinnt“ der beginnende Spieler [A88]. Da ein solcher, immer gewinnender Übergegner dem Zweck der Unterhaltung nicht dienlich ist, gilt es, das Spiel interessanter zu gestalten, beispielsweise indem man dem NPC Wissen vorenthält. Andere Spiele hingegen lassen sich nicht einmal durch vollständiges Wissen sicher gewinnen, da es hier keine Voraussage geben kann. Man denke an Kampfspiele, in denen der NPC die Reaktionen des menschlichen Spielers nicht vorhersehen kann.

2 Wahrscheinlichkeit

2.1 Motivation

Wie in der Einleitung erläutert, gilt es, das Wissen eines NPC zu beschränken. Somit verfügt dieser nur noch über unsicheres Wissen. Die bekannteste Möglichkeit dies zu behandeln, ist die Wahrscheinlichkeitstheorie. Hierbei ist bekannt, welche Ereignisse eintreten können und wie häufig deren Auftreten im Mittel ist. Unbekannt hingegen ist, welches Ereignis als nächstes auftreten wird.

Des Weiteren lassen sich durch Wahrscheinlichkeiten Charaktere von Computerspielern simulieren. So könnten Eigenschaften wie handwerkliches Geschick, Umgang mit Waffen, Hören von Lauten etc. als Erfolgswahrscheinlichkeiten angegeben werden. Bei jeder Aktion würde dann der Erfolg anhand der persönlichen Wahrscheinlichkeit zufällig bestimmt.

2.2 Definitionen

nach [GRS00] und [BS04].

Definition 1. *Ereignisraum. Der Ereignisraum ist die Menge aller möglichen Ereignisse. Er wird mit Ω bezeichnet. $\Omega = \{\omega_1, \omega_2, \dots, \omega_n\}$*

Definition 2. *Ereignis. Ein Ereignis ist eine nicht-leere Teilmenge des Ereignisraumes. Im Folgenden mit e bezeichnet. $e \in \wp(\Omega)$*

Definition 3. *Klassische Wahrscheinlichkeit. Ist ein endlicher Ereignisraum und Kenntnis der A-Priori-Wahrscheinlichkeiten gegeben, so ist die klassische Wahrscheinlichkeit das Verhältnis der Anzahl der möglichen Eintritte eines Ereignisses E (günstige Ereignisse) (n) zu der Gesamtanzahl aller möglichen Ereignisse (N). $p = P(E) = n/N$*

Definition 4. *Relative Häufigkeit. Ein Experiment mit zufälligem Ausgang werde unter denselben Bedingungen N -mal durchgeführt. $h_N(A)$ bezeichne die Zahl der Fälle, in denen A eintritt: die absolute Häufigkeit. Die relative Häufigkeit ist der Quotient $r_N(A) = \frac{h_N(A)}{N}$.*

2.3 Regeln

Diese Regeln wurden nach [BS04] zusammengestellt.

Regel 1. Die Wahrscheinlichkeit eines Ereignisses ist eine reelle Zahl zwischen 0 und 1. $0 \leq P(A) \leq 1$. Dabei bedeutet 0 absolute Sicherheit, dass das Ereignis nie eintritt und 1 absolute Sicherheit, dass das Ereignis eintritt.

Regel 2. Im Venn-Diagramm in Abb. 1 repräsentiert S den Ereignisraum, A und B zwei verschiedene Ereignisse und die Punkte einige Elementarereignisse.

$$P(S) = 1$$

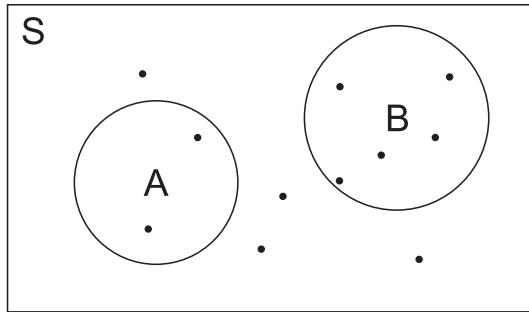


Abbildung 1. Venn-Diagramm des Ereignisraumes **S** (Regel 2)

Regel 3. Wenn die Wahrscheinlichkeit für das Eintreten von A $P(A)$ ist, so ist die Wahrscheinlichkeit des Nichteintretens von A $P(A') = 1 - P(A)$, die Gegenwahrscheinlichkeit. D.h. ein Ereignis tritt entweder ein oder nicht. Dies zeigt auch das Venn-Diagramm in Abb. 2:

$$A \cup A' = S$$

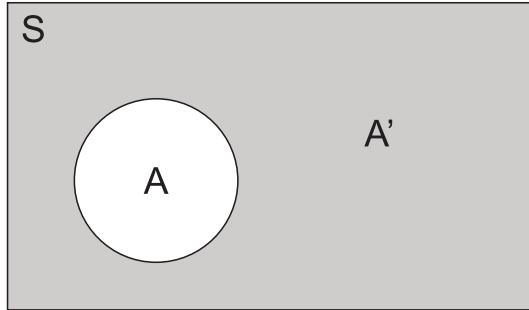


Abbildung 2. Wahrscheinlichkeit und Gegenwahrscheinlichkeit (Regel 3)

Regel 4. Wenn sich zwei Ereignisse gegenseitig ausschließen, kann nur eines der beiden Ereignisse zur gleichen Zeit eintreten. Die Ereignisse Leben und Tod z.B. schließen sich gegenseitig aus. Abbildung 3 zeigt die zwei sich gegenseitig ausschließenden Ereignisse A und B .

$$A \cap B = \emptyset$$

Die Wahrscheinlichkeit des Auftretens von A oder B ist

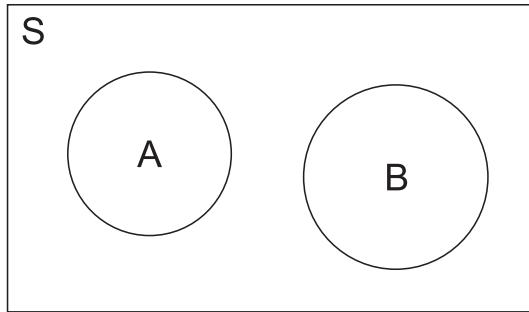


Abbildung 3. Gegenseitiger Ausschluss zweier Ereignisse (Regel 4)

$$P(A \cup B) = P(A) + P(B)$$

wobei $P(A \cup B)$ die Wahrscheinlichkeit für das Eintreten von A oder B und $P(A)$ für das Eintreten von A und $P(B)$ für das Eintreten von B bezeichnen. Diese Regel kann auf beliebig viele, sich gegenseitig ausschließende Ereignisse angewendet werden.

$$P(A \cup B \cup C \cup D \cup E) = P(A) + P(B) + P(C) + P(D) + P(E)$$

Regel 5. Falls sich zwei Ereignisse A und B nicht gegenseitig ausschließen, Abb. 4, so berechnet sich das Eintreten von A oder B zu:

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

Das heißt, die Wahrscheinlichkeit des Auftretens beider Ereignisse ($P(A \cap B)$) muss von der Summe der Wahrscheinlichkeiten des Eintretens von A oder von B abgezogen werden. Ein Beispiel für zwei sich nicht ausschließende Ereignisse wäre die Lebendigkeit eines Individuums und der Umstand einer Verletzung.

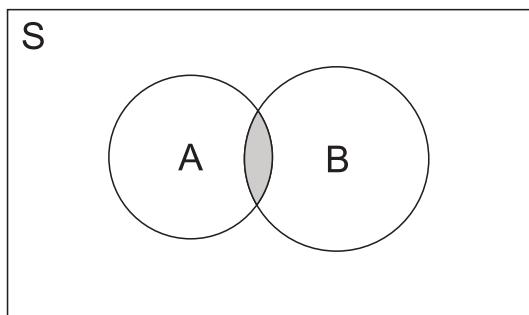


Abbildung 4. Zwei sich nicht ausschließende Ereignisse (Regel 5)

Wenn sich mehrere Ereignisse überschneiden, s. Abb. 5, so muss man die Wahrscheinlichkeiten aller Schnittmengen subtrahieren.

Man erhält hier also

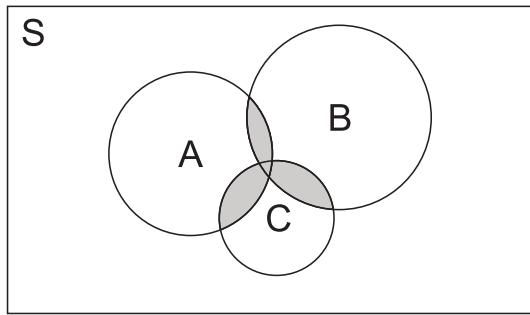


Abbildung 5. Überschneidung mehrerer Ereignisse (Regel 5)

$$P(A \cup B \cup C) = (P(A) + P(B) + P(C)) - (P(A \cap B) + P(A \cap C) + P(B \cap C)) + P(A \cap B \cap C)$$

Regel 6. Zwei Ereignisse A und B sind unabhängig voneinander, genau dann, wenn gilt, dass die Verbundwahrscheinlichkeit gleich dem Produkt der beiden Einzelwahrscheinlichkeiten ist:

$$P(A \cap B) = P(A) * P(B)$$

Regel 7. Wenn ein Ereignis B ein Ereignis A bedingt, spricht man von bedingter Wahrscheinlichkeit. Diese berechnet sich wie folgt:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}, P(B) \neq 0$$

Ein Schuss beispielsweise kann eine Verletzung bedingen.

Regel 8. Wenn ein Ereignis B von einem Ereignis A bedingt wird, berechnet sich die Verbundwahrscheinlichkeit aus

$$P(A \cap B) = P(A) * P(B|A).$$

Das heißt, wenn zum Beispiel ein Schuss eine Verletzung bedingt, müssen wir die Wahrscheinlichkeit eines Schusses mit der Wahrscheinlichkeit, dass dieser eine Verletzung bedingt, multiplizieren.

Regel 9. Aus Regel 8 können wir Bayes Regel herleiten:

$$P(A \cap B) = P(A) * P(B|A)$$
$$P(A \cap B) = P(B) * P(A|B)$$

$$P(A) * P(B|A) = P(B) * P(A|B)$$
$$P(B|A) = \frac{P(B) * P(A|B)}{P(A)}$$

Mit dieser können wir nun, die Bedingungen von A und B herumdrehen. Dies werden wir in Kapitel 3 nutzen.

2.4 Beispiele

Zu elementarer Verwendung der Grundlagen von Wahrscheinlichkeiten ließ sich keine Literatur finden. Eigene Recherchen allerdings ergaben die folgenden Erkenntnisse.

Doom Doom ist ein früher Egoshooter der Firma id-Software [DOOM]. Der erste Teil wurde im Jahr 1993 veröffentlicht. Der Zufallsgenerator ist eine einfache Look-Up-Table mit 256 Einträgen aus dem Intervall zwischen 0 und 255, wobei einige Zahlen gar nicht, andere bis zu fünfmal auftreten. Mithilfe einer Methode `P_Random()`, welche einen Integerwert zurückliefert, wird der nächste Wert aus dieser Tabelle zurückgegeben.

Der Schaden einer Pistole beispielsweise wird durch „`damage = ((P_Random()%5)+1) * 3;`“ berechnet. Hier wird demnach eine Zufallszahl modulo 5 gerechnet. D.h. es bleiben nach der Addition mit 1 die Werte von 1 bis 5, welche anschließend mit 3 multipliziert werden. Eine Pistole kann dem Gegner also einen Schaden von 3, 6, 9, 12 und 15 zufügen.

UFO - Enemy unknown Wie Tests von Benutzern zeigen [UFO], werden auch in dem Spielesklassiker „UFO - Enemy unknown“ Wahrscheinlichkeiten u.a. für Treffer verwendet.

Die Trefferwahrscheinlichkeit berechnet sich dort wie folgt:

$$\begin{aligned}
P(\text{Treffer}) &= a * b * c * d * e * f \\
a &= \text{Präzision des Schützen} \\
b &= \text{Präzision der Waffe} \\
c &= \text{Bonus, 115% falls Schütze kniet, 100% sonst} \\
d &= \text{Zweihandwaffenabzug, 80% falls eine Beidhandwaffe} \\
&\quad \text{mit nur einer Hand gehalten wird, 100% sonst} \\
e &= \text{Verwundungsabzug \% der verbleibenden Gesundheit} \\
f &= \text{Abzug für kritische Wunden, (100\% - (10\% pro} \\
&\quad \text{kritischer Arm- oder Kopfwunde, bis zu 90\%)}
\end{aligned}$$

Hier wird ersichtlich, dass die Entwickler davon ausgingen, dass die Ereignisse a bis f voneinander unabhängig sind und daher gemäß Regel 6 multipliziert werden können.

Beispiel 1. Gegeben sei ein kniender Schütze mit 88% Präzision, dessen Waffe, die er mit beiden Händen hält, eine Präzision von 60% hat. Er hat eine verbleibende Gesundheit von 87% und eine kritische Wunde.

Die Trefferwahrscheinlichkeit beläuft sich demnach auf:

$$0,88 * 0,6 * 1,15 * 1 * 0,87 * 0,9 \approx 48\%$$

3 Bayesnetze

3.1 Motivation

Allein durch Zufall gesteuerte Computergegner bieten auf Dauer keine Veränderung der Spielsituation. Interessanter lassen sich Spiele gestalten, wenn sich Gegner weder immer dumm noch immer perfekt noch immer zufällig verhalten. In diesem Kapitel wollen wir daher einfache Umsetzungen aufzeigen, die es einem NPC ermöglichen, Schlussfolgerungen zu ziehen und diese im Verlauf des Spiels weiter zu lernen. Tom M. Mitchell [M02] beschreibt, wie Bayes Theorem für Konzeptlernern eingesetzt werden kann. Allerdings weist er darauf hin, dass einige andere Algorithmen dieselben Hypothesen aufstellen, wie der Ansatz mit Bayes, jene aber nicht explizit Wahrscheinlichkeiten manipulieren und bedeutend effizienter sind.

3.2 Definitionen

Definition 5. *Bayesnetz [BKI06]. Ein Bayesnetz $B = (V, E, P)$ ist ein gerichteter, azyklischer Graph $G = (V, E)$ mit der Wahrscheinlichkeitsverteilung P*

über $V = A_1, \dots, A_n$, so dass jede Variable A_i bedingt unabhängig von ihren Nichtnachkommen $nd(A_i)$, gegeben ihre Elternknoten $pa(A_i)$ ist: $A_i \perp\!\!\!\perp_{P} nd(A_i) \mid pa(A_i)$ für alle $i = 1, \dots, n$

3.3 Schlussfolgern mit Bayesnetzen

Es gibt drei verschiedene Typen der Inferenzbildung in Bayesnetzen (s. [BS04]).

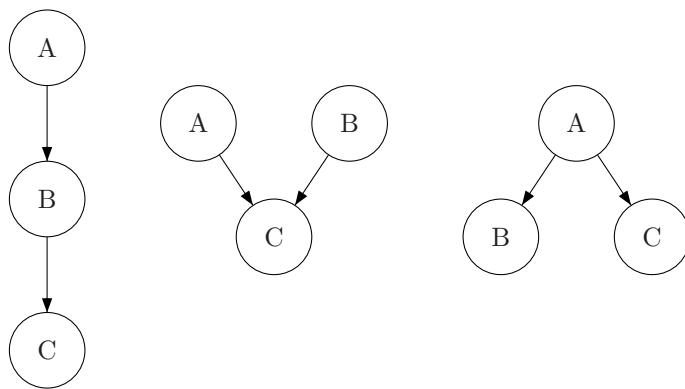


Abbildung 6. Inferenz in Bayesnetzen: Drei Beispiele - Prädiktives Schlussfolgern, Erklärung und Diagnostisches Schlussfolgern

Prädiktives Schlussfolgern Beim prädiktiven Schlussfolgern können wir aus dem Wissen, dass ein Ereignis aufgetreten ist, auf die Wahrscheinlichkeit eines, durch jenes bedingtes, Ereignisses schließen. (Im Beispiel links.)

Erklärung Eine Erklärung für das Eintreten eines Ereignisses wird gesucht. Im Beispiel in der Mitte würde beim Eintreten von C die Wahrscheinlichkeit für das Eintreten von B und A steigen, weil wir wissen, dass diese C bedingen.

Diagnostisches Schlussfolgern Hierbei wird aus dem Auftreten von Symptomen auf die Wahrscheinlichkeit einer Krankheit geschlossen. In dem rechten Netz wären B und C die Symptome, die von der Krankheit A bedingt werden.

3.4 Beispiel

Als Beispiel (vgl. Bourg et al. [BS04]) soll ein fiktives Spiel dienen, in dem ein Spieler Schätze in Schatztruhen verstecken kann. Diese kann er verschließen und Fallen in ihnen montieren. Ein Computergegner soll nun entscheiden, ob er eine Truhe öffnen soll oder nicht. Er könnte, wie in der Einleitung bereits erwähnt, betrügen und bereits wissen, welche Truhen mit Fallen ausgestattet sind. Nach dem Ansatz aus Kapitel 2 könnte er auch die Kisten zufällig öffnen. Hier soll er allerdings die Zusammenhänge zwischen der Verschlossenheit einer Truhe und der Immanenz eines Schatzes bzw. einer Falle ausnutzen und schlussfolgern können.

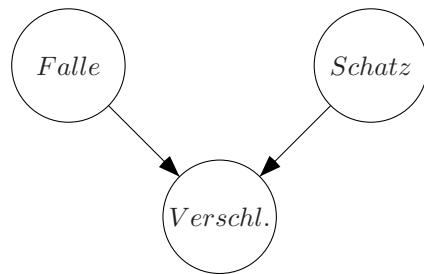


Abbildung 7. Bayesnetz: Falle und Schatz bedingen Verschlossen

Im Beispiel sieht man, dass die Verschlossenheit einer Truhe von dem Inhalt eines Schatzes oder einer Falle bedingt ist. Dies allerdings nur zu einer bestimmten Wahrscheinlichkeit. Genauso gut könnte eine Truhe verschlossen sein, ohne dass ein Schatz oder eine Falle enthalten ist. Oder aber eine Truhe könnte unverschlossen sein trotz einer Falle.



Abbildung 8. Bayesnetz: Falle bedingt Verschlossen

Betrachten wir nur die Abhängigkeit zwischen Falle und Verschlossen (siehe

Abb. 8), dann ergeben sich die Wahrscheinlichkeiten gemäß den unten stehenden Gleichungen. *Falle* ist eine zweiwertige Variable, die wahr ist, falls eine Falle installiert bzw. falsch, falls keine Falle installiert ist. Ebenso gilt für *Verschlossen*: wahr, falls die Truhe verschlossen bzw. falsch, falls die Truhe unverschlossen ist.

$$P(Falle = \text{wahr}) = p_F \quad (1)$$

$$P(Falle = \text{falsch}) = (1 - p_F) \quad (2)$$

Wir bezeichnen die Wahrscheinlichkeit für eine Falle mit p_F (1). Die Wahrscheinlichkeit, dass keine Falle enthalten ist, also die Gegenwahrscheinlichkeit, ist nach Regel 3 gleich $(1 - p_F)$.

$$P(Verschlossen = \text{wahr} | Falle = \text{wahr}) = p_{Vw} \quad (3)$$

$$P(Verschlossen = \text{wahr} | Falle = \text{falsch}) = p_{Vf} \quad (4)$$

$$P(Verschlossen = \text{falsch} | Falle = \text{wahr}) = (1 - p_{Vw}) \quad (5)$$

$$P(Verschlossen = \text{falsch} | Falle = \text{falsch}) = (1 - p_{Vf}) \quad (6)$$

p_{Vw} ist hier Bezeichner für die Wahrscheinlichkeit, dass die Falle verschlossen ist unter der Voraussetzung, dass eine Falle existiert.

Schlussfolgerungen ziehen Wie im Ansatz der relativen Häufigkeit können wir die Wahrscheinlichkeiten bestimmter Ereignisse über Verhältnisse von Eintreten bestimmter Ereignisse zur Anzahl der gesamten Ereignisse durch Mitzählen bestimmen. Somit lernt ein NPC im Laufe des Spieles hinzu.

$$P(Falle = \text{wahr}) = p_F = \frac{\#\text{geöffnete Truhen mit Falle}}{\#\text{geöffnete Truhen}} \quad (7)$$

Wenn die zu untersuchende Truhe nicht verschlossen ist, ist die Wahrscheinlichkeit für eine Falle p_F . Ist sie hingegen verschlossen, so lässt sich die Wahrscheinlichkeit wie folgt bestimmen.

$$P(Falle | Verschlossen) = \frac{P(Verschlossen | Falle) * P(Falle)}{P(Verschlossen)} \quad (8)$$

$P(Falle)$ und $P(Verschlossen | Falle)$ wurden im Verlauf des Spiels stetig mitgezählt. Zu berechnen bleibt $P(Verschlossen)$. Wir wissen, eine Truhe kann auf zwei Arten verschlossen sein: Entweder mit Falle oder ohne. Die Wahrscheinlichkeiten dafür sind $P(Verschlossen | Falle) * P(Falle)$ sowie $P(Verschlossen | \neg Falle) *$

$P(\neg Falle)$. Da sich beide gegenseitig ausschließen, dürfen wir die Wahrscheinlichkeiten addieren und erhalten:

$$P(Verschlossen) = P(Verschlossen|Falle)P(Falle) + P(Verschlossen|\neg Falle)P(\neg Falle) \quad (9)$$

Ein Beispiel mit Zahlen:
Der NPC hat folgendes mitgezählt.

- 120 geöffnete Truhen, davon
- 45 mit Fallen und
- $120 - 45 = 75$ ohne Fallen.
- 36 derer mit Falle waren verschlossen,
- 3 derer ohne Falle waren verschlossen.

$$P(Falle) = 45/120 = 0,375 \quad (10)$$

$$P(\neg Falle) = 75/120 = 0,625 \quad (11)$$

$$P(Verschlossen|Falle) = 36/45 = 0,800 \quad (12)$$

$$P(Verschlossen|\neg Falle) = 3/75 = 0,040 \quad (13)$$

$$P(Verschlossen) = (0,8 * 0,375 + 0,04 * 0,625) = 0,325 \quad (14)$$

$$\begin{aligned} P(Falle|Verschlossen) &= \frac{P(Falle) * P(Verschlossen|Falle)}{P(Verschlossen)} \\ &= \frac{0,375 * 0,8}{0,325} \approx 0,952 \end{aligned} \quad (15)$$

Gleichung (15) zeigt eindrucksvoll, dass die A-Posteriori-Wahrscheinlichkeit deutlich über der A-Priori-Wahrscheinlichkeit (10) liegt.

$$P(\neg Verschlossen|Falle) = 1 - P(Verschlossen|Falle) = 0,2 \quad (16)$$

$$P(\neg Verschlossen) = 1 - P(Verschlossen) = 0,675 \quad (17)$$

$$\begin{aligned}
P(Falle|\neg Verschlossen) &= \frac{P(Falle) * P(\neg Verschlossen|Falle)}{P(\neg Verschlossen)} \\
&= \frac{0,375 * 0,2}{0,675} \approx 0,11
\end{aligned} \tag{18}$$

Bei einer nicht verschlossenen Truhe würde der NPC laut (18) eine geringere Wahrscheinlichkeit für das Vorhandensein einer Falle annehmen.

4 Stochastische Automaten

4.1 Motivation

Eine naheliegende Erweiterung in Bezug auf Wahrscheinlichkeiten und Spiele ist die der endlichen Automaten zu stochastischen endlichen Automaten (englisch: probabilistic automaton, PA). Hierbei werden Übergänge zusätzlich zu den Symbolen mit Wahrscheinlichkeiten annotiert.

4.2 Definitionen

Arto Salomaa [S69] gab die folgenden Definitionen zu PA.

Definition 6. *Übergangswahrscheinlichkeit. Seien s_1, \dots, s_n die Zustände des Automaten, dann ist $p_i(s, x)$ die Übergangswahrscheinlichkeit vom Zustand s zu einem Zustand s_i , beim Lesen des Symbols x .*

Für alle s und x gilt:

$$\sum_{i=1}^n p_i(s, x) = 1, p_i(s, x) \geq 0.$$

Definition 7. *Startverteilung. Anstelle eines Startzustandes besitzt ein PA eine Startverteilung.*

Definition 8. *Stochastische Matrix. Mit einer (n -dimensionalen) stochastischen Matrix wird eine $(n \times n)$ Matrix bezeichnet mit*

$$M = \|p_{ij}\|_{1 \leq i, j \leq n}$$

bei der p_{ij} für jedes i, j nicht-negative reelle Zahlen sind und gilt

$$\sum_{i=1}^n p_{ij} = 1 \text{ für } (i = 1, \dots, n).$$

Definition 9. *Stochastischer Zeilen- (oder Spalten-)vektor. Mit einem n -dimensionalen stochastischen Zeilen- (oder Spalten-)vektor wird ein n -dimensionaler Vektor bezeichnet, dessen Einträge nicht-negativ sind und in Summe 1 ergeben.*

Definition 10. Endlicher stochastischer Automat. Ein endlicher stochastischer Automat über dem endlichen Alphabet I ist $PA = (S, s_0, M)$, mit $S = s_1, \dots, s_n$ mit $n \geq 1$, der endlichen Menge der Zustände, s_0 ein n -dimensionaler Zeilenvektor (der Startverteilung) und M bildet von I in eine Menge n -dimensionaler stochastischer Matrizen ab.

4.3 Beispiel

Ein kleines Beispiel soll genügen, um die Anwendung stochastischer Automaten in Spielen zu erläutern. In einem fiktiven Spiel gebe es einen NPC, welcher als Händler agiert. Mit diesem kann man fair oder unfair tauschen oder ihm Waren schenken. Je nach seiner bisherigen Erfahrung ist dieser entweder loyal, gutmütig, neutral oder gereizt gestimmt. Dieses wollen wir in einem einfachen stochastischen Automaten simulieren.

$$PA_{Händler} = (\{loyal, gutmütig, neutral, gereizt\}, s_0, M),$$

das Alphabet sei

$$I = \{fairerTausch, unfairerTausch, geschenk\}.$$

Wir nehmen an, die meisten Händler seien, wenn sie den Spieler noch nicht kennen, neutral. Auch seien einige wenige gutmütige und gereizte Gemüter dabei, allerdings ist kein Händler von vornherein loyal. Eine Startverteilung könnte demnach so aussehen:

$$s_0 = (0, \frac{3}{100}, \frac{9}{10}, \frac{7}{100})$$

Die (erdachten) Übergangswahrscheinlichkeiten werden in den folgenden Matrizen dargestellt:

$$M(fairerTausch) = \begin{pmatrix} \frac{8}{10} & \frac{2}{10} & 0 & 0 \\ \frac{3}{10} & \frac{4}{10} & \frac{3}{10} & 0 \\ \frac{10}{10} & \frac{10}{10} & \frac{10}{10} & \frac{1}{10} \\ 0 & \frac{2}{10} & \frac{7}{10} & \frac{1}{10} \\ 0 & 0 & \frac{3}{10} & \frac{7}{10} \end{pmatrix}$$

$$M(unfairerTausch) = \begin{pmatrix} \frac{5}{10} & \frac{4}{10} & 0 & \frac{1}{10} \\ 0 & \frac{10}{10} & \frac{10}{10} & 0 \\ 0 & 0 & \frac{10}{10} & \frac{6}{10} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$M(geschenk) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \frac{7}{10} & \frac{2}{10} & \frac{1}{10} & 0 \\ 0 & \frac{4}{10} & \frac{6}{10} & 0 \\ \frac{1}{10} & 0 & 0 & \frac{9}{10} \end{pmatrix}$$

Die Zeilen und Spalten werden hier jeweils von den Zuständen des Automaten bezeichnet. Das heißt, die Wahrscheinlichkeit eines Überganges von neutral nach gutmütig beträgt bei einem Geschenk $\frac{4}{10}$.

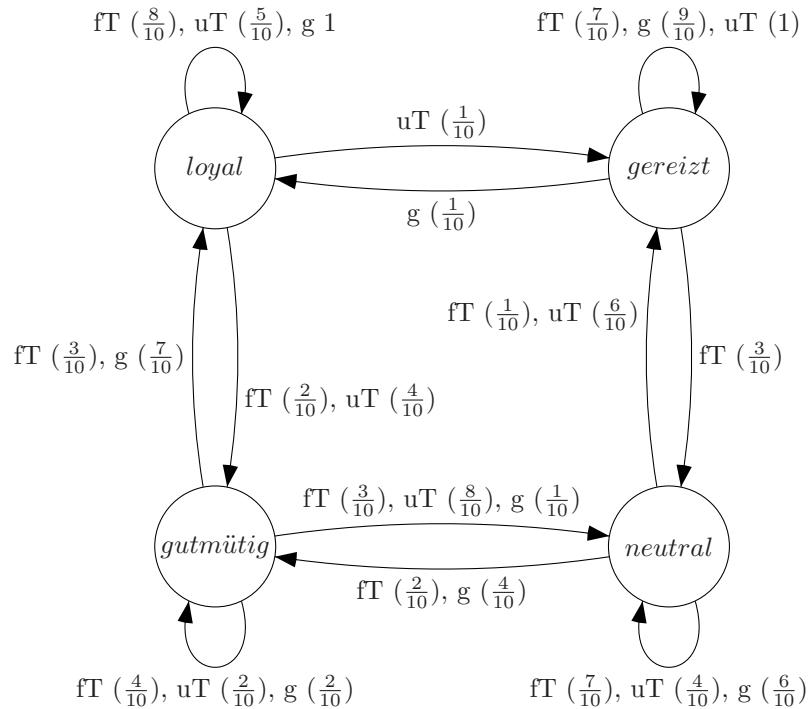


Abbildung 9. Stochastischer Automat: $PA_{Händler}$, fT (fairerTausch), uT (unfairerTausch), g (geschenk)

Abbildung 9 zeigt den Automaten grafisch. Zustände werden durch Kreise mit annotierten Zustandsnamen dargestellt, Zustandsübergänge durch Pfeile, wobei die Pfeilspitze das Ziel markiert. An diesen sind jeweils die (abgekürzten) Sym-

bole mit den zugehörigen Übergangswahrscheinlichkeiten notiert, sofern diese ungleich 0 sind. Bei Werten gleich 0 gibt es keinen Übergang.

Wenn der Automat nun ein Zeichen liest, bestimmt dieses Zeichen *und* eine danach bestimmte Zufallszahl, in welchen Zustand der Automat wechselt.

An kleinen Beispielen werden wir Anwendung eines stochastischen Automaten besser kennen lernen.

Beispiel 2. Im Folgenden sei (der besseren Lesbarkeit halber) $P(s \rightarrow s_i, x) = p_i(x)$ und $P(s_i) = s_{0i}$. Eine einfache Frage, die uns der Automat beantworten kann, ist z.B. „Wie Wahrscheinlich ist es, dass ein neuer Händler nach drei Geschenken gereizt ist?“. Bei näherer Betrachtung des PA fällt auf, dass es nur einen einzigen Übergang bei einem Geschenk in den Zustand gereizt gibt, und zwar, wenn der Händler bereits gereizt war. Das heißt, wir müssen nur die Übergänge betrachten, wie sie in Abbildung 10 zu sehen sind. In den Abbildungen aufeinanderfolgende Ereignisse sind voneinander unabhängig. Daher werden die Wahrscheinlichkeiten nach Regel 8 für die Verbundwahrscheinlichkeit dort multipliziert.

Die Wahrscheinlichkeit, dass ein Händler zu Beginn gereizt ist, beträgt $\frac{7}{100}$.

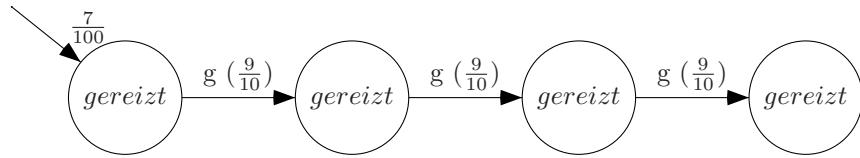


Abbildung 10. „Wie Wahrscheinlich ist es, dass ein neuer Händler nach drei Geschenken gereizt ist?“

Bei drei Geschenken ist der einzige Pfad am Ende wieder in gereizt zu bleiben derjenige, der (jeweils mit einer Wahrscheinlichkeit von $\frac{9}{10}$) aus gereizt zurück nach gereizt führt. Die Gesamtwahrscheinlichkeit ergibt sich nach (19) zu etwa 5%.

$$\begin{aligned}
 & P(\text{gereizt}) * P(\text{gereizt} \rightarrow \text{gereizt}, g) \\
 & * P(\text{gereizt} \rightarrow \text{gereizt}, g) * P(\text{gereizt} \rightarrow \text{gereizt}, g) = \frac{7}{100} * \frac{9}{10} * \frac{9}{10} * \frac{9}{10} \quad (19) \\
 & = \frac{5103}{100000} = 5,103\%
 \end{aligned}$$

Beispiel 3. Unser zweites Beispiel behandelt die Frage „Zu was wird ein gutmütiger Händler nach einem, durch ein Geschenk wieder wettgemachten, unfairen Tausch?“. Aus gutmütig führt die Eingabe unfairerTausch nach gutmütig oder neutral. Ein Geschenk führt aus gutmütig zu loyal, gutmütig oder neutral und aus neutral zu gutmütig oder neutral. Es gibt also nur die Fälle, die in den Abbildungen 11, 12 und 13 gezeigt sind. Wir erwarten also, dass sich die ergebenden Wahrscheinlichkeiten zu 1 addieren werden. Die drei oben genannten Fälle las-



Abbildung 11. Gutmütiger Händler wird nach einem unfairem Tausch und einem Geschenk loyal

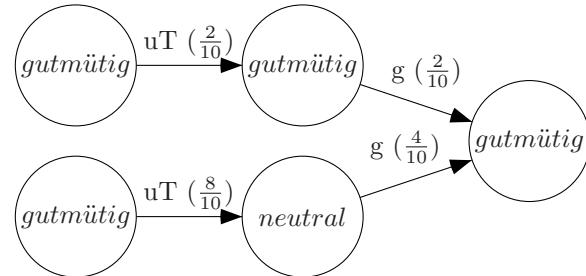


Abbildung 12. Gutmütiger Händler bleibt nach einem unfairem Tausch und einem Geschenk gutmütig

sen sich nun berechnen. In Abbildung 12 und 13 sind die letzten Zustände jeweils verschmolzen worden, da hier unsere Betrachtung am selben Zustand endet. Es handelt sich aber um zwei sich gegenseitig ausschließende Ereignisse. Nach Regel 4 werden die Wahrscheinlichkeiten dann addiert.

Um loyal zu werden, muss der Händler laut Abbildung 11 zunächst gutmütig

werden und danach loyal.

$$\begin{aligned}
 P(\text{wirdLoyal}) &= P(\text{gutm\"utig} \rightarrow \text{gutm\"utig}, uT) * P(\text{gutm\"utig} \rightarrow \text{loyal}, g) \\
 &= \frac{2}{10} * \frac{7}{10} \\
 &= \frac{14}{100}
 \end{aligned} \tag{20}$$

Die Wahrscheinlichkeit, dass der Händler loyal wird, beträgt laut (20) 14%.

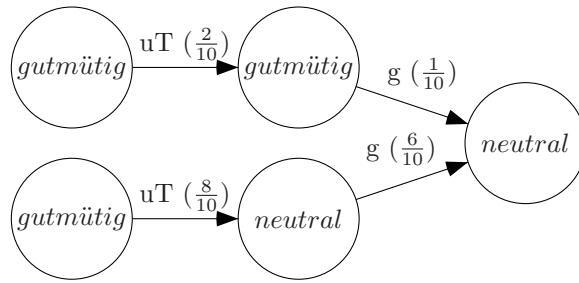


Abbildung 13. Gutm\"utiger Händler wird nach einem unfairem Tausch und einem Geschenk neutral

Zwei Möglichkeiten gibt es, bei denen er gutm\"utig bleibt. Diese sind: Zweimal gutm\"utig zu bleiben oder erst neutral und anschließend wieder gutm\"utig zu werden.

$$\begin{aligned}
 P(\text{bleibtGutm\"utig}) &= P(\text{gutm\"utig} \rightarrow \text{gutm\"utig}, uT) * P(\text{gutm\"utig} \rightarrow \text{gutm\"utig}, g) \\
 &\quad + P(\text{gutm\"utig} \rightarrow \text{neutral}, uT) * P(\text{neutral} \rightarrow \text{gutm\"utig}, g) \\
 &= \frac{2}{10} * \frac{2}{10} + \frac{8}{10} * \frac{4}{10} \\
 &= \frac{36}{100}
 \end{aligned} \tag{21}$$

In 36 von 100 Fällen bleibt der Händler gutm\"utig.

Letztlich bleiben noch zwei mögliche Pfade, auf denen er neutral wird: Er bleibt

gutmütig und wird neutral oder er wird neutral und bleibt es.

$$\begin{aligned}
 P(\text{wirdNeutral}) &= P(\text{gutmütig} \rightarrow \text{gutmütig}, uT) * P(\text{gutmütig} \rightarrow \text{neutral}, g) \\
 &\quad + P(\text{gutmütig} \rightarrow \text{neutral}, uT) * P(\text{neutral} \rightarrow \text{neutral}, g) \\
 &= \frac{2}{10} * \frac{1}{10} + \frac{8}{10} * \frac{6}{10} \\
 &= \frac{50}{100}
 \end{aligned} \tag{22}$$

Und in der Hälfte aller Fälle ($\frac{50}{100}$) wird er neutral. Dass wir nun alle Fälle tatsächlich erfasst haben, zeigt sich dadurch, dass die Summe aller Wahrscheinlichkeiten $\frac{14}{100} + \frac{36}{100} + \frac{50}{100} = 1$ ergibt.

5 Resümee

In den vorangegangenen Kapiteln wurde gezeigt, wie auf einfache Weise Wahrscheinlichkeiten genutzt werden können, um Spiele um künstliche Intelligenz zu erweitern. Hierzu wurden zunächst die theoretischen Grundlagen erläutert. Ein erster fortgeschritten Ansatz ist die Verwendung der Bayesregel. Mit dieser kann ein Programm Rückschlüsse ziehen um nicht ausschließlich Wahrscheinlichkeiten durch relative Häufigkeiten zu speichern. Ein weiteres Beispiel für das Lernen durch Bayes Regel ist BILL [KFM90]. In den späten 80er Jahren des zwanzigsten Jahrhunderts wurde IAGO, das bis dahin führende Programm bei Othello (auch Reversi), von BILL geschlagen. Dieses nutzte unter anderem Bayes'sches Lernen.

Im letzten Kapitel wurden Wahrscheinlichkeiten mit dem bekannten Konzept der endlichen Automaten verknüpft. Obgleich diese Idee bereits sehr alt ist, ließ sich hierüber wenig Literatur und noch weniger reale, diese Idee implementierende Spiele finden. Ohnehin gab es gewisse Schwierigkeiten, Informationen über Methoden der künstlichen Intelligenz von (aktuellen) Spielen zu finden, was aus kommerziellen Gründen durchaus verständlich ist.

Literatur

- [A88] Victor Allis. *A Knowledge-Based Approach of Connect-Four. The Game is Solved: White wins.* M.Sc. Thesis, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam. 1988.
- [BKI06] Christoph Beierle, Gabriele Kern-Isberner. *Methoden wissensbasierter Systeme.* Wiesbaden. 2006.
- [BS04] David M. Bourg, Glenn Seemann. *AI for game developers.* Beijing. 2004.
- [DOOM] *Doom source code.*
http://doom.wikia.com/wiki/Doom_source_code

- [GRS00] Günther Görz, Claus-Rainer Rollinger, Josef Schneeberger.
Handbuch der künstlichen Intelligenz. München. 2000.
- [KFM90] Lee, Kai-Fu and Sanjoy Mahajan. *The Development of a World Class Othello Program*. Artificial Intelligence 43 Issue 1: 21-36. 1990.
- [M02] Tom M. Mitchell. *Machine learning*. New York. 2002.
- [S69] Arto Salomaa. *Theory of automata*. Oxford. 1969.
- [UFO] *UFOPaedia*. <http://www.ufopaedia.org>

– Seminararbeit –

Computational Intelligence & Games Klassische KI In Computerspielen

Tobias Hein
`tobias.hein@cs.uni-dortmund.de`

Fachbereich Informatik
Lehrstuhl für Algorithm Engineering (LS11)

TU Dortmund – D-44221 Dortmund – Germany

Zusammenfassung Im Rahmen des Seminars *Computational Intelligence & Games* befasst sich diese Seminararbeit mit den „klassischen“ KI Methoden in Videospielen. Die Konzepte und Techniken von Entscheidungsbäumen, insbesondere Spielbäumen, Finite-state Machines und des A^* -Algorithmus sollen hier anhand konkreter Anwendungsbeispiele genauer betrachtet werden.

1 Decision Trees

Decision Trees, auch Entscheidungsbäume genannt, finden ihren Ursprung im Bereich des Machine Learnings. Neben den Finite-state Machines gehören sie zu den Techniken, die schon zu Beginn der Entwicklung künstlicher Intelligenzen in Videospielen zum Einsatz kamen. Die Datenstruktur und ihre Algorithmen erlauben es, den Prozess der intelligenten Entscheidungsfindung virtueller Agenten effizient umzusetzen. Nachfolgend soll das Konzept der Decision Trees, sowie deren Verwendung in Computerspielen genauer betrachtet werden.

1.1 Definition

Decision Trees sind baumartige Datenstrukturen. Formal lassen sie sich als zusammenhängende und kreisfreie Graphen $G = (V, E)$ mit Knotenmenge V und Kantenmenge E definieren. Darüber hinaus existiert mit $v_{\text{root}} \in V$ ein ausgewezeichneter Wurzelknoten. Die Knoten der untersten Ebene werden als Blattknoten bezeichnet, die übrigen Knoten als innere Knoten. Die Kanten des Baums sind gerichtet und weisen stets in Richtung der Blattknoten.

Wie in der Einleitung bereits erwähnt, dienen Decision Trees der Entscheidungsfindung. Die graphische Darstellung des Baums kann als Hierarchie von

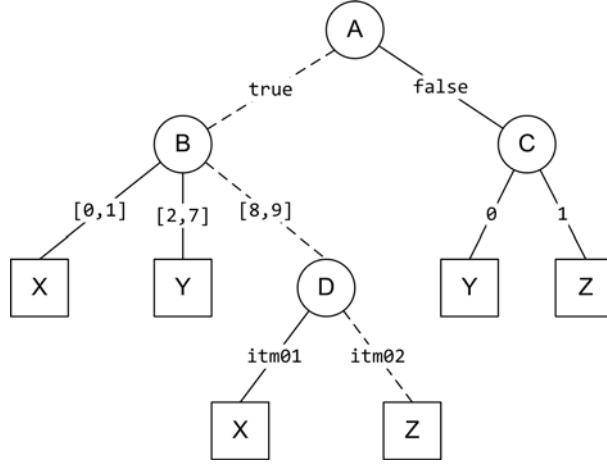


Abbildung 1: Aufbau eines einfachen Decision Trees mit Attributen A, B, C, D . Die gestrichelte Linie markiert den gewählten Pfad für die konkrete Eingabe ($A = \text{true}$, $B = 8$, $C = 1$, $D = \text{itm02}$).

Einzelentscheidungen angesehen werden. Ausgehend von der Wurzel wird an jedem inneren Knoten eine Entscheidung getroffen, welche den weiteren Weg im Baum vorgibt. Dieser Vorgang wird bis zum Erreichen eines Knotens mit maximaler Tiefe fortgesetzt, welcher letztendlich die Lösung des Problems enthält.

Formal beschrieben, repräsentiert jeder innere Knoten des Baums einen bedingten Test. Dazu tragen die Knoten Attribute, welche als Selektionskriterium dienen. Man spricht hier von *predictor* Variablen. Die ausgehenden Kanten symbolisieren dabei die jeweiligen Ausprägungen der Attribute. Zwar ist die Verwendung von Attributmengen, durch Konjunktion der Selektoren möglich, jedoch vergrößert dies die Zahl der nötigen Kanten drastisch. Kombinierte Attributtests ermöglichen weitaus mehr zu berücksichtigende Entscheidungen. Zudem unterscheidet sich das Ergebnis nicht vom „seriellen“ Vorgehen, welches daher vorgezogen werden sollte. Einen Sonderfall bilden schliesslich die Blätter des Baums, welche anstelle eines Attributs, eine Lösung des Problems enthalten. Diese werden in der Regel als *response* Variablen bezeichnet. Eine Lösung des Entscheidungsproblems ergibt sich damit als Pfad vom Wurzelknoten zu einem der Blätter. Dieser Pfad entspricht einer Konjunktion der bedingten Attributtests. Der Baum selbst kann dabei als Disjunktion der Konjunktionen verstanden werden, die alle möglichen Pfade verknüpft. Abbildung 1 zeigt den Aufbau eines

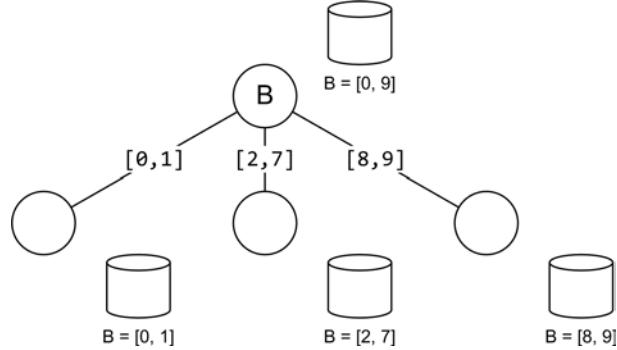


Abbildung 2: Zerlegung der Eingabemenge am Beispiel des Attributs B für eine Eingabe $B = [0, 9]$.

einfachen Decision Trees mit *predictor* Variablen

$$\begin{aligned} A &= \{\text{true, false}\} \\ B &= [0, 9] \\ C &= \{0, 1\} \\ D &= \{\text{itm01, itm02}\} \end{aligned}$$

und *response* Variable

$$\text{class} = \{x, y, z\}.$$

Die Eingabe des Baums erfolgt als Menge von Attributen und deren Ausprägungen, welche dem Baum an seiner Wurzel zur Verfügung steht:

$$\langle A = \{\text{true, false}\}, B = [0, 9], C = \{0, 1\}, D = \{\text{itm01, itm02}\} \rangle.$$

Der gewählte Pfad für die konkrete Eingabe

$$\langle A = \text{true}, B = 8, C = 1, D = \text{itm02} \rangle$$

ist als gestrichelte Linie dargestellt. Der Baum liefert hier die Klassifizierung Z . Das weitere Vorgehen des Decision Trees ähnelt einem Sortierprozess durch sukzessives Zerlegen der Eingabe in Teilmengen, gemäß der getroffenen Einzelentscheidungen an jedem der Knoten, siehe Abbildung 2.

Wie zu erkennen ist, können die Wertebereiche von *predictor* und *response* Variablen frei gewählt werden, sowohl diskrete als auch kontinuierliche Werte sind möglich. Die Wahl wirkt sich mitunter jedoch stark auf die Struktur des Decision Trees aus. Kontinuierliche Werte vergrößern die Zahl der möglichen Entscheidungen erheblich, was zu einem Aufblättern des Baums führt. Daher sollten Prädiktoren geschickt gewählt werden. Zudem lassen sich in Abhängigkeit der *response* Variablen zwei Arten von Decision Trees unterscheiden.

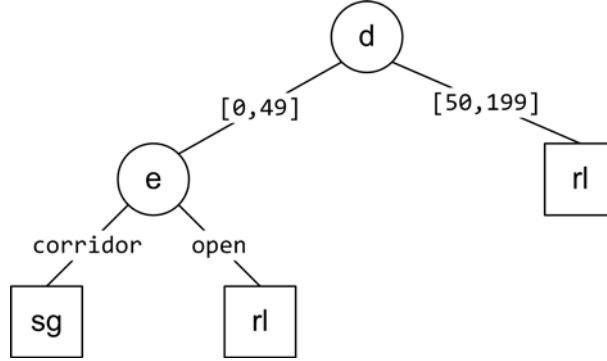


Abbildung 3: Beispiel eines *classification trees* zur Waffenwahl in 3D-Shootern mit Attributen d (distance) und e (environment). Die *response* Variable klassifiziert die Eingabe gemäß der zu verwendenden Waffen sl (*shotgun*) und rl (*rocket launcher*).

Classification Trees verwenden diskrete *response* Variablen und werden zur Klassifikation der Eingaben verwendet. Das Symbol der Rückgabe repräsentiert hier die spezifische, zugewiesene Klasse.

Regression Trees hingegen verwenden kontinuierliche *response* Variablen, in Form einzelner Werte oder Intervallen. Das Ergebnis kann als Durchschnittswert der Eingabe verstanden werden.

1.2 Anwendungsbeispiele

Agenten in Computerspielen müssen häufig in der Lage sein, eine Vielzahl von taktischen Entscheidungen treffen zu können. So auch die Agenten von 3D-Shootern. Ihre Entscheidungen und die daraus resultierenden Handlungen basieren dabei auf unterschiedlichen Faktoren, beispielsweise der eigenen Gesundheit, verfügbarer Ausrüstung oder aber den Gegebenheiten der Umgebung. Die Waffenwahl in Kampfsituationen zählt zu eben jenen taktischen Entscheidungen, die durch die Verwendungen von Decision Trees unterstützt werden können. Abbildung 3 zeigt die vereinfachte Darstellung eines Baums, welcher in ähnlicher Form in vielen Spielen genutzt wird.

Als Elemente der Eingabemenge stehen die Attribute

$$\begin{aligned} d &= [0, 200], \\ e &= \{\text{open}, \text{corridor}\} \end{aligned}$$

zur Verfügung, welche die Distanz zum Gegner, sowie die Art der Umgebung beschreiben. Als *response* Variable für die Ausgabe sei die Menge

$$\text{weapon} = \{\text{sg}, \text{rl}\}$$

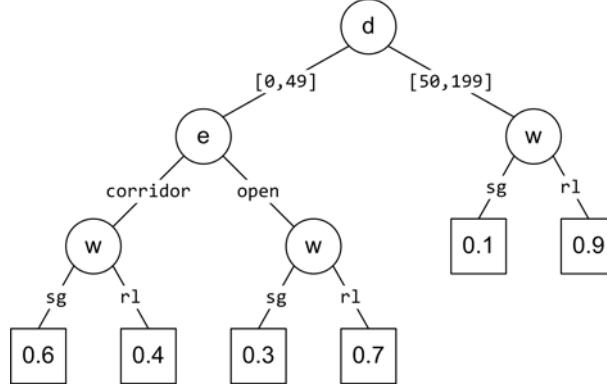


Abbildung 4: Beispiel eines *regression trees* zur Waffenwahl in 3D-Shootern mit Attributen d (distance), e (environment) und w (weapon). Die *response* Variablen liefern eine Bewertung der situationsabhängigen Waffenwahl des Agenten.

definiert, welche Symbole für die Waffen *shotgun* und *rocket launcher* enthält. Die Eingabe wird vom Entscheidungsbaum, einem *classification tree*, ausgewertet und in ein Ergebnis überführt. Das Ergebnis, als eindeutige Klassifizierung der Eingabe, repräsentiert schliesslich die zu verwendende Waffe. Für die Eingabe

$$\langle d = 30, e = \text{corridor} \rangle$$

liefer der, Baum so die Lösung *rl*, was der Verwendung des *rocket launcher*s entspricht.

Eine Alternative bietet die Verwendung eines *regression trees*. Dieser liefert keine eindeutige Klassifizierung, sondern eine durchschnittliche Bewertung der Eingabe. Er kann so dazu genutzt werden, eine Abschätzung der Güte der zu verwendenden Waffe, und damit der Taktik zu liefern. Abbildung 4 zeigt den modifizierten Entscheidungsbaum. Die Eingaben wurden um das Attribut

$$w = \{\text{sg}, \text{rl}\}$$

erweitert. Als neue *response* Variable mit kontinuierlichem Wertebereich wurde

$$\text{efficiency} = [0, 1]$$

zur Bewertung der Effizienz eingeführt. Im Fall der Eingabe

$$\langle d = 30, e = \text{open}, w = \text{sg} \rangle$$

wird diese vom Baum mit 0.3 bewertet. Bei Unterschreitung eines gegebenen Grenzwerts könnte dies den Agenten möglicherweise zur Wahl einer alternativen Taktik bewegen.

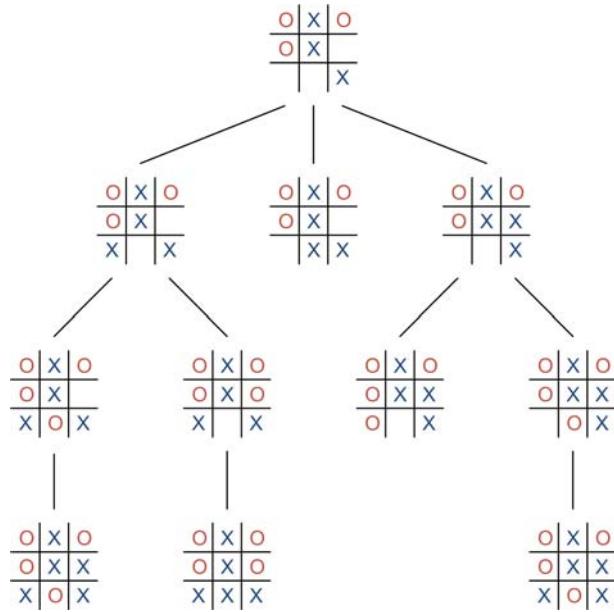


Abbildung 5: Spielbaum eines fortgeschrittenen Tic-Tac-Toe Spiels. Spieler X ist am Zug.

1.3 Game Trees

Game Trees, oder Spielbäume, sind eine besondere Art von Entscheidungsbäumen. Trotz der Komplexität des Themas, sollen sie dennoch an dieser Stelle, wenn auch nur oberflächlich, betrachtet werden.

Spielbäume finden sich in vielen KI-Systemen wieder. Vor allem Brettspielumsetzungen greifen häufig auf das Konzept der Game Trees zurück. Zu den prominentesten Beispielen gehören Spiele wie Dame, Schach, Othello und Go, aber auch das im weiteren Verlauf betrachtete Spiel Tic-Tac-Toe gehört dazu. Alle Spiele besitzen eine Gemeinsamkeit: Es handelt sich um 2-Personen Spiele mit perfekter Information. Die komplette Spielinformation ist jedem der Spieler zu jedem Zeitpunkt zugänglich. Dies ist eine notwendige Bedingung für die Verwendung von Spielbäumen, die so eine vollständige Abbildung aller Spielverläufe ermöglichen. Mit Hilfe des *minmax*-Algorithmus kann eine Bewertung der einzelnen Spielsituationen vorgenommen werden, welche als Gütekriterium einer zielorientierten Wahl der Folgezüge dient.

Der *minmax*-Algorithmus Der *minmax*-Algorithmus, als einer der wichtigen Spielbaumalgorithmen, soll hier anhand des Beispiels Tic-Tac-Toe die Anwendung der Game Trees in Computerspielen verdeutlichen. Abbildung 5 zeigt den Spielbaum aller möglichen Folgesituationen eines bereits fortgeschrittenen Spiels.

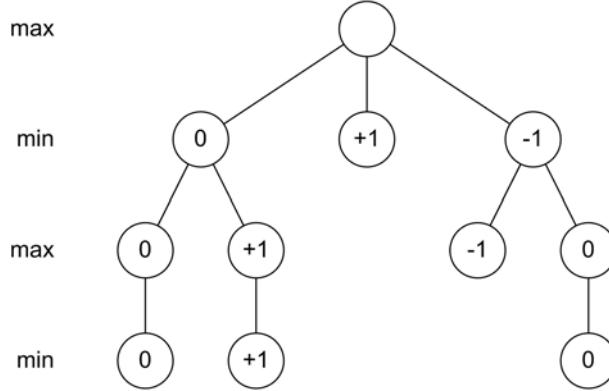


Abbildung 6: Bewerteter Spielbaum des Tic-Tac-Toe Spiels aus Abbildung 5.

Ausgehend von der Ausgangssituation an der Wurzel des Baums, verzweigt sich dieser an allen Knoten in die jeweils möglichen Folgesituationen, bis hin zu den Blättern, welche die Spieldurchgänge repräsentieren. Der Algorithmus versucht die, im Sinne des Spielers, „beste“ Spielstellung zu wählen. Dazu werden die einzelnen Spielzüge nach spielabhängigen Kriterien mit Punkten bewertet. Im Fall von Tic-Tac-Toe erfolgt die Bewertung nach einer einfachen Regel:

- +1 : X gewinnt,
- 0 : unentschieden / Spiel offen,
- 1 : O gewinnt.

Da beide Spieler das selbe Ziel verfolgen, versucht Spieler X seine Punkte zu maximieren, während Spieler O diese zu minimieren versucht. Die Spieler werden daher auch häufig *max (maximizer)* und *min (minimizer)* bezeichnet. Zum besseren Verständnis des Algorithmus werden zudem die Ebenen des Baums mit den Spielern gekennzeichnet. Dies ist ohne weiteres möglich, da beide Spieler ihre Züge abwechselnd durchführen.

Letztendlich ist nur der Spieldurchgang von Interesse, daher erfolgt die Bewertung der Spielstellungen *bottom-up*, indem der Baum in Postorder durchlaufen wird. Die Bewertung der Blätter ist trivial. Auf allen anderen Ebenen erfolgt die Bewertung nach einem einfachen Prinzip. Knoten, auf mit *max* gekennzeichneten Ebenen erhalten das Maximum der Punkte ihrer Kinder, die Knoten der *min*-Ebenen dementsprechend das Minimum der Punkte ihrer Kinder. Dieses Verfahren wird bis zum Erreichen der Wurzel fortgesetzt. Abschliessend kann der Baum erneut durchlaufen werden, diesmal *top-down*. Der Pfad folgt dabei, in Abhängigkeit des aktuellen Spielers, stets der Richtung des Knotens mit maximaler bzw. minimaler Bewertung. Abbildung 6 zeigt den Spielbaum nach vollständiger Bewertung.

Im Fall des Beispiels ist Spieler *max*, also X am Zug. Es stehen drei mögliche Spielzüge zur Verfügung. Der Zug mit Wert 0 liefert keine Spielentscheidung, beide Spieler können im weiteren Verlauf des Spiels den Sieg für sich verbuchen. Der Zug mit Wert -1 hingegen ermöglicht Spieler *min* im Anschluss, das Spiel für sich zu entscheiden und wird daher bei der Auswahl von Spieler *max* vernachlässigt. Der Zug mit Bewertung $+1$ schliesslich, führt direkt zu einem Sieg und besitzt somit die höchste Priorität.

Das hier vorgestellte Spiel Tic-Tac-Toe ist einfach gehalten und in seiner Komplexität stark eingeschränkt. Die Anwendung von Spielbäumen und des *minmax*-Algorithmus kann problemlos erfolgen. Dennoch zeigen sich an dieser Stelle mögliche Nachteile der Game Trees: Der Aufbau eines vollständigen Baums ist für viele Spiele nicht möglich. Dame, Schach oder Go bieten zuviele taktische Möglichkeiten um alle Situationen erfassen zu können. Im Fall von Go, bei einer Spielbrettgröße von 19×19 Feldern, besitzt der Baum bereits einen durchschnittlichen Verzweigungsgrad von 250. Bei einer durchschnittlichen Spieldauer von 150 Zügen ergibt dies annähernd 10^{360} mögliche Spielsituationen.

Eine mögliche Lösung des Problems ist, die Bäume nur bis zu einer fest definierten Tiefe aufzubauen. Die Blätter, die nun keine vollständigen Informationen über den Spieldurchgang tragen, müssen mit Hilfe geeigneter Heuristiken bewertet werden, um die Güte der gewählten Strategie bestimmen zu können. Andererseits imitiert gerade dies das Verhalten menschlicher Spieler, welche ebenfalls nur eine bestimmte Zahl von Zügen vorausschauend planen können. Eine weitere Möglichkeit bietet das *alpha-beta-Pruning*, eine optimierte Version des *minmax*-Algorithmus, welche von Russel und Norvig in [RN03] beschrieben wird.

1.4 Fazit

Aufgrund ihrer Flexibilität können *decision trees* auf eine große Zahl von Problemen angepasst werden. Die Eigenschaft beliebige Werte verarbeiten zu können, macht den Einsatz zur Klassifizierung oder zur Bewertung von Eingaben möglich. Die speichereffiziente Datenstruktur erlaubt den Einsatz in vielen Bereichen. Die zugehörigen Algorithmen erlauben zudem Echtzeit-Anwendungen.

Eine bedeutende Eigenschaft von *decision trees*, welche im Rahmen der Seminararbeit nicht behandelt wurde, ist die Fähigkeit des Lernens. Die hier vorgestellten Beispiele gehen von einer Konstruktion durch einen Experten aus. Dieser besitzt genügend Informationen über mögliche Eingaben sowie das gewünschte Verhalten der Bäume und kann diese gezielt modellieren. Jedoch können Entscheidungsbäume auch trainiert werden, um ein gewünschtes Verhalten zu zeigen. Dies setzt jedoch eine ausreichend große Menge von Trainingsdaten voraus. Die Eingaben werden um ein Attribut der gewünschten Klassifizierung erweitert, Algorithmen versuchen den Baum auf Basis der vorliegenden Daten bestmöglich zu konstruieren, so dass alle Eingaben gemäß ihrer gewünschten Klassifizierung erkannt werden. Dies kann unter Umständen jedoch auch Probleme mit sich bringen. Zu komplexe Baumstrukturen oder nicht ausreichend viele Trainingsdaten können das Training behindern und zu fehlerhaften Auswertungen führen. Zudem

lässt sich das Verhalten der trainierten Entscheidungsbäume im Vorfeld nur grob abschätzen. Ein Fehlverhalten zeigt sich hier erst im praktischen Einsatz.

2 Finite-State Machines

Finite-state Machines sind ein formales Konzept der Automatentheorie. Ihr Einsatz in Videospielen besteht in der Regel aus der Umsetzung diverser Kontrollstrukturen, wie sie zum Beispiel zur Verhaltensmodellierung virtueller Agenten genutzt werden. Dieser Anwendungsfall soll der folgenden Betrachtung als Grundlage dienen.

2.1 Definition

Grundsätzlich definiert sich eine Finite-state Machine über die Menge ihrer Eingaben Σ , die Menge ihrer Zustände Q und ihrer Transitionen, in Form einer Zustandsübergangsfunktion δ . Zusätzlich existiert mit $q_0 \in Q$ ein ausgezeichneter Startzustand. Bei den Mengen Σ und Q handelt es sich um endliche Mengen, woraus sich eine ebenfalls endliche Menge von Zustandsübergängen ergibt. Damit sind auch die Möglichkeiten der Finite-state Machine selbst beschränkt. Formal lässt sich eine Finite-state Machine damit durch das 4-Tupel

$$\text{FSM} = \{\Sigma, Q, q_0, \delta\}$$

darstellen. Je nach Definition und Typ der Finite-state Machine wird die formale Darstellung um weitere Symbole ergänzt.

Transduktoren Transduktoren bezeichnen eine Familie der Finite-state Machines, welche in der Lage sind, Eingabesequenzen auf Ausgabesequenzen abzubilden. Die sequentielle Generierung der Ausgabe erfolgt dabei auf Basis einer Ausgabefunktion λ . Die Art der Funktion zerlegt die Menge der Transduktoren zudem in zwei Klassen: Mealy-Automaten und Moore-Automaten.

Mealy-Automaten erzeugen Ausgaben stets in Abhängigkeit der Eingabe und des erreichten Folgezustands. Die Funktion zur Generierung der Ausgabe ist definiert als

$$\lambda_{\text{mealy}} : Q \times \Sigma \rightarrow \Delta \cup \{\epsilon\}.$$

Moore-Automaten hingegen erzeugen Ausgaben nur auf Basis des erreichten Folgezustands mit Ausgabefunktionen der Form

$$\lambda_{\text{moore}} : Q \rightarrow \Delta \cup \{\epsilon\}.$$

Wird λ der allgemeinen Definition der Finite-state Machines hinzugefügt, so ergibt sich die Definition eines Transduktors als:

$$\text{FSM}_{\text{transducer}} = \{\Sigma, \Delta, Q, q_0, \delta, \lambda\}.$$

Eine Eingabe $w = w_1 \dots w_l \in \Sigma^*$ wird vom Transduktoren zeichenweise verarbeitet, startend im Zustand q_0 . Bei aktuellem Zustand $q \in Q$ und Eingabesymbol $w_i \in \Sigma$ wechselt der Automat in den Zustand $\delta(q, w_i)$. Im Fall des Mealy-Automaten wird dabei die Ausgabe $\lambda(\delta(q, w_i), w_i)$, im Fall des Moore-Automaten die Ausgabe $\lambda(\delta(q, w_i))$ erzeugt.

Akzeptoren Einen besonderen Funktionstyp der Finite-state Machines bilden Akzeptoren. Sie können dazu genutzt werden, bestimmte Muster oder Sequenzen innerhalb der Eingabe zu erkennen. Daher werden sie auch häufig als *sequence detectors* bezeichnet. Im Gegensatz zu den Transduktoren wird die Ausgabe nicht mehr während der Verarbeitung der Eingabe erzeugt, sondern erst mit Erreichen eines ausgezeichneten Zustands. Das Erreichen solch eines *akzeptierenden* Zustands kann mit dem Akzeptieren der Eingabesequenz gleichgesetzt werden. Akzeptoren können somit als boolsche Funktionen verstanden werden, welche Eingaben auf ihre Gültigkeit hin überprüfen.

Die Definition eines Akzeptors ähnelt der allgemeinen Definition der Finite-state Machine. In der Regel wird allerdings auf die Angabe des endlichen Ausgabealphabets und der dazugehörigen Ausgabefunktion verzichtet, da lediglich der erreichte Zustand bei Terminierung interessiert. Zusätzlich wird mit F eine endliche Menge von akzeptierenden Zuständen eingeführt. Damit ergibt sich die formale Definition des Akzeptors als

$$\text{FSM}_{\text{acceptor}} = \{\Sigma, Q, q_0, \delta, F\}.$$

Eine Eingabe $w = w_1 \dots w_l \in \Sigma^*$ wird vom Akzeptor zeichenweise verarbeitet, startend im Zustand q_0 . Bei aktuellem Zustand $q \in Q$ und Eingabesymbol $w_i \in \Sigma$ wechselt der Automat in den Zustand $\delta(q, w_i)$. Wird nach Verarbeiten der Eingabe ein Zustand $q_t \in F$ erreicht, so wird die Eingabesequenz akzeptiert.

2.2 Einsatz in Spielen

Ein hohes Maß an Flexibilität ermöglicht es, das formale, mathmatische Konzept der Finite-state Machines auf den praktischen Einsatz in Computerspielen zu übertragen. Eingabe- und Ausgabealphabet werden dahingehend konkretisiert, dass beide spezifische Ereignisse der Spielumgebung repräsentieren. Die Implementierung reicht dabei von einfachen boolschen Kontrollfunktionen, bis hin zu komplexen, objektorientierten Ansätzen in Form von „Event-Systemen“, welche Ereignisse als eigenständige Objekte verwalten. Damit werden Zustandsübergänge der Finite-state Machines stets durch das Eintreten bestimmter Ereignisse ausgelöst. Darüber hinaus können die Zustände selbst als Ereignisse aufgefasst werden. Agenten, deren KI auf Finite-state Machines dieser Art basieren, werden daher häufig als *state-driven agents* bezeichnet. Letztendlich folgt das Konzept dem intuitiven Verständnis von Interaktion: Handlungen erzeugen Ereignisse, Ereignisse lösen Handlungen aus, Reaktionen als Antwort auf Aktionen.

Die nachfolgenden Beispiele sollen einen kleinen Überblick über die verschiedenen Einsatzmöglichkeiten von Finite-state Machines in Videospielen geben.

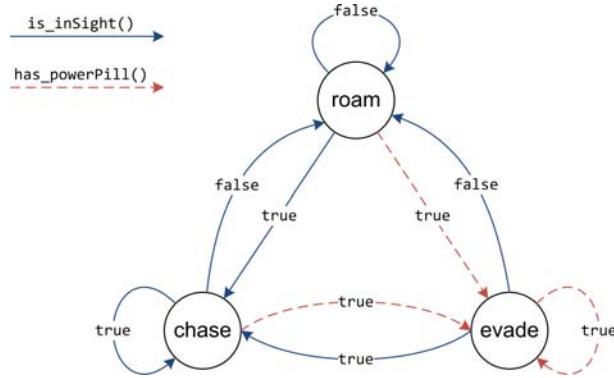


Abbildung 7: Darstellung der Geister-KI des Spiels Pac-Man als Finite-state Machine.

Agentensteuerung Häufig werden Finite-state Machines dazu genutzt, Verhaltensmuster zu modellieren. Je nach Komplexität der eingesetzten FSM können so Agenten mit einfachen oder aber stark ausgereiften Handlungsabläufen umgesetzt werden. Das hier vorgestellte Beispiel zeigt, wie Finite-state Machines im 1980 von Namco veröffentlichten Spiel Pac-Man zum Einsatz kommen. Die vier Agenten des Spiels Pac-Man sind darauf beschränkt durch das Spiellabyrinth zu navigieren und, abhängig von der Spielsituation, mit der Figur des menschlichen Spielers zu interagieren. Der KI des Spiels liegen dazu vier Finite-state Machines zugrunde, welche auf dem Prinzip der in Abschnitt 2.1 vorgestellten Transduktoren basieren.

Da die Möglichkeiten der KI im Spiel Pac-Man stark eingeschränkt sind, reichen bereits drei Zustände aus, um das Verhalten der Geister vollständig abzubilden. Die Zustände repräsentieren die unterschiedlichen Verhaltensmodi Jagen (*chase*), Fliehen (*evade*) und Suchen (*roam*). Das Verhalten der Geister ist dabei stets abhängig von der aktuellen Spielsituation. Die Transitionen werden daher durch zwei Ereignisse der Spielwelt, sowie deren Komplemente, ausgelöst. Damit ergibt sich schliesslich die in Abbildung 7 dargestellte Finit-state Machine.

Aufgrund der Definition des Moore-Automaten, welcher Ausgaben nur aufgrund des erreichten Zustands erzeugt, können die drei Zustände der Finite-state Machine als direkte Repräsentation der Handlungen des Agenten aufgefasst werden. Das Erreichen eines Zustands wird mit dem Erzeugen der Ausgabe gleichgesetzt. Aus Sicht der Implementierung bedeutet dies, den Aufruf spezieller Methoden des Agenten, welche letztendlich das gewünschte Verhalten umsetzen. Auch die Ereignisse, wie das Auftauchen des menschlichen Spielers in Sichtweite oder das Aufsammeln eines Power-Ups, welche für das Auslösen von Zustandsübergängen verantwortlich sind, werden auf Ebene der Implementierung durch Methodenaufrufe bzw. deren Rückgabewerte repräsentiert. Im Beispiel der Pac-Man KI existieren zu diesem Zweck die zwei boolschen Funktionen

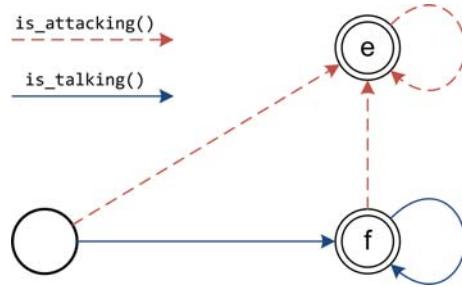


Abbildung 8: Akzeptor zur Freund/Feind Klassifikation von Agenten. Zustand *e* repräsentiert die Klassifizierung *enemy*, Zustand *f* die Klassifizierung *friend*.

`is_inSight()` und `has_powerPill()`. Das Eintreten von Ereignissen innerhalb der Spielwelt kann leicht überprüft werden. In positiven Fällen schaltet die FSM des Agenten dementsprechend, worauf hin dieser seinen Zustand wechselt.

Freund/Feind Klassifikation Die in Abschnitt 2.1 vorgestellten Akzeptoren können dazu genutzt werden, Agenten aufgrund ihrer Interaktion mit der Umgebung zu klassifizieren. Trifft ein Spieler auf einen neutralen Agenten, so wird dieser, auf Basis der Aktionen des Spielers, Position beziehen und sein Verhalten dementsprechend ändern. Dem Agenten liegt dazu, die in Abbildung 8 dargestellte FSM zugrunde. Aufgrund der Handlungen des Spielers kann der Agent diesen den Klassen *Freund* oder *Feind* zuordnen. Solange der Spieler kein aggressives Verhalten zeigt, und den Agenten attackiert, wird er als *Freund* angesehen werden. Sollte er jedoch versuchen den Agenten anzugreifen, wird dieser ihn als *Feind* erkennen und dementsprechende Reaktionen zeigen.

Ferner ist es möglich, dass mit Erreichen bestimmter Zustände, neue *events* ausgelöst werden, welche als externe Eingaben weiterer Finite-state Machines dienen. So könnte der Zustand `is_enemy()` den Agenten dazu veranlassen, in einen Angriffsmodus überzugehen oder die Flucht zu ergreifen und damit sein bisheriges Verhalten zu ändern. Dieses Konzept der Hierarchischen Finite-state Machines ermöglicht es, das Verhalten von Agenten modular zu gestalten. Zudem können die einzelnen Finite-state Machines stark spezialisiert werden.

2.3 Fazit

Die hier vorgestellten Finite-state Machines dienen häufig nur als Designkonzept. In der Regel erfolgt die Implementierung durch einfache Fallunterscheidungen in Form von **IF-THEN** Konstrukten. Nur in den seltesten Fällen kommen Automateninterpretier zum Einsatz. Ein Verhaltensmodell in Form einer FSM kann so leicht in Programmcode überführt werden. Daneben bieten Finite-state Machines noch eine Reihe weiterer Vorteile, die sie zu einem bevorzugten Werkzeug bei der Entwicklung von Spiele-KIs machen.

Finite-state Machines folgen einem einfachen, intuitiven Konzept. Zudem ermöglichen grafische Design-Werkzeuge die Modellierung auf einer Ebene, die keine Programmierkenntnisse voraussetzt. Ihre Flexibilität erlaubt es, das Verhalten eines Agenten durch Hinzufügen weiterer Zustände und entsprechender Transitionen zu erweitern. Das abzubildende Verhalten legt dabei auch die Komplexität der FSM fest. Zwar ist die Skalierbarkeit nicht beschränkt, jedoch wird das Design komplexer Systeme schnell unübersichtlich. Die Konzepte der hierarchischen Finite-state Machines oder Fuzzy-state Machines bieten zudem weitere, interessante Anwendungsfälle.

Das vielleicht größte Manko der Finite-state Machines und vieler klassischer KI-Methoden allgemein, ist die hohe Vorhersagbarkeit. Das deterministische Verhalten ist vor allem bei einfach gehaltenen Modellen, die nur wenige Zustände besitzen, leicht erkennbar. Zudem lässt sich das Design der Finite-state Machines während der Laufzeit nicht ändern. Ein dynamische, adaptives Verhalten lässt sich so nicht umsetzen.

3 Wegfindung mittels A^* -Algorithmus

Wegfindung ist eine der häufigen Aufgaben bei der Entwicklung Künstlicher Intelligenzen (kurz *KIs*) für Spiele. Besonders Spiele, deren virtuelle Welten von einer großen Zahl computergesteuerter Einheiten oder vom Spieler gesteuerte Einheiten bevölkert werden, kommen nicht um einen effektiv und effizient arbeitenden Wegfindungsalgorithmus herum. Dieses Kapitel befasst sich daher mit einem der am häufigsten angewandten Wegfindungsalgorithmus – dem A^* -Algorithmus.

Der A^* -Algorithmus, auch „A-Stern“ oder „A-Star“ genannt, wurde erstmals 1968 von Hart, Nilsson und Raphael in [HNR68] beschrieben. Er findet sich häufig in den für die Bewegung der Entitäten zuständigen Sub-Systemen einer KI wieder. Während dort auf Mikroebene Algorithmen für die eigentliche Durchführung der Bewegungen sowie die Koordination einzelner NPCs untereinander (vgl. Schwarmverhalten/Flocking) zuständig sind, arbeitet der A^* -Algorithmus auf Makroebene, in einer viel größeren Auflösung. So wird er für die Planung von Routen (im folgenden als *Pfade* bezeichnet) eingesetzt und ermöglicht die Umsetzung intelligenter und zielgerichteter Bewegungsabläufe.

3.1 Definition

Der A^* -Algorithmus ist ein Suchalgorithmus zur Berechnung kürzester Pfade in einem kantengewichteten Graphen $G = (V, E)$, mit Knotenmenge V und Kantenmenge E und gehört zur Klasse der Best-First-Search Algorithmen. Best-First-Search optimiert die Breitensuche, indem vielversprechende Knoten bei der Erforschung des Suchraums bevorzugt behandelt werden. A^* nutzt eine Fitnessfunktion $f(n)$ zur Bewertung der „Güte“ eines Knotens, indem die eigentlichen Kosten zum Erreichen des Knotens $g(n)$, sowie die ausstehenden Kosten des Pfades zum Ziel $h(n)$ betrachtet werden. Da letztere noch nicht bekannt sind

-
1. Sei s Startknoten
 2. Berechne $g(s)$, $h(s)$ und $f(s)$
 3. Füge s der OpenList hinzu
 4. Finde Knoten n aus OpenList mit kleinstem $f(n)$
 - Falls n Zielknoten, Abbruch da Pfad gefunden
 - Falls OpenList leer, Abbruch da kein Pfad gefunden werden kann
 5. Sei m gültiger, adjazenter Knoten von n
 - Berechne $g(m)$, $h(m)$ und $f(m)$
 - Falls m in ClosedList und neuer Pfad günstiger, Pfad aktualisieren
 - sonst m der OpenList hinzufügen
 - Wiederhole 5. für alle m
 6. Wiederhole 4.
-

Abbildung 9: Der A^* -Algorithmus in Pseudocode.

– schliesslich würde der Pfad schon die Lösung des Suchproblems darstellen – können diese nicht genau bestimmt werden. Daher bedient sich A^* einer Heuristik, um jene Kosten abzuschätzen. Formal lässt sich die Fitnessfunktion als Summe von $g(n)$ (vom englischen *goal*) und $h(n)$ (vom englischen *heuristic*) darstellen:

$$f(n) = g(n) + h(n).$$

Auf Basis dieser Bewertung breitet sich der Algorithmus stetig im Suchraum aus. Ausgehend vom Startknoten verlagert er sich in Richtung des fittesten Knotens. Hierzu protokolliert der Algorithmus seinen Fortschritt in zwei Listen. Die Liste **OPEN** enthält alle Knoten, die sich am Rand des vom Algorithmus untersuchten Gebiets befinden. Gleichzeitig stellen diese Knoten mögliche Kandidaten für die genauere Betrachtung dar. Die Liste kann als Priority-Queue angesehen werden, welche die eingetragenen Knoten gemäß ihrer Fitness absteigend sortiert. Die zweite Liste ist die sogenannte **CLOSED**-Liste. In ihr befinden sich alle bereits untersuchten Knoten, die einen Teil des derzeit kürzesten Pfades bilden. Für den Fall, dass ein kürzester Pfad zwischen Start- und Zielknoten existiert, wird die Liste nach Terminierung des Algorithmus durchlaufen, um den Pfad rekonstruieren zu können.

3.2 Eigenschaften

Den A^* -Algorithmus zeichnen mehrere Attribute aus, die ihn zu einem effizienten Werkzeug bei der Lösung von Suchproblemen machen. So arbeitet er zum einen *korrekt*, das Ergebnis des Suchlaufs ist stets eine gültige Lösung des Suchproblems, zum anderen arbeitet er *vollständig*. Sofern eine Lösung für das Suchproblem existiert, wird diese vom Algorithmus gefunden. Darüber hinaus arbeitet der Algorithmus *optimal*, liefert als Ergebnis des Suchproblems also stets einen

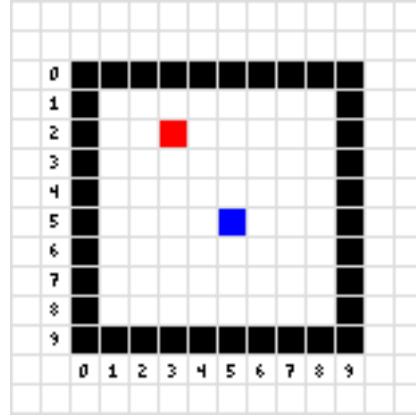


Abbildung 10: Einfaches 2D-Raster mit Startpunkt (blau), Zielpunkt (rot) und Barrieren (schwarz)

kürzesten Pfad. Allerdings ist dieses Verhalten stark von der verwendeten Heuristik zur Abschätzung der verbleibenden Kosten abhängig. Vereinfacht gesagt, muss die Abschätzung optimistisch ausfallen, und darf die verbleibenden Kosten nie überschätzen. Ein Beweis hierfür findet sich zum Beispiel in [RN03].

Die Heuristik bietet, neben der Kostenfunktion, viele Möglichkeiten zur Anpassung und Feinabstimmung des Algorithmus und sollte problemabhängig und gut überlegt gewählt werden. Zwar mag dies auf den ersten Blick einen Kritikpunkt des Algorithmus darstellen, gleichzeitig verleiht es ihm dadurch aber ein hohes Maß an Flexibilität (siehe 3.4).

3.3 Veranschaulichung am Beispiel

Anhand eines einfachen Beispiels soll die Funktionsweise des A^* -Algorithmus veranschaulicht werden. Als Ausgangssituation sei das in Abbildung 10 dargestellte 2D-Raster gegeben. Die Aufgabe des A^* -Algorithmus besteht darin, einen kürzesten Pfad vom Startpunkt zum Zielpunkt zu berechnen. Die in Abschnitt 3.1 beschriebenen Grundlagen für die Anwendung des Algorithmus können leicht herbeigeführt werden.

Die Spielumgebung lässt sich, aufgrund der Raster-Struktur, leicht durch einen Graphen darstellen. Jeder Knoten des Graphen repräsentiert dabei eine Kachel des Rasters, während Kanten Nachbarschaftsrelationen einzelner Kacheln repräsentieren. Ausgehend von der Position besitzt jede „innere“ Kachel somit acht, jede „Randkachel“ fünf und jede „Eckkachel“ drei Nachbarn. Zusätzlich enthalten die Knoten Informationen über die Position der Kachel in Form von Koordinatenpaaren.

Die Kosten für das Betreten einer Kachel werden konstant auf 1 gesetzt. Damit fällt jede einzelne Kachel eines Pfades gleich schwer ins Gewicht. Natürlich

ist dieser Fall stark vereinfacht und lässt keine Bewertung in Abhängigkeit des zugrundeliegenden Terrains oder der Umgebung zu. Da es sich aber um ein simples Beispiel handelt, genügt die gewählte Funktion der Veranschaulichung. Für einen Knoten n und seinen Nachbarn m ergibt sich damit folgende Kostenfunktion:

$$g(m) = g(n) + 1.$$

Als geeignete Heuristik bietet sich der Euklidische Abstand an. Der Euklidische Abstand ist eine Metrik, welche die Distanz zwischen zwei Punkten a und b wie folgt definiert:

$$d(a, b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}.$$

Die Funktion d liefert eine gute, optimistische Abschätzung der Kosten. Das Ergebnis wird häufig kleiner als die tatsächlich verbleibenden Kosten ausfallen, diese aber in keinem Fall übertreffen. Für solch eine Heuristik arbeitet der A^* -Algorithmus optimal und liefert als Lösung einen kürzesten Pfad, sofern dieser existiert. Für die weitere Betrachtung soll daher der Euklidische Abstand als Heuristik $h(n)$ dienen. Damit sind die Grundvoraussetzungen für die Anwendung des A^* -Algorithmus geschaffen.

Der weitere Ablauf folgt dem, in Abbildung 9 dargestellten Algorithmus und wird hier nur grob beschrieben. Die Knoten werden durch ihre eindeutigen Koordinatenpaare bezeichnet. Aufgrund der verwendeten Kostenfunktion betragen die Kosten für einen Knoten stets 1. Damit ist die Wahl des Knotens n mit kleinstem $f(n)$ stets abhängig von $h(n)$, also der Abschätzung der verbleibenden Kosten bis zum Erreichen des Zielknotens. Wie sich leicht überprüfen lässt, wird in der ersten Iteration Knoten $(4, 4)$ mit $f(n) = 3,236$ gewählt und in **CLOSED** aufgenommen. Die Nachbarknoten werden, sofern nicht schon geschehen, in **OPEN** aufgenommen. Dies geschieht analog für Knoten $(3, 3)$ mit $f(n) = 3$. In der dritten Iteration wird schliesslich Knoten $(3, 2)$ gewählt. Da es sich um den Zielknoten handelt, terminiert der Algorithmus letztendlich und rekonstruiert den berechneten Pfad (siehe Abbildung 11). Hierzu wird die Liste **CLOSED** durchlaufen, die nun alle Knoten des berechneten kürzesten Pfads enthält.

3.4 Anpassung der Kostenfunktion

Die Möglichkeit der Anpassung von Kostenfunktion und Heuristik verleihen dem A^* -Algorithmus ein hohes Maß an Flexibilität. Durch eine Modifizierung der Kostenfunktion kann zudem ein natürlicheres Verhalten bei der Suche kürzester Pfade erreicht werden. So lässt sich der Algorithmus auf eine große Anzahl von Situationen übertragen. Konstante lokale Kosten, also Kosten für die Verwendung eines Knotens, und eine einfache Kostenfunktion wurden für das Beispiel aus Abschnitt 3.3 bewusst gewählt. Sie dienten der Veranschaulichung der Funktionsweise des Algorithmus, ihre Praxistauglichkeit ist jedoch eher fragwürdig.

Die Wegfindung in Spielen basiert auf einer Vielzahl unterschiedlicher Faktoren. Die Art des Terrains beispielsweise hat großen Einfluss die Bewegungsge-

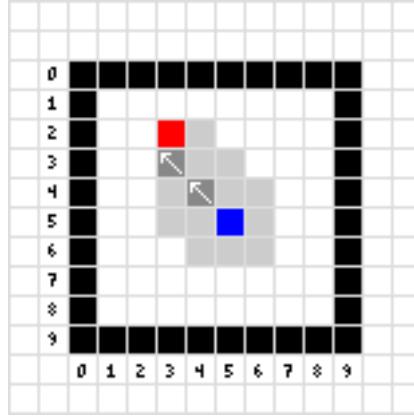


Abbildung 11: Ergebnis des A^* Suchlaufs. Die graue Fläche zeigt den vom Algorithmus betrachteten Suchraum. Pfeile markieren den vom Algorithmus gewählten kürzesten Pfad.

schwindigkeit der in Spielen gesteuerten virtuellen Agenten. Dichter und unwegsamer Untergrund schränkt die Bewegungen erheblich ein, daher sollte dies als Faktor in die Bewertung mit einfließen. Sei `costs()` eine Methode zur Berechnung der lokalen Kosten eines Knotens, so lässt sich die Kostenfunktion auf das gewünschte Verhalten anpassen:

$$g(m) = g(n) + \text{costs}(m).$$

Die Kosten für einen Pfad durch Knoten m setzen sich nun aus den Kosten für den Pfad durch Knoten n sowie den jeweiligen lokalen Kosten für die Verwendung von m zusammen, welche als eine Art Strafterm aufsummiert werden.

Die neue Kostenfunktion erzeugt bereits ein ausgereifteres Suchverhalten des Algorithmus. Pfade werden weiterhin auf Basis ihrer *absoluten* Kosten verglichen. Dies kann in manchen Anwendungsfällen jedoch nicht ausreichen. Sollen beispielsweise Knoten ähnlicher „Güte“ bei der Berechnung eines Pfads präferiert werden, so muss die Kostenfunktion weiter modifiziert werden. Ein Beispiel für die Verwendung von *relativen* Kosten sei der Fall, dass sich die lokalen Kosten eines Knotens durch seine Höhenlage in der Spielwelt errechnen: Je höher die Lage umso niedriger die Kosten. Sei `costs()` wie zuvor eine Methode zur Bestimmung der lokalen Kosten, so kann die Kostenfunktion folgendermaßen modifiziert werden:

$$g(n) = \begin{cases} g(m) + 1, & |\text{height}(m) - \text{height}(n)| \leq 5 \\ g(m) + 10, & |\text{height}(m) - \text{height}(n)| > 5 \end{cases}.$$

Abbildung 12 zeigt den Vergleich zweier A^* -Suchläufe unter Verwendung der absoluten und relativen Kosten.

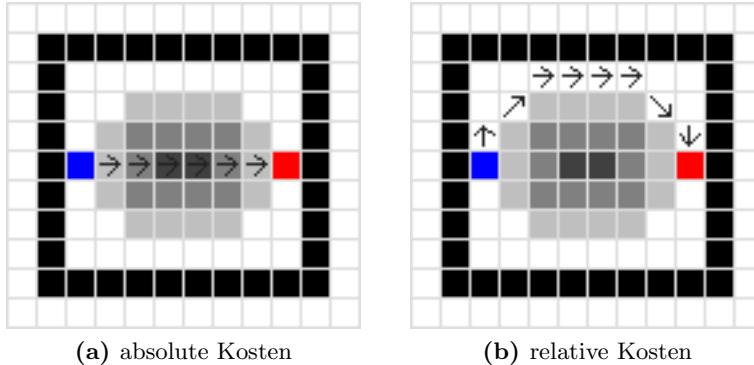


Abbildung 12: Vergleich des A^* -Algorithmus auf Basis absoluter Kosten (a) und relativer Kosten (b). Die Kosten der einzelnen Kacheln ergeben sich aus ihrer Höhenlage in der Spielwelt. Je dunkler die dargestellte Fläche, umso geringer die Höhen.

3.5 Fazit

Das hier gezeigte Beispiel ist simpel gehalten und dient nur der Veranschaulichung der Funktionsweise. Seine wahre Stärke zeigt der A^* -Algorithmus, wenn die Bedingungen verschärft werden, und Hindernisse und terrainabhängige Kosten ins Spiel kommen.

Obwohl der A^* -Algorithmus zu einem der effizientesten Suchalgorithmen gehört, sollte er dennoch gut überlegt angewandt werden. Aufgrund der verwendeten Listen kann der Speicherverbrauch, je nach Spielumgebung, immense Dimensionen annehmen. Daneben spielen natürlich Laufzeit und vorhandene CPU-Kapazität eine große Rolle.

Auch wenn der A^* -Algorithmus hier am Beispiel der Wegfindung beschrieben wird, ist er nicht auf dieses Einsatzgebiet beschränkt. Er lässt sich auf alle Suchprobleme übertragen, deren Lösung ein kürzester Pfad darstellt. So kann der Algorithmus beispielsweise für die Berechnung kürzester Pfade in *Tech-Trees* angewandt werden um die kurz möglichste, und damit günstigste Baureihenfolge von Gebäuden festzulegen.

Als größtes Manko des Algorithmus sei zuletzt die Situation genannt, in dem kein Pfad zwischen Start- und Zielknoten existiert. In diesem Fall wird der Algorithmus alle erreichbaren Knoten untersuchen und letztendlich terminieren.

Literatur

- [AI] AI HORIZON: *AI Horizon: Computer Science and Artificial Intelligence Programming.* <http://www.aihorizon.com>. – Stand: August, 2007
- [Bau] BAUR, Stefan K.: *Pfadsuche-Applet.* <http://www.stefan-baur.de/cs.web.mashup.pathfinding.html>. – Stand: August 2007

- [BS04] BOURG, David M. ; SEEMANN, Glenn: *AI For Game Developers*. O'Reilly, 2004
- [Buc05] BUCKLAND, Mat: *Programming Game AI By Example*. Wordware Pub., 2005
- [Cha] CHAMPANDARD, Alex J.: *AI-Depot.com*. <http://ai-depot.com>. – Stand: August 2007
- [Cha03] CHAMPANDARD, Alex J.: *AI Game Development : Synthetic Creatures With Learning And Reactive Behaviors*. New Riders, 2003
- [DeL01] DELOURA, Mark A.: *Game Programming Gems 2*. Charles River Media, 2001
- [DeL03] DELOURA, Mark A.: *Game Programming Gems*. Charles River Media, 2003
- [HNR68] HART, Peter ; NILSSON, Nils ; RAPHAEL, Bertram: A Formal Basis for the Heuristic Determination of Minimum-Cost Paths. In: *IEEE Transactions on Systems Science and Cybernetics SSC-4* (1968), Nr. 2, S. 100–107
- [Mit97] MITCHELL, Tom M.: *Machine Learning*. McGraw Hill, 1997
- [Pat] PATEL, Amit J.: *Amit's Game Programming Information*. <http://www-cs-students.stanford.edu/~amitp/gameprog.html>. – Stand: August 2007
- [Rab] RABIN, Steve: *AIWisdom.com*. <http://www.aiwisdom.com>. – Stand: Juli, 2007
- [Rab02] RABIN, Steve: *AI Game Programming Wisdom*. Charles River Media, 2002
- [Rab04] RABIN, Steve: *AI Game Programming Wisdom 2*. Charles River Media, 2004
- [RN03] RUSSELL, Stuart J. ; NORVIG, Peter: *Artificial Intelligence : A Modern Approach*. Prentice Hall, 2003

Evolutionäre Algorithmen in Computerspielen

„Computational Intelligence in Games“

Frank Behler

Universität Dortmund, 44221 Dortmund, Germany,
frank.behler@cs.uni-dortmund.de,
WWW home page: <http://ls11-www.cs.uni-dortmund.de/>

Zusammenfassung Künstliche Intelligenz in Computerspielen spielt eine immer größere Rolle. In dieser Arbeit werden grundlegende Konzepte von evolutionären Algorithmen vorgestellt, welche dabei helfen können, einen künstlichen Spieler im Umgang mit schwierigen Spielsituationen zu verbessern. Beispiellohaft wird ein Computergegner eines Echtzeitstrategiespiels, der mit dynamischen Skripten arbeitet, durch evolutionäre Algorithmen verbessert. Der evolutionäre Algorithmus entwickelt hierbei gegen besonders ausgereifte Strategien eine bessere Taktik. Die auf diese Weise gewonnenen Erkenntnisse werden so verallgemeinert, dass eine Strategie entworfen werden kann, die das ursprüngliche Skript ersetzt oder verbessert.

1 Einleitung

In modernen Computerspielen ist neben einer guten Graphik immer häufiger auch die Spielstärke der künstlichen Intelligenz (KI) ein Aushängeschild der Produktqualität [3]. Häufig ist es, aufgrund der sehr hohen Komplexität und Möglichkeiten, jedoch schwierig, alle Spielsituationen vorherzusehen. Besonders für diese Fälle ist es sehr wichtig, dass eine KI sich den einzelnen Strategien des Spielers anpassen kann, um das Ausnutzen offensichtlicher Schwächen durch den Spieler zu verhindern. Meistens sind Abläufe in Computerspielen über so genannte Skripte kodiert, die eine mögliche Abfolge der auszuführenden Befehle enthalten. Diese einzelnen Möglichkeiten der Entscheidungen können in ihrer Wahrscheinlichkeit mit Gewichten versehen werden, um sie der Spielstärke oder auch der Taktik des Gegners anzupassen. Um neue Strategien zu entwickeln, können mit Hilfe evolutionärer Algorithmen neue Regeln für die Regelbasis gefunden werden. Aus Erfahrungen, die in Spielen gegen Spezialtaktiken erworben werden können neue Regeln extrahiert und in die Regelbasis aufgenommen werden. Die KI spielt in diesem Falle in einem Offline Lernverfahren gegen sich selbst und probiert neue Strategien aus von denen erfolgreiche gespeichert oder mit Hilfe von Evolutionären Algorithmen weiter verbessert werden. Evolutionäre Algorithmen sind eine Klasse von Such- und Optimierungstechniken, die von der darwinistischen Evolution inspiriert sind. Das Konzept umfasst in sogenannten Chromosomen kodierte Probleme bei denen natürliche Vorgänge wie Crossover, Mutationen und natürliche Selektion durch den Computer simuliert werden, um

zu einer Lösung zu gelangen. Die Problematik bei der Implementierung ist die Kodierung der Spielstrategien in den Chromosomen und die Variation der Strategien sowie die Auswahl der Regeln, welche zur Erweiterung der alten Regelbasis genutzt werden sollen.

2 Evolutionäre Algorithmen

2.1 Einführung

Evolutionäre Algorithmen sind eine zu den probabilistischen Optimierungsverfahren gehörende Klasse von Algorithmen. Sie verbessern initiale Lösungen für ein gegebenes Problem, in Anlehnung an die natürliche Evolution, indem neue potenzielle Lösungen erzeugt und verworfen werden. Ihr Ursprung lässt sich nicht eindeutig auf eine Person oder einen Zeitpunkt festlegen, da verschiedene Verfahren, die sich an natürliche Evolution anlehnen, unabhängig voneinander entstanden sind. So entwickelten Fogel 1962 die „evolutionäre Programmierung“ (EP)[5], Holland 1962 die „genetischen Algorithmen“ (GA) [4] und Rechenberg und Schwefel 1965 „Evolutionsstrategien“ (ES)[6][7]. Während ursprünglich in der EP als Lösungen für das gegebene Problem endliche Automaten gesucht werden, sind GA und ES auf allgemeinere Probleme mit reelwertigen oder kombinatorischen Lösungen anwendbar. Sie unterscheiden sich insbesondere in der Repräsentation der Lösungen des Problems: Während in ES diese nicht vorgegeben ist, sondern abhängig vom Problem gewählt wird, ist sie bei GA immer ein Vektor von Binärwerten. Es existieren jedoch noch weitere Unterschiede zwischen den Algorithmen. Allgemein werden alle diese und auch aus ihnen hervorgehende Verfahren unter dem Sammelbegriff „Evolutionäre Algorithmen“ (EA) zusammengefasst. Somit umfasst der Begriff EA auch Algorithmen, die sich nicht eindeutig der EP, den GA oder den ES zuordnen lassen. Die Grenzen zwischen den einzelnen Kategorien der evolutionären Algorithmen verwischen aber zunehmend und es ist keine klare Trennung mehr vorhanden.

Die Grundstruktur eines EA lässt sich durch das wiederholte Ausführen einer Schleife beschreiben, in der nach einem einmaligen initialen Erzeugen von Startlösungen für das jeweilige Problem, innerhalb des Algorithmus als „Individuen“ bezeichnet, eine Bewertung nach den vom Problem definierten Maßstäben berechnet wird. Aus der vorhandenen Menge von Lösungen (Population) werden anschließend Gruppen definierter Größe ausgesucht, aus welchen durch einen als „Rekombination“ bezeichneten Schritt neue Individuen erzeugt werden. Daraufhin werden die neu erzeugten Individuen ebenfalls in Anlehnung an natürliche Evolution verändert, dies wird als „Mutation“ bezeichnet. Die so geschaffenen Individuen werden erneut auf ihre Qualität hin bewertet, diese Auswertung wird auch als Auswertung der „Fitnessfunktion“ bezeichnet. Die neuen Individuen mit dem „besten“ Fitnesswert werden als neue Population selektiert. Je nach Art des EA geschieht dies auch einschließlich derer der vorhergegangenen Generation. Ist ein definiertes Terminierungskriterium, wie zum Beispiel eine vorher festgelegte Anzahl von Schleifendurchläufen, erreicht, so wird der Algorithmus

beendet. Ansonsten beginnt der Schleifendurchlauf von neuem. Der Ablauf wird in Abbildung 1 grafisch dargestellt.

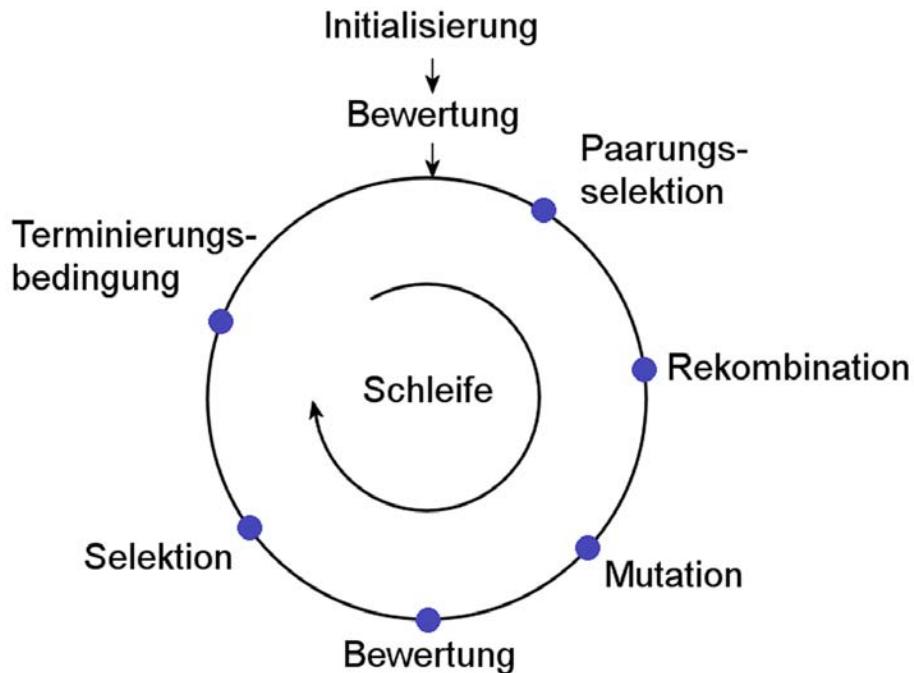


Abbildung 1. Darstellung des Ablaufs eines evolutionären Algorithmus. Nach der Initialisierung einer Startpopulation wird diese zunächst bewertet und durchläuft so lange die EA-Schleife, bis eine zuvor vorgegebene Terminierungsbedingung erfüllt ist. Während des Durchlaufs werden Eltern für die Rekombination selektiert. Die neuen entstandenen Individuen mutiert und anschließend vor der Auswahl der nächsten Generation bewertet.

Ein nach diesem Schema aufgebauter Algorithmus bietet gegenüber deterministischen Algorithmen, wie beispielsweise gradientenbasierten Verfahren, den Vorteil, dass er bei der Suche im Lösungsraum lokale Minima überwinden kann und außerdem aufgrund der recht allgemeinen Formulierung vielseitig und robust anwendbar ist. Diese Eigenschaften sind bei vielen in der Praxis relevanten Problemen sehr vorteilhaft, weshalb evolutionäre Algorithmen bei vielen verschiedenen Problemstellungen angewendet werden können. Zur genaueren Beschreibung der einzelnen Schritte werden im Folgenden Evolutionsstrategien als spezielle Variante der evolutionären Algorithmen anhand des Artikels von Beyer und Schwefel [8] genauer erläutert.

2.2 Der ES-Basisalgorithmus

Um den Ablauf eines evolutionären Algorithmus genauer beschreiben zu können, sind eine Vielzahl von Parametern notwendig. Für eine ES sind einige davon:

- μ : Anzahl Individuen in der Population.
- λ : Anzahl erzeugter Individuen pro Generation.
- ρ : Anzahl der Individuen, die für das Erzeugen eines neuen Individuums benutzt werden.
- \mathbb{P}_p : Population, aus der neue Individuen erzeugt werden („Elternpopulation“).
- \mathbb{P}_o : Population neu erzeugter Individuen („Nachkommen“).

Wird in den Schleifendurchläufen der ES \mathbb{P}_p stets durch eine Teilmenge von \mathbb{P}_o ersetzt, so spricht man von einer „Komma-Strategie“. Sofern Individuen aus \mathbb{P}_p auch im nächsten Schleifendurchlauf existieren, wird von einer „Plus-Strategie“ gesprochen. Als verkürzende Schreibweise wird $(\mu/\rho, \lambda)$ bzw. $(\mu/\rho + \lambda)$ benutzt. Als Verallgemeinerung der Selektionsstrategie kann auch ein Parameter κ eingeführt werden, der die Anzahl der Generationen angibt, die ein Individuum in die nächste Generation übernommen werden kann. Für den Parameter κ entspricht $\kappa = 0$ der „Komma-Strategie“ und $\kappa = \infty$ der „Plus-Strategie“. Die einzelnen Individuen a_i einer Population lassen sich als

$$a_i := (\mathbf{y}_i, \mathbf{s}_i, F(\mathbf{y}_i)) \quad (1)$$

beschreiben, wobei \mathbf{y}_i die eigentliche Lösung des Problems enthält (auch als Objektparameter bezeichnet), während \mathbf{s}_i Strategieparameter enthält, die Einfluss auf den Ablauf des Algorithmus, insbesondere die Mutation, nehmen können. Diese Parametereinstellungen können bei verschiedenen Individuen unterschiedlich sein und während des Algorithmus verändert werden, weshalb sie auch als endogene Strategieparameter bezeichnet werden. $F(\mathbf{y}_i)$ in Formel 1 ist der Fitnesswert, der sich durch die Funktion F berechnen lässt. Dieser bestimmt die Qualität der von a_i repräsentierten Lösung. Der Ablauf einer allgemeinen ES wird im Algorithmus 1 in Pseudocode dargestellt. Die „marriage-Funktion“ wählt ρ Individuen aus \mathbb{P}_p und erzeugt aus diesen durch die Rekombinationsoperationen ein neues Individuum. Die Auswahl der Elternindividuen verläuft dabei zufällig gleichverteilt. Das neue Individuum wird nach Anwendung von Mutationen in \mathbb{P}_o eingefügt. Die Realisierung der Mutationsoperatoren sowie die Festlegung der Selektions- und Terminierungskriterien sind dabei nicht festgelegt, sondern sollten problemspezifisch gewählt werden.

2.3 Rekombination

Zur Rekombination von Individuen sind zwei Arten von Rekombinationsoperatoren, diskrete und intermediäre Rekombination, gebräuchlich. Bei der intermediären Rekombination werden alle Elternindividuen gleichermaßen berücksichtigt, indem das neue Individuum durch Berechnung der eventuell gewichteten Mittelwerte

$$r = \frac{1}{\rho} \sum_{m=1}^{\rho} (x_m)$$

```

 $g := 0;$ 
initialize  $\left( \mathbb{P}_p^{(0)} := \left\{ \left( \mathbf{y}_m^{(0)}, \mathbf{s}_m^{(0)}, F(\mathbf{y}_m^{(0)}) \right) \mid m = 1, \dots, \mu \right\} \right);$ 
repeat
  for  $l := 1$  to  $\lambda$  do
     $\mathbb{E}_l := \text{marriage}\left( \mathbb{P}_p^{(g)}, \rho \right);$ 
     $\mathbf{s}_l := \text{s-recombination}(\mathbb{E}_l);$ 
     $\mathbf{y}_l := \text{y-recombination}(\mathbb{E}_l);$ 
     $\tilde{\mathbf{s}}_l := \text{s-mutation}(\mathbf{s}_l);$ 
     $\tilde{\mathbf{y}}_l := \text{y-mutation}(\mathbf{y}_l, \tilde{\mathbf{s}}_l);$ 
     $\tilde{F}_l := F(\tilde{\mathbf{y}}_l);$ 
  end
   $\mathbb{P}_o^{(g)} := \left\{ \left( \tilde{\mathbf{y}}_l, \tilde{\mathbf{s}}_l, \tilde{F}_l \right), l = 1, \dots, \lambda \right\};$ 
  case selection-type =  $(\mu, \lambda)$ 
     $\mathbb{P}_p^{(g+1)} := \text{selection}\left( \mathbb{P}_o^{(g)}, \mu \right);$ 
  end
  case selection-type =  $(\mu + \lambda)$ 
     $\mathbb{P}_p^{(g+1)} := \text{selection}\left( \mathbb{P}_o^{(g)}, \mathbb{P}_p^{(g)}, \mu \right);$ 
  end
   $g := g + 1;$ 
until termination-condition ;

```

Algorithmus 1 : Der ES-Basisalgorithmus von Bayer und Schwefel [8]

erzeugt wird. Dies ist für reellwertige Suchräume problemlos möglich. In der diskreten Rekombination wird jeder Wert a_i^j eines neu erstellten Individuums a_i aus genau einem Elternindividuum entnommen. Das Elternindividuum wird dabei zufällig aus allen ρ Elternindividuen ausgewählt. Diese Methode findet, aufgrund der Struktur der Individuen, auch in GA Verwendung.

2.4 Mutation

Die Funktionsweise eines Mutationsoperators ist problemabhängig. Abgesehen von Restriktionen des Lösungsraumes muss problemspezifisches Wissen in die Entwicklung eines erfolgreichen Operators einfließen. Mutationsoperatoren sollten folgende drei Anforderungen erfüllen:

Erreichbarkeit: Jedes Individuum kann durch eine endliche Anzahl von Anwendungen des Operators zu jedem Punkt des Lösungsraumes mutieren.

Fairness: Die Mutation hat keine Tendenz in Richtung eines Optimums sondern eine Verbesserung des Fitnesswertes ist genau so wahrscheinlich wie eine Verschlechterung.

Skalierbarkeit: Die Mutationsstärke kann angepasst werden, sodass die Individuen sich bezüglich ihres Fitnesswertes stark oder schwach verändern.

Das Einhalten der genannten Forderungen ist zwar empfehlenswert, aber nicht immer zwingend notwendig. Abhängig von der Problemstellung kann auch ein nicht vollständiges Erfüllen der Forderungen zu guten Ergebnissen der ES führen. Für die in der Mutation verwendeten Zufallszahlen sollte in reellwertigen Lösungsräumen eine Normalverteilung und in ganzzahligen Lösungsräumen eine geometrische Verteilung benutzt werden.

2.5 Selektion

Im Unterschied zur Mutation erfolgt die Selektion gerichtet. Es werden immer die Individuen mit den besten Fitnesswerten ausgewählt. Dabei wird zwischen Komma- und Plus-Selektion unterschieden. Da bei der Verwendung der Komma-Selektion die Elternindividuen verworfen werden, muss hier $\lambda > \mu$ gelten. Der Fall $\lambda = \mu$ würde zu einer rein zufälligen Suche führen. Bei der Plus-Selektion wird immer das beste Individuum beibehalten. Selektionsmethoden mit dieser Eigenschaft werden auch als elitär bezeichnet. Für die Größe von λ gibt es dabei keine allgemeinen Richtlinien oder Einschränkungen. Plus-Selektion wird für diskrete Lösungsräume wie bei kombinatorischen Optimierungsproblemen empfohlen, während Komma-Selektion bei unbeschränkten Lösungsräumen, insbesondere \mathbb{R}^n , empfehlenswert ist. Bei allgemeinen EA sind auch andere Selektionsmethoden üblich. So wird beispielsweise eine Turnierselektion benutzt, die zufällig gleichverteilt eine Menge von Individuen auswählt, aus der diejenigen mit dem besten Fitnesswert selektiert werden.

2.6 Anpassung von Strategieparametern

Eine ES kann viele Parameter enthalten, die Einfluss auf den Ablauf des Algorithmus haben. Einer der wichtigsten ist die Mutationsstärke σ , welche die Fortschrittsrate φ , also die Geschwindigkeit mit der der Algorithmus gegen ein lokales Optimum konvergiert und die Wahrscheinlichkeit I der Erzeugung eines verbesserten Individuums entscheidend beeinflusst. Dabei gelten die Aussagen 2 und 3[8].

$$\sigma \rightarrow 0 : I \rightarrow 1/2, \varphi \rightarrow 0 \quad (2)$$

$$\sigma \rightarrow \infty : I \rightarrow 0, \varphi \rightarrow 0 \quad (3)$$

Experimente mit einer $(1+1)$ -ES und verschiedenen Fitnessfunktionen ergaben, dass φ maximal wird, wenn die gemessenen Werte von I bei ungefähr $1/5$ liegen. Daraus ergibt sich die so genannte „1/5-Regel“ die besagt, dass für maximales φ die Mutationsstärke so gewählt werden sollte, dass I bei $1/5$ liegt. Ihre Anwendung ist allerdings nur im Spezialfall der $(1+1)$ -ES sinnvoll. Sie kann wie in Algorithmus 2 beschrieben zur Anpassung von σ benutzt werden. Neben der 1/5-Regel existieren zahlreiche weitere Verfahren zur Anpassung von σ und weiterer Strategieparameter, die sich nicht nur auf $(1+1)$ -ES beschränken. Einige dieser Verfahren, wie die Selbstadaptation, werden im Artikel von Beyer und Schwefel [8] beschrieben.

1. $(1+1)$ -ES über G Generationen ausführen,
 σ bleibt konstant,
zähle G_s : Anzahl verbesserter Individuen.
2. $I := G_s/G$
3. $\sigma := \begin{cases} \sigma/c, & \text{wenn } I > \frac{1}{5} \\ \sigma \cdot c, & \text{wenn } I < \frac{1}{5} \\ \sigma, & \text{wenn } I = \frac{1}{5} \end{cases}$ mit einer Konstanten c : $0.85 \leq c < 1$.
4. Gehe zu 1.

Algorithmus 2 : Die 1/5-Regel

3 Beispielanwendung bei Computerspielen

3.1 Verbessern des Computergegners bei einem Echtzeitstrategiespiel

Der folgende Abschnitt erläutert Beispielhaft den möglichen Einsatz evolutionsärer Algorithmen zur Verbesserung der KI eines Computergegners bei einem Echtzeitstrategiespiel und bezieht sich größtenteils auf die Arbeit von Marc Ponsen [3] und seinen Artikel bei der „International Conference on Computer Games“ 2004 [2]. Für seine Arbeit stellte sich Ponsen die Frage, inwieweit evolutionäre Algorithmen neue Taktiken und Strategien für Echtzeitstrategiespiele entdecken können und ob neue „offline“ entdeckte Taktiken und Strategien die Qualität der ursprünglichen dynamischen Regelbasis verbessern können. Für seine Experimente wählte er das Open Source Spiel „Wargus“, welches auf der „Stratagus“-Engine aufsetzt. Wargus ist ein Klon des bekannten Spiels Warcraft II, welches zwar grafisch nicht mit heutigen Spielen mithalten, dessen Gameplay aber immer noch als aktuell bezeichnet werden kann. Ein Screenshot des Spiels befindet sich in Abbildung 2. Spiele KI für komplexe Spiele wird meistens über so genannte Skripte definiert, die eine Liste von Regeln sind, die nacheinander ausgeführt werden. Der Nachteil an dieser Methode ist häufig, dass aufgrund der hohen Komplexität schwächen enthalten sind, die von Spielern leicht ausgenutzt werden können, um das Spiel zu gewinnen. Ponsen bedient sich deshalb der „dynamischen Skripte“, welche eine Adaption der KI während der Laufzeit des Spiels ermöglichen und bereits erfolgreich in kommerziellen Rollenspielen eingesetzt wurden. Da die KI von Wargus in Skripten definiert ist, ließ sich dieses Verfahren leicht übernehmen.

Online Lernen Online Lernen ermöglicht der KI, nach Veröffentlichung des Spiels Schwächen der Skripte, die von Spielern ausgenutzt werden, automatisch zu verbessern und sich an die Taktiken und den Spielstil des Spielers anzupassen. Online Lernen kann entweder überwacht oder unüberwacht stattfinden. Beim überwachten Online Lernen, welches von Ponsen nicht benutzt wurde, ist es nötig dass der Spieler nach jeder Partie bewertet, wie erfolgreich der Computerspieler gewesen ist. Wichtige Aspekte hierbei sind die Geschwindigkeit, um keine

Verzögerung beim Spielfluss zu haben, und die Robustheit, um die vielen Zufallsfaktoren im Spiel berücksichtigen zu können. Außerdem sollte der Algorithmus so effizient sein, dass Verbesserungen nach nur wenigen Spielen zu erkennen sind und diese natürlich dann auch mindestens so erfolgreich sind wie handgeschriebene Skripte. Eines dieser unüberwachten Online Lernverfahren ist das dyna-



Abbildung 2. Screenshot aus dem Warcraft II Klon Wargus. Auf dem Bild erkennt man eine kleine Basis eines Spielers mit fünf verschiedenen Gebäudetypen, einem Arbeiter und einer Kampfeinheit.

mische Skripten. Aufgrund der nichtdeterministischen Art von Computerspielen ist es nötig eine große Menge bereichsspezifischen Wissens zu benutzen, welches normalerweise über deterministische Experimente erhalten wird. Die Regeln die in einem Skript benutzt werden, ergeben sich hierbei aus einer adaptiven Regelbasis, welche nur von Hand erstellte Regeln enthält. Die Wahrscheinlichkeit zur Nutzung einer Regel ergibt sich über einen Gewichtswert, der jeder Regel zugeordnet ist. Diese Gewichte ändern sich abhängig von dem Erfolg, den die Benutzung der einzelnen Regeln in einem Skript erbracht hat.

Offline Lernen Im Gegensatz zum Online lernen passt sich die KI beim Offline Lernen ohne menschliches Eingreifen beim Spielen gegen sich selbst an. Dies ist der Hauptunterschied der beiden Lernverfahren. KI Verbesserungen sind aufgrund der großen Menge an Möglichkeiten häufig sehr problematisch. Abhängig von der Menge der Parameter kann es zu einer unmöglichen Aufgabe werden innerhalb einer kurzen Zeit alle möglichen Szenarien durchzutesten. Ein Offline Lernalgorithmus bietet hier die Möglichkeit sehr viel mehr Variationen zu testen, als es ein einzelner Entwickler könnte. Eine interessante Anwendung für Offline Lernen ist das Entwickeln neuer Strategien für eine Gegner KI, indem der Prozess des dynamischen Skripts durch neu entdeckte Taktiken verbessert wird, welche dann in die Regelbasis aufgenommen werden. Auf diese Weise kann die KI auch gegen Spielertaktiken bestehen, die die Entwickler nicht vorhergesehen haben und deshalb keine Regel als Gegenmaßnahme in die Regelbasis eingetragen haben.

3.2 Dynamische Skripte in Wargus

Dynamische Skripte starten mit dem zufälligen Selektieren einer Regel für den ersten Zustand. Falls die so gewählte Regel zu einem Zustandsübergang führt, können von nun an nur noch Regeln des neuen Zustands ausgewählt werden. Um monotonen Verhalten zu verhindern entschied sich Ponsen dazu, jede Regel in jedem Zustand nur einmal ausführen zu können. Sobald das Ende des Skripts erreicht wird, sorgt eine Schleife dafür, dass fortgesetzt Angriffe auf den Gegner gestartet werden. Wie in den meisten Echtzeitstrategiespielen, bestimmen auch in Wargus die bereits errichteten Gebäude, welche Einheitentypen erstellt und welche Technologien erforscht werden können. Die entsprechende Zustandsbasierte Baureihenfolge für die Gebäude findet sich in Abbildung 3.

Hieraus ergibt sich, dass Zustandsübergänge dadurch erzeugt werden, dass durch das Ausführen bestimmter Regeln neue Gebäudetypen errichtet werden. Die Anfängliche Regelbasis für Wargus enthielt 50 Regeln, welche in allen Zuständen zu finden sind. Aufgrund der Notwendigkeit jederzeit die Möglichkeit haben zu können, Angriffe zu führen und eine gute Verteidigung der eigenen Basis zu besitzen, waren die Hälfte der Regeln militärische Regeln (siehe Tabelle 1). Eine vollständige Aufzählung und Erläuterung aller Regeln findet sich in [3].

Regeltyp	Beispiel	Anzahl
Bauregeln	Erstellen von Barracks	12
Ökonomieregeln	Bau neuer Arbeiter zum Rohstoffabbau	4
Militärregeln	Angriff auf den Gegner	25
Forschungsregeln	Aufrüsten der Waffen	9

Tabelle 1. Verhältnis von Regeln der einzelnen Typen zu Beginn des Trainings.

Für die Erstellung dieser Regeln verwandte Ponsen bereichsspezifisches Wissen aus Strategie-Guides für Warcraft II. Eine typische Regel erlaubt dem dy-

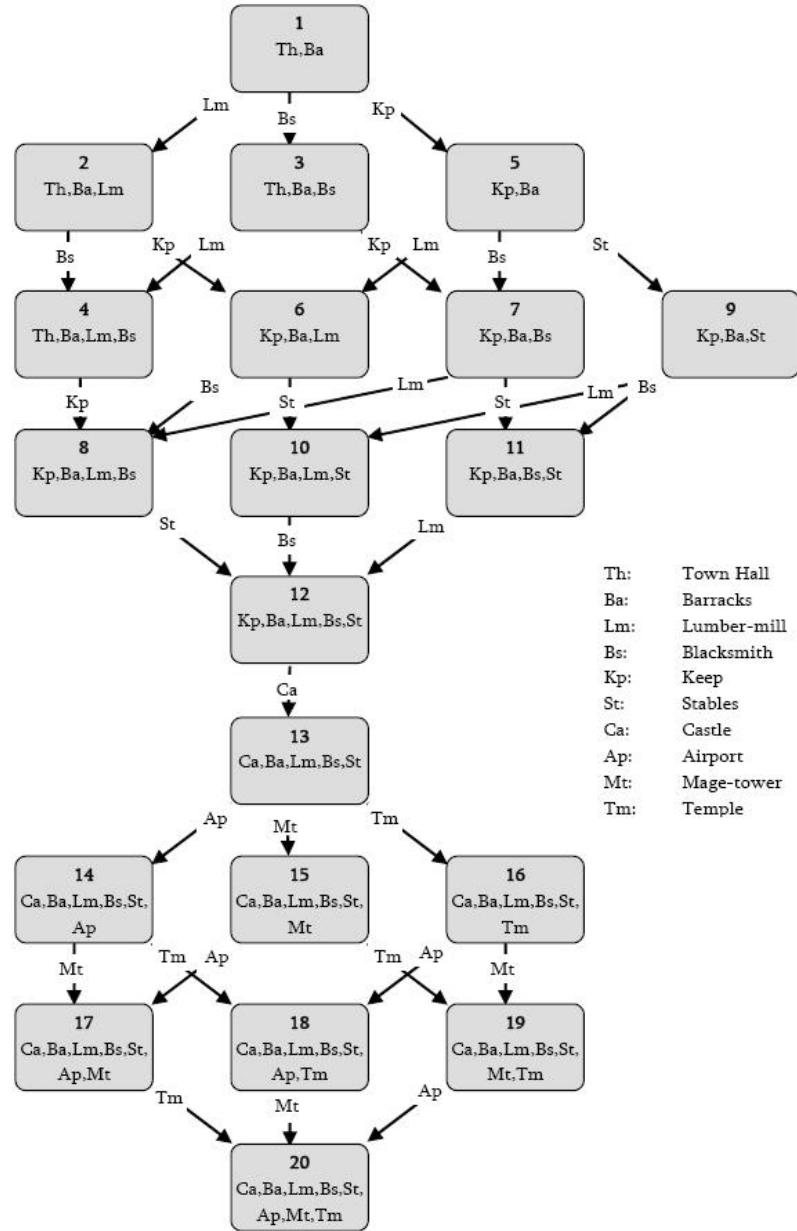


Abbildung 3. Spielzustände für Wargus. Die Kästchen stehen für die Zustände in denen die Gebäude eingetragen sind, die der Spieler bereits errichtet hat. Die Pfeile repräsentieren die Zustandsübergänge. Beim Bau einer „Lumbermill“ in Zustand 1 würde der Spieler Beispielsweise in Zustand 2 übergehen. „Town Hall“ und „Barracks“ sind bei dem Versuchsaufbau von Ponsen Startgebäude, die jeder Spieler zu Anfang des Spiels besitzt.

namischen Spieler hierbei einen Angriff auf den Gegner. Das bereichsspezifische Wissen liegt dann darin, dass es in Wargus ratsam ist, immer mit den fortschrittlichsten Einheiten anzugreifen. Eine andere Art des spezifischen Wissens ist die Tatsache das es wichtig ist, mehr als eine „Barracks“ (ein Gebäude in dem Einheiten ausgebildet werden) zu errichten, wohingegen der Bau einer zweiten „Blacksmith“ (ein spezielles Forschungsgebäude) nicht von nutzen ist, weshalb die KI daran gehindert wird. Die Wahrscheinlichkeit für das Ausführen einer Regel ist abhängig von der Gewichtung der Regel in dem entsprechenden Zustand. Bei 20 Zuständen und einer Regelbasis der Größe 50 wären das insgesamt 1000 Gewichte, welche ein schnelles Lernen unmöglich machen würden. Unter der Berücksichtigung, dass nicht alle Regeln in jedem Zustand ausgeführt werden können, grenzt sich die Anzahl der Regeln pro Zustand auf etwa 30 ein. Das Minimum liegt bei 21 und das Maximum bei 42 je nach Zustand. Auf diese Weise sucht die KI nur angemessene Regeln für die Verbesserungen aus.

Die einzelnen Gewichte der Regeln werden über eine Aktualisierungsfunktion angepasst, welche auf zwei anderen Funktionen beruht. Die Fitnessfunktion evaluiert den Erfolg des ganzen Spiels und die Zustandsfitnessfunktion berücksichtigt die während des Spiels besuchten Zustände. Der durch dynamische Skripte kontrollierte Spieler hat bei einem Fitnesswert von unter 0.5 normalerweise das Spiel verloren und mit einem Fitnesswert von über 0.5 gewonnen. Umso näher die Fitness bei 0 liegt, umso eindeutiger war die Niederlage wohingegen ein Fitnesswert nah bei der 1 einen eindeutigen Sieg repräsentiert. Im Gegensatz zu der allgemeinen Fitness gibt die Zustandsfitness nicht einen Wert für Sieg oder Niederlage wieder, sondern die Veränderung des Spielstands in einem bestimmten Zustand. Die Fitnessfunktion ist definiert als:

$$F = \begin{cases} \min\left(\frac{S_d}{S_d + S_o}, b\right), & \text{falls d verloren} \\ \max\left(\frac{S_d}{S_d + S_o}, b\right), & \text{falls d gewonnen} \end{cases}$$

S_d ist die Punktezahl des dynamischen Spielers und S_o die Punktezahl des Gegners. $b \in [0, 1]$ ist der Wert für ein Unentschieden, bei dem die Gewichte unverändert bleiben. Die Zustandsfitness F_i für Zustand $i \in [0, 20]$ wird definiert als:

$$F_i = \begin{cases} \frac{S_{d,i}}{S_{d,i} + S_{o,i}}, & i = 1 \\ \frac{S_{d,i}}{S_{d,i} + S_{o,i}} - \frac{S_{d,i-1}}{S_{d,i-1} + S_{o,i-1}}, & i > 1 \end{cases}$$

$S_{d,i}$ ist der Punktestand des Spielers nach Zustand i . Entsprechend ist $S_{o,i}$ die Punktezahl des Gegenspielers nach Zustand i . Die Punkte ergeben sich aus den im Spiel angezeigten Punkten für die unterschiedlichen Bereiche. Um diese nach ihrer Relevanz zu gewichten, ergab sich Ponsens Bewertung zu 70% aus militärischen und zu 30% aus Gebäudepunkten.

Nach Ende jedes Spiels wurden die Gewichte mit Hilfe der berechneten Fitnesswerte angepasst. Die Gewichtsfunktion W für den dynamischen Spieler ist definiert als:

$$W = \begin{cases} \max(W_{min}, W_{org} - 0.3 \frac{b-F}{b} P - 0.7 \frac{b-F_i}{b} P), & \text{falls } F < b \\ \min(W_{max}, W_{org} + 0.3 \frac{F-b}{1-b} R + 0.7 \frac{F_i-b}{1-b} R), & \text{sonst} \end{cases}$$

W ist das neue Gewicht, welches sich mit obiger Formel berechnen lässt. W_{org} ist der ursprüngliche Gewichtswert, P die maximale Strafe und R die maximale Belohnung. W_{max} und W_{min} bilden jeweils die obere und untere Grenze für die Gewichtsgrößen. Bei dieser Formel liegt eine stärkere Gewichtung in den Erfolgen der einzelnen Zustände als in dem Gesamterfolg. Hierdurch wird vermieden, dass erfolgreiche Regeln durch den Verlust eines Spiels zu stark bestraft werden.

Anwendung Evolutionärer Algorithmen in Echtzeitstrategiespielen

Das größte Problem beim Design von evolutionären Algorithmen für Echtzeitstrategiespiele ist die Kodierung der Informationen und die Auswertung der Fitness der einzelnen Lösungen. Die Kodierung sollte so gehalten sein, dass alle möglichen Lösungen repräsentiert werden können, unzulässige Lösungen jedoch ausgeschlossen werden. Auf diese Weise wird dem EA eine hohe Freiheit bei der Regelauswahl gegeben und gleichzeitig verhindert, dass unzulässige Regeln ausgeführt werden. Ponsen löst dieses Problem dadurch, dass er die Spielzustände einzelnen Mengen von Regeln zuweist, welche der EA frei austauschen kann.

Eine entsprechend genaue Fitnessfunktion ist essenziell, damit der EA effektiv arbeiten kann. Eine bessere Lösung sollte immer mit einer höheren Fitness bewertet werden. Eine ausreichend gute Problemspezifikation, in diesem Fall also das Besiegen der gegnerischen Einheiten auf einer gegebenen Karte, sollte so gestaltet sein, dass ein überwältigender Sieg mit einem höheren Fitnesswert belohnt werden sollte als ein nur knapper Sieg. Ein letzter Punkt der noch zu beachten ist, ist die Populationsgröße. Da die Bauprioritäten bei vielen Echtzeitstrategiespielen von entscheidender Bedeutung sind, sollte eine Population groß genug sein, um genügend Variationen für die Tests der einzelnen Strategien zu beinhalten. Bei einer zu kleinen Population könnte leicht eine Konvergenz in ein schlechtes lokales Optimum erfolgen.

Bei der Initialisierung des EA wird eine feste Menge an zufälligen Lösungen generiert. Neue Lösungen spielen dann gegen eine statische KI und ihr Erfolg wird gemessen. Sobald die Population generiert und die einzelne Fitness bewertet ist, erfolgt eine Rekombination der erfolgreichen Lösungen. Im nächsten Schritt wählt der EA einen seiner Operatoren und führt diesen aus. Die Evolutionsschleife wird danach solange durchgeführt bis das Abbruchkriterium erfüllt ist[3].

Kodierung der Lösungen Die Gene der einzelnen Chromosomen sind in Zuständen gruppiert. Ein Zustand wird dann aktiviert sobald die KI mindestens ein Gen dieses Zustands ausgeführt hat. In allen Chromosomen ist mindestens Zustand 1 aktiviert abhängig von der Baupriorität ,siehe Abbildung 3. Bei Wargus unterscheiden sich vier verschiedene Gentypen für Bauen, Forschung, Ökonomie und Kampf.

Bauregeln beginnen bei der Kodierung mit dem Buchstaben „B“ gefolgt von einer Zahl zwischen 1 und 12, welche die unterschiedlichen Gebäudetypen repräsentiert. Forschungsgene, die für das entdecken neuer Technologien und den Fortschritt innerhalb des Spiels benötigt werden beginnen mit einem „R“ (research) gefolgt von „13“ bis „21“. Ökonomische Gene beginnen mit einem „E“ (economy) gefolgt von einer Zahl welche als Parameter (Beispielsweise für die Anzahl der erstellten Arbeiter) verwendet wird. Militärische Aktionen sind in den „Kampfgenen“ kodiert. Sie beginnen mit einem „C“ gefolgt von einer Zahl, die dem derzeitigen Zustand entspricht. Der erste darauf folgende Parameter ist eine der maximal zehn möglichen von Wargus unterstützten Armeen (0 – 9) und der letzte Parameter bestimmt, ob sich die Armee defensiv oder eher offensiv verhält. Die anderen Parameter variieren je nach Zustand. Regeln im ersten Zustand verfügen über nur einen Parameter, welcher die Anzahl der Soldaten angibt, wohingegen im letzten Zustand bis zu sechs verschiedene Parameter auftauchen können. Eine komplette Übersicht aller Kodierungen findet sich in [3].

Auswertung der Fitnessfunktion Um den Erfolg einer KI zu bewerten, die durch ein Chromosom kodiert ist, verwendet Ponsen folgende Fitnessfunktion F für einen dynamischen Spieler d im Intervall $[0, 1]$:

$$F = \begin{cases} \min(b, \frac{GC}{EC} \cdot \frac{M_d}{M_d + M_o}), & \text{falls } d \text{ gewonnen} \\ \max(b, \frac{M_d}{M_d + M_o}), & \text{falls } d \text{ verloren} \end{cases}$$

M_d sind die militärischen Punkte des dynamischen Spielers und M_o die des Gegenspielers. GC repräsentiert die Spieldauer und EC die maximale Spieldauer nach der ein Spiel abgebrochen wird. Ist die maximale Spieldauer erreicht wird das Spiel abgebrochen und der derzeitige Punktestand ausgelesen. Die Spielzeit wird deshalb mit in Betracht gezogen, weil eine KI, die den Kampf zwar verliert, aber länger gegen den Gegner durchhält, höher bewertet werden sollte als eine KI, die das Spiel schneller verliert.

Operatoren Operatoren werden häufig so entworfen, dass sie stark an das Problem angepasst sind. Ponsen hat für seine KI in Wargus vier verschiedene Operatoren entworfen. Einer davon ist eine Rekombination, zwei Operatoren sind Mutationen und ein Operator erstellt zufällig ein völlig neues Individuum. Pro Generation wird ein Operator aufgerufen. Der letzte Operator wird mit 10 prozentiger Wahrscheinlichkeit aufgerufen. Die anderen drei mit jeweils 30% Pon.

Rekombination: Für die Rekombination werden zwei Eltern ausgewählt, die mindestens drei übereinstimmende aktivierte Zustände besitzen. Das Chromosom des erzeugten Nachkommen wird so aufgebaut, dass Gene von beiden Elternteilen übernommen werden, um zu verhindern, dass nur einer der Eltern kopiert wird. Zwischen zwei übereinstimmenden Zuständen werden alle Gene jeweils eines Elternteils in den Nachkommen kopiert. Auf diese Weise

wird verhindert, dass unzulässige Individuen entstehen. Nach dem letzten gemeinsamen Zustand werden wiederum alle Gene von jeweils einem Elternteil in den Nachkommen übernommen.

Regelmutation: Wähle ein Individuum und ersetze jede Forschungs-, Ökonomie- und Kampfregel mit einer Wahrscheinlichkeit von 25% in einem aktvierten Zustand. Bauregeln sind in diesem Fall als Ersatz ausgeschlossen, da diese zu einem Zustandswechsel führen könnten und so das Chromosom defekt wird. Gene in nicht aktiven Zuständen werden ignoriert, da diese nicht ausgeführt werden und als derzeit inaktiv gelten.

Mutation benutzter Regeln: Wähle ein Individuum und mutiere in jedem aktiven Zustand die Parameter einer Kampf- oder Ökonomieregel mit 50%. Die Mutation findet innerhalb der vorgegebenen Grenzen des Regeltyps statt. Bei diesem Mutationstyp sind Bau- und Forschungsregeln nicht betroffen. Ebenso wird dieser Mutationstyp nicht in inaktiven Genen ausgeführt.

Zufallserzeugung: Es wird ein komplett neues Chromosom für das Individuum erstellt.

Selektion Für die Selektion wurde eine Turnierselektion implementiert. Die Turnierselektion sucht zufällig M „Siegerchromosomen“ aus den N aus, um die neue Generation zu bilden. Umso höher der Wert von N umso größer ist der Selektionsdruck und umso niedriger das N umso größer wird die Vielfalt der Lösungen. Diese Methode führt wahrscheinlicher zu guten Ergebnissen und verhindert eine zu frühe Konvergenz. Neben der Turnierselektion existieren viele andere Selektionsverfahren, Ponsen empfindet die Turnierselektion jedoch als geeignet, da sie leicht zu implementieren ist und es ihm möglich ist durch geringen Selektionsdruck dafür zu sorgen, dass die Population eine große Variation besitzt[2]. In seiner Implementierung vergleicht er jeweils drei zufällige Individuen von denen das jeweils Beste in die nächste Generation übernommen wird.

Abbruchkriterium Wenn der Fitnesswert einen bestimmten Zielwert überschreitet, ist eine gewünschte Lösung gefunden. In Wargus ist eine Fitnessbewertung von 0.7 ein nahezu eindeutiger Sieg. Da es keine Garantie gibt, dass eine solche Lösung gefunden wird, ist als zusätzliches Abbruchkriterium ein Generationszähler eingebaut. Sobald eines der beiden Abbruchkriterien erreicht ist, wird das aktuelle Ergebnis gespeichert und der Lauf von neuem gestartet.

Ergebnisse des evolutionären Algorithmus Als statische KI wählt Ponsen die zwei spezialisierten Taktiken „Soldier Rush“ und „Knight Rush“ welche mit der Ursprüngliche KI nicht geschlagen werden konnten. Die Populationsgröße wurde auf 50 festgelegt, was ausreichend groß für den nicht sonderlich großen Suchraum in Wargus ist. Das Abbruchkriterium wurde für „Sodier Rush“ auf 0.75 und für „Knight Rush“ auf 0.7 festgelegt. Der zweite Wert ist deshalb etwas kleiner, da es auf größeren Karten länger dauert, sich an eine Gegnertaktik anzupassen und sich „Soldier Rush“ Spiele sehr schnell entscheiden. Die Anzahl der

Generationen wurde auf 250 festgelegt, da vorherige Versuche ergeben haben, dass bereits in dieser Zeit gute Ergebnisse zu erwarten sind.

Fast alle Experimente endeten vor der 250. Generation durch den Abbruch über die ausreichend hohe Fitness. Die direkte Schlußfolgerung hieraus ist, dass es möglich ist mit evolutionären Algorithmen neue erfolgreiche Taktiken und Strategien zu entdecken, um gegen optimierte Gegnertaktiken zu gewinnen gegen die das normale dynamische Skripten verliert. Das Ergebnis der Durchläufe ist in Tabelle 2 dargestellt.

KI-Typ	Kleinstes Wert	Höchster Wert	Durchschnitt	250
Soldier Rush	0.73	0.85	0.78	2
Knight Rush	0.71	0.84	0.75	0

Tabelle 2. Fitnesswerte der von dem EA gefundenen Lösungen gegen die spezialisierten Taktiken. Die Spalte „250“ gibt an wie viele der zehn Läufe nach 250 Generationen noch keine Lösung gefunden haben.

Ein überraschendes Ergebnis bei dem Versuch war, dass einige erfolgreiche vom EA entdeckte Strategien im Gegensatz zu den zuvor gesammelten Strategietipps aus dem Expertenwissen standen. So waren beispielsweise Katapulte wegen ihres Flächenschadens erfolgreich gegen große Angreifermengen, obwohl Katapulte im allgemeinen in ihrem Preis-Leistungs-Verhältnis als nutzlos angesehen werden. Spezifischere Informationen zu den Strategien in Wargus finden sich in [3] und [2].

Verbesserung der Regelbasis des dynamischen Skripts Zur Verbesserung der ursprünglichen Regelbasis werden die Erkenntnisse aus den Ergebnissen des Evolutionären Algorithmus genutzt. Auf diese Weise soll erreicht werden, dass die nun allgemein gehaltene KI, die nicht nur gegen die speziellen Taktiken optimiert ist mit diesen dennoch zurecht kommt - mindestens jedoch so gut wie die ursprüngliche KI. Ponsen untersuchte alle gefundenen Strategien aus seinen Versuchen und veränderte die Regelbasis mit fünf neuen durch den EA gefundenen Regeln:

1. Ein offensichtlicher Zug gegen den Soldier Rush, der von fast allen Strategien genutzt wurde, war das frühe Bauen einer „Blacksmith“ und die Erforschung besserer Waffen, wodurch es möglich war, den Gegner mit stärkeren Einheiten zu besiegen. Dieses Verhalten wurde als neue Regel mit dem Namen „AntiSoldiersRush“ in die Regelbasis aufgenommen.
2. In fast allen erfolgreichen Lösungen gegen den „Knight’s Rush“ baut die KI sehr schnell Fortschrittliche Einheiten. Dies führte zu einer neuen Regel, die der KI vorschreibt dann ein neues Gebäude zu errichten, wenn genau dieses Gebäude zu einer Fortschrittlicheren Einheit führt und dann mit dieser anzugreifen.

3. Eine weitere Information die aus den Spielen gegen die „Knight’s Rush“-KI erhalten wurde war, dass das Fördern der Ökonomie durch Expandieren zu neuen Rohstoffquellen wichtig für den Spielerfolg ist. Obwohl die ursprüngliche Regelbasis dieses Verhalten zwar schon enthielt, war es häufig so, dass neu erschlossene Rohstoffvorkommen schnell zerstört wurden. Die Erfolgreichen Individuen in dem EA expandierten nur dann, wenn sie auch die nötige Verteidigung hatten, um ihre neuen Expansionen zu verteidigen. Aus diesem Grund wurde die Regel für die Expansion so geändert, dass die Erschließung einer neuen Rohstoffquelle erst nach Bau einer Verteidigungsarmee erfolgen kann.
4. Die vierte neu hinzugefügte Regel, war eine Gegentaktik gegen den „Knight’s Rush“ gegen den es sonst nur schwer ist zu gewinnen. Sie wurde direkt in ihrer Gesamtheit aus einem Individuum, dass sehr erfolgreich war, übernommen.
5. Die letzte Regeländerung war keine neue Regel sondern eine Änderung der Parameter der bereits bestehenden Regeln. Aufgrund der Experimentdaten konnte erschlossen werden zu welchen Zeiten des Spiels idealerweise mit welchen Einheiten angegriffen werden sollte. Durch die Parameteränderungen wurde besonders auch der Bau von Katapulten, wie zuvor diskutiert, wahrscheinlicher. Die ursprüngliche Regelbasis enthielt kaum Regeln, weder für Angriff noch Verteidigung, die Katapulте nutzten.

Um das Verhältnis der Regeln zueinander nicht zu ändern, wurden neue Regeln nicht zusätzlich aufgenommen, sondern ersetzten alte Regeln. Die ursprünglichen Regeln für Luftkampfeinheiten wurden ersetzt, da sie in kaum einer erfolgreichen Strategie zum Einsatz kamen. Andere weniger genutzte Regeln erhielten zudem ein Gewicht, welches die Wahrscheinlichkeit der Ausführung verringert. Weitere mögliche Verbesserungen finden sich auch in [2]. Den auf diese Weise verbesserten dynamischen Spieler lies Ponsen wiederum gegen die zwei spezialisierten und zwei weitere ausgewogene Taktiken antreten, gegen die er auch die ursprüngliche KI spielen lies. Gegen ausgewogene Taktiken gewann die KI in „66.4%“ der Fälle auf großen und „72.5%“ der Fälle auf kleineren Karten. Gegen die spezialisierte Taktik „Soldier’s Rush“ („27.5%“) und „Knight’s Rush“ („10.1%“) hatte die verbesserte KI zwar bessere Chancen verlor aber dennoch die meisten Spiele. Mit seinem Experiment hat Ponsen nicht nur gezeigt, dass sich dynamische Skripte erfolgreich in Echtzeitstrategiespielen einsetzen lassen, sondern auch eine Verbesserung der Regelbasis mit Hilfe von evolutionären Algorithmen eine große Steigerung der Leistung erbringen kann. Die Tatsache, dass die KI dennoch gegen so stark spezialisierte Taktiken verliert begründet Ponsen damit, dass bereits von Beginn des Spiels direkt eine Gegentaktik ausgeführt werden muß, um das Spiel noch gewinnen zu können. Als Lösung schlägt er vor, bereits beim Entwurf des Spiels solche Taktiken zu verhindern, die das Spiel aus der Balance werfen. Eine andere Möglichkeit die von Ponsen nicht in Betracht gezogen wird, in heutigen Echtzeitstrategiespielen aber üblich ist, ist das frühe Auskundschaften des Gegenspielers, um direkt mit Gegenstrategien kontern zu können, anstatt nur festgefahren eine spezielle Taktik auszuführen.

3.3 Zusammenfassung

In dieser Arbeit wurde nach einer kurzen Motivation eine Einführung in die grundlegenden Konzepte Evolutionärer Algorithmen gegeben. Nach einem Beispiel der so genannten Evolutionsstrategien und der Vertiefung in das Gebiet, wurde als Anwendungsmöglichkeit die Verbesserung von Computergegnern bei Echtzeitstrategiespielen erläutert. Der Leser hat einen Überblick über die Unterschiede von Online und Offline-Lernverfahren erhalten sowie einen groben Einblick in die Vor- und Nachteile der Verfahrens bekommen. Hauptschwerpunkt war die Möglichkeit der Verbesserung von dynamisch geskripteten Computergegnern mit Hilfe evolutionärer Algorithmen. Nach der Vorstellung einer Kodierung der einzelnen Strategien wurden die Operatoren erläutert und eine Fitnessfunktion sowie eine Abbruchbedingung formuliert. Das positive Ergebnis war eine deutliche Verbesserung der Performanz mit nur wenigen Durchläufen einer kleinen Anzahl von Generationen, obwohl das Verfahren in seiner Handlungsfreiheit sehr eingeschränkt ist und keine Steuerung einzelner Einheiten ermöglicht. Änderungen die aufgrund der Erkenntnisse in den EA-Durchläufen an der ursprünglichen dynamische geskripteten KI vorgenommen wurden verbesserten ebenfalls das Spielverhalten - nicht nur in Hinblick auf die Spezialtaktiken sondern auch gegen allgemeine gehaltene Strategien. Alles in allem sind evolutionäre Algorithmen ein sehr viel versprechendes Werkzeug für die Weiterentwicklung und Verbesserung bereits vorhandener Spiele-KIs.

Literatur

1. David M. Bourg, Glenn Seemann: AI for Game Developers, 2004
2. Marc Ponsen, Pieter Spronck. Artificial Intelligence, Design and Education (eds. Quasim Mehdi, Norman Gough, Stéphane Natkin and David Al-Dabass), pp. 389-396. University of Wolverhampton, Presented at the CGAIDE, 2004
3. Marc Ponsen: Improving Adaptive Game AI with Evolutionary Learning, 2004
4. Holland, J.-H.: Outline for a Logical Theory of Adaptive Systems. In: JACM 9, 1962
5. Fogel, L.-J.: Autonomous Automata. In: Industrial Research 4, 1962
6. Rechenberg, I. Cybernetic Solution Path of an Experimental Problem. 1965
7. Schwefel, H.-P.: Kybernetische Evolution als Strategie der Experimentellen Forschung in der Strömungstechnik, Technische Universität Berlin, Hermann Föttinger-Institut für Strömungstechnik, Diplomarbeit, März 1965
8. Beyer, H.-G. ; Schwefel, H.-P: Evolution Strategies - a Comprehensive Introduction. Kluwer academic publishers, 2002

Seminar

Fuzzy-Logik in

Computerspielen

Computational Intelligence bei Computerspielen

Ausgearbeitet von Sebastian Schnelker

1 Abstrakt

Die Theorie der Fuzzy-Logik hat in vielen industriellen Anwendungen Fuß gefasst. Insbesondere in der Regelungstechnik wurde der große Nutzen erkannt, wie mit der Fuzzy-Logik einfach und umgangssprachlich Expertenwissen in Regelsystemen abgebildet werden kann. Diese Vorteile lassen sich auch auf die Anwendung in Computerspielen übertragen. Mit Fuzzy-Regelsystemen können schnell Module für die künstliche Intelligenz erzeugt werden, die zur Steuerung von Einheiten, Entscheidungsfindung oder Klassifikation genutzt werden können. Insbesondere bietet die Fuzzy-Logik den Nebeneffekt, dass die Ergebnisse – wie beispielsweise zur Steuerung von Einheiten – viel natürlicher wirken, als ein strikt programmiertes System. Im ersten Teil dieses Seminarbeitrags wird kurz auf die Theorie der Fuzzy-Logik eingangen und im zweiten Teil wird mit der Waffenauswahl eine konkrete Problemstellung aus der Spielewelt besprochen und im Detail vorgeführt. Diese Problemstellung fällt im Bereich der Entscheidungsfindung. Abgerundet wird der Seminarbeitrag mit einer Betrachtung der Problematik der kombinatorischen Explosion von Regelwerken. Als eine Lösungsmöglichkeit wird die Combs-Methode vorgeschlagen und am Beispiel der Waffenauswahl angewendet.

2 Fuzzy-Logik

Im Jahre 1965 hat Lotfi Zadeh, Professor an der Universität von California in Berkeley, ein Paper präsentiert, in dem er die Fuzzymengen-Theorie einführt [Zad65]. Ein Zitat von Zadeh beschreibt die Fuzzy Logik wie folgt: „Fuzzy logic is a means of presenting problems to computers in a way akin to the way humans solve them“. Menschen lösen und analysieren Probleme häufig auf eine ungenaue Art und Weise. Wir Menschen beschreiben Tatsachen häufig mit nicht genau festgelegten Begrifflichkeiten. Nehmen wir als einfaches Beispiel die Größe einer Person. Eine Person ist nicht bei einer genauen Längenangabe von beispielsweise 190 cm groß, sondern wir würden eher einen Menschen in der Größenordnung von 180 cm bis 190 cm als groß bezeichnen. Ist demnach aber eine Person, die 179.5 cm groß ist, dann kein großer Mensch? Wir sehen, dass die Einteilung keine

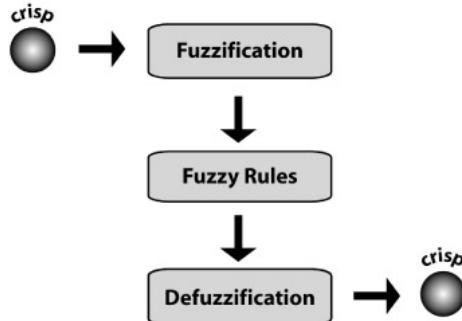


Abbildung 1. Vorgehensweise bei der Fuzzy-Inferenz [Buc05]

scharfen Grenzen kennt, sondern ein fließender Übergang vorherrscht. Personen, die 175 cm lang sind, würden wir eher als groß bezeichnen als Personen, die nur 170 cm lang sind.

Computer, die bisher nur mit scharfen Aussagen gearbeitet haben, sollen durch die Fuzzy-Logik so erweitert werden, dass ein einfacher Transfer von menschlichen Expertenwissen in die digitale Welt durchgeführt werden kann. Die Theorie der Fuzzy Logik versucht, die unscharfe Beschreibung von sogenannten linguistischen Termen – wie z.B. *groß*, *heiss* oder *weit entfernt* – auf Computer zu übertragen. Dabei soll eine fließende Formulierung verwendet werden, die nicht nur die scharfen Werte 0 oder 1 beinhaltet. Eine bestimmte Temperatur kann beispielsweise eine sogenannte Zugehörigkeit zum Term *heiss* von 0.7 besitzen. Je höher der Zugehörigkeitswert bzgl. eines Terms ist, desto eher kann der Werte diesem Term zugeordnet und als dieser interpretiert werden.

Der Seminarbeitrag beschäftigt sich im Weiteren hauptsächlich mit der Inferenz von Fuzzy-Regelwerken. Der Vorgang der Fuzzy-Inferenz umfasst typischerweise drei Schritte. Für scharfe (engl. crisp) Eingabedaten wird die Zugehörigkeit zu linguistischen Termen ermittelt. Dieser Vorgang wird Fuzzyfizierung genannt. Beispielsweise könnte bei gegebener Temperatur ermittel werden, ob oder wie stark die Temperatur als heiß aufgefasst wird. Anschließend werden diese ermittelten Zugehörigkeitswerte während der Inferenz der Regeln verwendet. Zum Schluss wird bei der Defuzzyfizierung aus den ermittelten unscharfen Werten ein scharfer Ausgabewert ermittelt, siehe Abbildung 1.

2.1 Kurze Einführung in die Fuzzy-Theorie

Fuzzy-Logik wird auch als eine unscharfe Logik bezeichnet. Der Begriff der Unschärfe entstand daraus, dass die Fuzzy-Theorie das Konzept der klassischen Logik erweitert, so dass nicht nur Werte $\in \{0, 1\}$ möglich sind, sondern auch Werte aus dem Intervall $[0, 1]$ verarbeitet werden können. Herkömmliche scharfe Mengen können über die Indikatorfunktion charakterisiert werden [Rud07]:

$$A(x) := 1_{[x \in A]} := 1_A(x) := \begin{cases} 1 & x \in A \\ 0 & x \notin A \end{cases}$$

Diese scharfe Definition lässt nur die beiden Fälle zu, in denen entweder ein Element innerhalb der Menge oder nicht in der Menge enthalten ist. Diese Einschränkung wird durch die Einführung einer unscharfen Mengendefinition erweitert.

Definition 1: Eine Abbildung $F : A \rightarrow [0, 1] \subset \mathbb{R}$, die jedem Element $x \in A$ seinen Zugehörigkeitswert $F(x)$ zu F zuordnet, wird Fuzzy-Menge oder unscharfe Menge genannt.

Üblicherweise können Fuzzy-Mengen graphisch sehr anschaulich dargestellt werden. Dabei werden Zugehörigkeitswerte gegen scharfe Werte aufgetragen, siehe Abbildung 2.

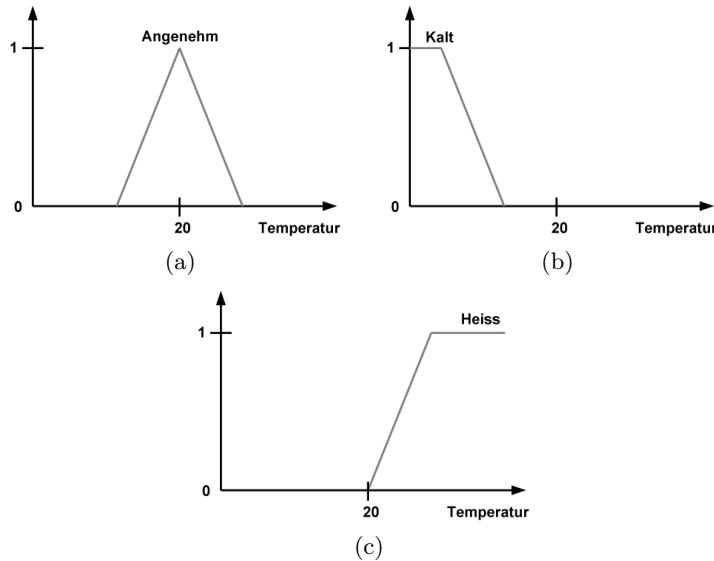


Abbildung 2. Verschiedene Fuzzy-Mengen: (a) Dreiecksfunktion (b) Rechtsseitig abfallende Funktion (c) Linksseitig abfallende Funktion

Die Theorie der Fuzzy-Mengen beinhaltet Interpretationen der Mengenoperatoren Vereinigung, Durchschnitt, Komplement sowie Gleichheit und Teilmengenbeziehungen [Rud07].

Definition 2: Gegeben sind zwei unscharfe Mengen A und B über einem Universum X und entsprechende Zugehörigkeitsfunktionen $A(\cdot)$ und $B(\cdot)$.

- Die **Vereinigung** C von A und B wird wie folgt definiert:
 $C := A \cup B$ mit $C(x) := \max\{A(x), B(x)\}$ für alle $x \in X$
- Der **Durchschnitt** C von A und B wird wie folgt definiert:
 $C := A \cap B$ mit $C(x) := \min\{A(x), B(x)\}$ für alle $x \in X$
- Das **Komplement** C von A wird wie folgt definiert:
 $C := A^c$ mit $C(x) := 1 - A(x)$ für alle $x \in X$
- **Gleichheit:** $A = B$ falls $\forall x \in X : A(x) = B(x)$
- **Teilmenge:** $A \subseteq B$ falls $\forall x \in X : A(x) \leq B(x)$
- **echte Teilmenge:** $A \subset B$ falls $A \subseteq B$ und $\exists x \in X : A(x) < B(x)$

Fuzzy-Mengen erfüllen mit den in Definition 2 beschriebenen Operatoren zahlreiche Gesetzmäßigkeiten, so dass Fuzzy-Mengen mit den Operationen \cup und \cap einen distributiven Verband mit neutralem Element und Einselement bilden. Darüber hinaus bildet die Theorie der Fuzzy-Mengen allerdings keine Boolesche Algebra. Aus dieser Tatsache lässt sich schließen, dass das Prinzip vom ageschlossenen Dritten (tertium non datur) nicht gültig ist. Diese Tatsache ist aber gerade erwünscht, weil die Fuzzytheorie das Prinzip der Unschärfe in den Vordergrund stellt. Es lässt sich aber festhalten, dass viel Struktur durch den distributiven Verband gewonnen wird [Rud07].

Im Rahmen der Fuzzy-Logik können die Standard-Operatoren durch neue Operatoren ausgetauscht werden. Trivialer Weise können neue Operatoren nicht willkürlich gewählt werden, sondern sie müssen bestimmte Charakteristiken aufweisen. Die Charakterisierung findet über vorgegebene Axiome statt. Jeder Operator für eine bestimmte Operation muss einige wenige wichtige Eigenschaften aufweisen, um sinnvoll im Kontext der Fuzzymengen eingesetzt zu werden. Im Folgenden werden für das Komplement, den Durchschnitt und die Vereinigung Axiome vorgestellt.

Fuzzy-Komplement Axiome: Sei $c : [0, 1] \rightarrow [0, 1]$.

- $c(0) = 1$ und $c(1) = 0$
- $\forall a, b \in [0, 1] : a \leq b \Rightarrow c(a) \geq b$ (monoton)
- optional: $c(\cdot)$ ist stetig
- optional: $\forall a \in [0, 1] : c(c(a)) = a$ (involutiv)

Die Verallgemeinerung der Schnittmengenoperation wird t-Norm genannt.

Fuzzy-Durchschnitt Axiome: Sei $i : [0, 1] \times [0, 1] \rightarrow [0, 1]$ und $a, b, d \in [0, 1]$

- $i(a, 1) = a$
- $b \leq d \Rightarrow i(a, b) \leq i(a, d)$ (monoton)
- $i(a, b) = i(b, a)$ (kommutativ)
- $i(a, i(b, d)) = i(i(a, b), d)$ (assoziativ)

Die Verallgemeinerung der Vereinigung wird s-Norm bezeichnet und entsprechend über Axiome definiert. Es gelten wie bei der t-Norm die Monotonie, die Kommutativität und Assoziativität. Allerdings ist das neutrale Element nicht die 1 sondern die 0, so dass sich das erste Axiom zu $i(a, 0) = a$ ergibt.

In der klassischen Mengenlehre ist die Vereinigung und der Schnitt dual bzgl. des Komplements, was gleichbedeutend mit dem Erfüllen der DEMORGANSche Gesetze ist. In der Fuzzy-Mengenlehre ist eine t-Norm und s-Norm dual bzgl. dem Fuzzy-Komplement, wenn

$$\begin{aligned} c(t(a, b)) &= s(c(a), c(b)) \\ c(s(a, b)) &= t(c(a), c(b)) \end{aligned}$$

gilt. Ein Tripel (t, s, c) einer t-Norm, s-Norm und einem Komplement c heisst duales Tripel, wenn t und s dual zu c sind. In der nachfolgenden Tabelle sind übliche Beispiele für duale Tripel aufgeführt.

t-Norm	s-Norm	Komplement
$\min\{a, b\}$	$\max\{a, b\}$	$1 - a$
ab	$a + b - ab$	$1 - a$
$\max\{0, a + b - 1\}$	$\min\{1, a + b\}$	$1 - a$

2.2 Fuzzy-Inferenz

In dem vorangegangenen Abschnitt wurde eine kurze Einführung in die Theorie gegeben und Beispiele für einfache Fuzzy-Mengen aufgezeigt. Beispielsweise wurde eine Fuzzy-Menge graphisch dargestellt, die eine angenehme Temperatur beschreiben soll. Diese wörtliche Beschreibung umfasst zwei Begrifflichkeiten. Zum einen die Kenngröße *Temperatur* und zum anderen einen Wert der Kenngröße *angenehm*. In der Terminologie der Fuzzy-Logik wird die *Temperatur* als linguistische Variable bezeichnet und der Wert *angenehm* als linguistischer Term. Die einzelnen Werte eines linguistischen Terms werden durch Fuzzy-Mengen ausgedrückt. Mit diesen Hilfsmitteln können nun unscharfe Aussagen p formuliert werden:

$$p : \text{Temperatur ist angenehm}$$

Im Weiteren ist der Wahrheitsgrad der unscharfen Aussage von Bedeutung. Dieser kann für eine scharfe Eingabe genau ermittelt werden, indem der Zugehörigkeitswert zu der Fuzzy-Menge errechnet wird.

Interessant – gerade im Hinblick auf den Nutzen in Computerspielen – wird das Einführen von Fuzzy-Regeln. Das Konzept der klassischen logischen IF-THEN-Regeln kann einfach übertragen werden. Allerdings werden Regeln nicht scharf formuliert und besitzen nicht nur Wahrheitswerte $\in \{0, 1\}$, sondern können beliebige Werte aus dem Intervall $[0, 1]$ annehmen. Eine einfache Regel könnte folgendermaßen formuliert werden:

$$\mathbf{IF} \text{ Heizung ist heiß } \mathbf{THEN} \text{ Energieverbrauch ist hoch}$$

Die linguistischen Variablen sind in diesem Beispiel *Heizung* und *Energieverbrauch*, wobei hingegen *heiß* und *hoch* linguistische Terme darstellen. Es sei nochmal darauf hingewiesen, dass die Terme durch Fuzzy-Mengen charakterisiert werden und nicht die Variablen. Allgemein betrachtet, wird mit der Regel eine Relation zwischen der Temperatur der Heizung und der Höhe des Energieverbrauchs aufgestellt. Diese Relation kann als Implikation über die Domain Heizung H und Energieverbrauch E aufgefasst werden:

$$\forall(x, y) \in H \times E : R(x, y) = \text{Impl}(H(x), E(y))$$

Eine geeignete Implikation muss nun gewählt werden, die auch der logischen Aussage gerecht wird. Als Ausgangspunkt der Überlegungen dient die scharfe Implikation $a \Rightarrow b$, welche äquivalent zu $\bar{a} \vee b$ ist. Aus der Sicht der Fuzzy-Theorie könnte nun eine sogenannte S-Implikation abgeleitet werden, die sich zu $\text{Imp}(a, b) = s(c(a), b)$ ergibt. Es gibt weitere zahlreiche Ansätze für geeignete Implikationen. In der Regelungstechnik und in der Spieleindustrie werden die Implikationen relativ einfach als Minimum aufgelöst. Diese Methode hat ihren Ursprung in der Regelungstechnik und wird dort als Mamdani-Regler bezeichnet [MB75].

Der Wahrheitswert einer Regel entspricht genau dem Wert der Implikation, welcher für scharfe Werte einfach ausgerechnet werden kann. Für den Einsatz in Computerspielen ist es jedoch erforderlich, nicht nur eine Regel zu evaluieren, sondern komplexe Regelsysteme auszuwerten, die z.B. die Bewegung eines Spielers steuern. Sei ein Regelsystem mit n Regeln gegeben.

$$\begin{aligned} \text{IF } X \text{ ist } A_1 \text{ THEN } Y \text{ ist } B_1 &\longrightarrow R_1(x, y) = \text{Impl}_1 = (A_1(x), B_1(y)) \\ \text{IF } X \text{ ist } A_2 \text{ THEN } Y \text{ ist } B_2 &\longrightarrow R_2(x, y) = \text{Impl}_2 = (A_2(x), B_2(y)) \\ \text{IF } X \text{ ist } A_3 \text{ THEN } Y \text{ ist } B_3 &\longrightarrow R_3(x, y) = \text{Impl}_3 = (A_3(x), B_3(y)) \\ &\vdots \\ \text{IF } X \text{ ist } A_n \text{ THEN } Y \text{ ist } B_n &\longrightarrow R_n(x, y) = \text{Impl}_n = (A_n(x), B_n(y)) \end{aligned}$$

Ist als Prämisse eine scharfe Eingabe x_0 gegeben, so kann die Implikation einer Regel als eine Ausgabe-Fuzzymenge über y betrachtet werden. Wir erhalten die folgenden resultierenden Fuzzy-Mengen B'_i :

$$\begin{aligned} B'_1(y) &= \text{Impl}_1 = (A_1(x_0), B_1(y)) \\ &\vdots \\ B'_n(y) &= \text{Impl}_n = (A_n(x_0), B_n(y)) \end{aligned}$$

Die lokalen Inferenzen der Teilregeln wurden nun damit ausgeführt. Um die Teilergebnisse der einzelnen Regeln miteinander in Beziehung zu setzen, muss eine Aggregation der lokalen Inferenzen durchgeführt werden. Aus der Aggregation entsteht eine Lösungs-Fuzzymenge B' des Regelwerks.

$$\text{Aggregation: } B'(y) = \text{aggr}\{B'_1(y), \dots, B'_n(y)\}, \text{ wobei } \text{aggr} = \begin{cases} \min \\ \max \end{cases}$$

Diese Vorgehensweise, in der erst eine Inferenz jeder Regel stattfindet und danach die Aggregation, wird in der Literatur als FITA-Prinzip (First Inference, Then Aggregate) bezeichnet. Alternativ kann erst eine Superrelation aus den Relationen jeder Regel durch eine Aggregierungsfunktion gebildet werden. Anschließend wird die Inferenz auf diese Superrelation durchgeführt, weshalb diese Vorgehensweise auch als FATI-Prinzip (First aggregate, then inference) bezeichnet wird. Es kann gezeigt werden, dass bei einer scharfen Eingabe für die Prämisse, die beiden Vorgehensweisen äquivalent sind [Rud07].

Regeln besitzen üblicherweise nicht nur eine einzelne Bedingung in der Prämisse. Fuzzy-Regeln stellen in dieser Hinsicht auch keine Beschränkung dar. Bedingungen können beliebig mit dem AND-Operator oder OR-Operator verknüpft werden. Die UND-Verknüpfung wird allgemein mit einer t-Norm umgesetzt und entsprechend wird die OR-Verknüpfung mit einer s-Norm umgesetzt.

UND-Verknüpfung:

IF X_1 ist A_{11} AND X_2 ist A_{12} AND … AND X_m ist A_{1m} THEN Y ist B_1

⋮

IF X_1 ist A_{n1} AND X_2 ist A_{n2} AND … AND X_m ist A_{nm} THEN Y ist B_n

Zusammenfassen der Prämisse von Regel k : $A_k(x_1, \dots, x_m) = t(A_{k1}(x_1), \dots, A_{km}(x_m))$

ODER-Verknüpfung:

IF X_1 ist A_{11} OR X_2 ist A_{12} OR … OR X_m ist A_{1m} THEN Y ist B_1

⋮

IF X_1 ist A_{n1} OR X_2 ist A_{n2} OR … OR X_m ist A_{nm} THEN Y ist B_n

Zusammenfassen der Prämisse von Regel k : $A_k(x_1, \dots, x_m) = s(A_{k1}(x_1), \dots, A_{km}(x_m))$

Analog wird für die Verneinung in Regeln das Fuzzy-Komplement verwendet.

2.3 Defuzzifizierung

Nachdem die Fuzzifizierung und die Inferenz durchgeführt worden sind, steht als Ergebnis nur eine Fuzzy-Menge zur Verfügung und noch kein scharfer Wert, der verwendet werden kann. Es existieren viele Verfahren, wie aus der resultierenden Fuzzy-Menge ein scharfer Wert gewonnen werden kann [Rud07]. Im Folgenden werden einige Ansätze vorgestellt. Ausgangspunkt stellt immer die aus der Inferenz resultierende Fuzzy-Menge $B'(y)$ dar.

- **Maximum-Methode:** Nur die Regel mit höchstem Erfüllungsgrad wird berücksichtigt

$$\hat{y} = \operatorname{argmax} B'(y)$$

- **Maximummittelwert-Methode:** alle Regeln mit höchstem Erfüllungsgrad werden berücksichtigt

$$\hat{y} = \frac{1}{|Y^*|} \sum_{y \in Y^*} B(y) \text{ mit } Y^* = \{y \in Y : B'(y) = hgt(B')\}$$

- **Schwerpunkt-Methode:** alle Regeln werden berücksichtigt, dadurch sehr rechenintensiv

$$\hat{y} = \frac{\int y \cdot B'(y) dy}{\int B'(y) dy}$$

- **Flächen-Methode:** gedacht als Approximation von der Schwerpunktmethode

$$\hat{y} = \frac{\sum_k \hat{y}_k \cdot B'_k(y_k)}{\sum_k B'_k(y_k)} \text{ mit } \hat{y}_k \text{ Schwerpunkt von } B'_k(y)$$

Die Schwerpunktmethode ist nach ihrer Berechnungsvorschrift die genaueste Methode, jedoch auch die ineffizienteste. Dieser Tradeoff zwischen Genauigkeit und Effizienz rückt die Bedeutung anderer Methoden in den Vordergrund. In realen Anwendungen werden häufig schnelle Ergebnisse des Regelwerkes verlangt, so dass die Schwerpunktmethode praktisch nicht einsetzbar wird. Alternativ werden häufig Approximationen verwendet, wie die Flächenmethode, deren Berechnung einen praktischen Vorteil bietet. Eine weitere Diskussion wird in Kapitel 4.4 fortgesetzt.

3 Fuzzy-Logic in Games

Fuzzy-Logik kann in vielerlei Hinsicht in Computerspielen angewendet werden. Die Problemstellungen aus den industriellen Anwendungen finden sich in Computerspielen oft sehr genau wieder. Im Bereich der Regelung oder Steuerung von Prozessen, wie zum Beispiel steuern von Zügen, Klimaanlagen, Robotern oder ähnlichen Anwendungen, findet die Fuzzy-Logik Anwendung. Ähnliche Aufgaben gilt es auch in Computerspielen zu bewältigen. Die Steuerung von Non-Player-Character Einheiten oder Fahrzeugen wäre ein Beispiel. Im Vordergrund steht dabei immer eine Steuerung, die natürlich reagiert und keine abrupten Entscheidungen trifft. Das Verfolgen von sich bewegenden Zielen ist eine schwer zu programmierende Aufgabe, wenn es darum geht, dass die Einheiten eine natürlich wirkende Bewegung vollziehen. Gerade bei Richtungswechseln des Ziels, werden bei einer einfachen Implementierung abrupte unnatürliche Bewegungen schnell durchgeführt. Mit einer Modellierung durch Fuzzy-Regeln basierend auf Zielangaben wie *weit_rechts*, *rechts* oder *geradeaus* könnte das Problem elegant umgangen werden.

Des Weiteren besteht eine typische Anwendung im Bereich der Entscheidungsfindung. Das Abschätzen von Gefahren wäre eine Problemstellung, die in dem

Bereich der Entscheidungsfindung fällt. Ein Computergegner kann die Gefahr eines Gegners ungefähr einschätzen, in dem zum Beispiel auf der Basis von Termen zur Entfernungangabe und Truppenstärke eine Entscheidung getroffen wird.

Fuzzy-Logik kann auch zur Klassifikation von Daten herangezogen werden. In einem Spiel könnten Spieler bewertet werden, um daraus weitere Entscheidungen zu treffen. Eine Bewertung könnte auf Truppenstärke, Effektivität und weiteren beliebigen Faktoren basieren und den Spieler in Klassen wie zum Beispiel *einfach*, *moderat* oder *spielstark* einteilen.

Ein Vorteil ist aus den kurz angerissenen Beispielen direkt ersichtlich: Lösungsansätze können sehr schnell und ohne viel Implementierungsaufwand generiert werden. Vorhandenes Expertenwissen ist schnell in Regelwerken integriert und kann gut getestet werden. Zusätzlich ist das Anpassen des Verhaltens sehr einfach. Fuzzy-Mengen können beliebig geändert und Regeln verfeinert werden. Ist das erwartete Verhalten nicht ausreichend, kann die Komplexität des Regelwerkes durch Hinzufügen von weiteren Variablen und/oder Termen gesteigert werden.

Im Folgenden wird anhand des Beispiels der Waffenauswahl in Computerspielen der gesamte Entwicklungsprozess im Detail besprochen. Dies umfasst die Festlegung von Variablen, die Modellierung von Fuzzy-Mengen, das Aufstellen des Regelwerkes sowie die Ergebnisfindung durch eine Inferenz.

4 Aufbau eines Moduls zur natürlichen Waffenauswahl mittels eines Fuzzy-Regelsystems [Buc05]

Menschliche Entscheidungsprozesse können sehr gut durch Fuzzy-Regelsysteme modelliert werden. Der Mensch bewertet seine Umwelt nicht durch scharfe Entscheidungsvariablen, sondern durch unscharfe verbale Beschreibungen. Diese verbalen Beschreibungen können durch linguistische Terme in der Fuzzy-Logik passend beschrieben werden.

Die Waffenauswahl eines Spielers in einer bestimmten Spielsituation ist ein verbal gut charakterisierbarer Prozess. Ein Spieler wählt – so die Annahme in dem Beispiel – auf Basis der Distanz zum Gegner und zur vorhandenen Anzahl an Munition eine Waffe aus. Intuitiv sollte eine Nahkampfwaffe nicht für eine große Distanz eingesetzt werden und keine Munition sinnlos verschwendet werden. Die Charakteristika werden durch die linguistischen Variablen *Distance to Target*, *Ammo Status* und *Desirability* beschrieben. Darauf basierend werden Regeln aufgestellt, die für ein bestimmtes Spielszenario einem Nonplayer-Character vorgeben sollen, ob die aktuell gewählte Waffe gut geeignet ist.

Für den Modellierungsprozess des Fuzzy-Systems ist es entscheidend, dass zuerst der genaue Zweck des Regelsystems festgelegt wird und die zu nutzenden linguistischen Variablen definiert werden. Im Weiteren werden zu jeder Variablen linguistische Terme definiert. Die Definition erfolgt über die Modellierung der entsprechenden Fuzzy-Mengen. Da die Fuzzy-Mengen eine wichtige Rolle im Regelsystem spielen und das Ergebniss entscheidend prägen, sollten diese

sorgfältig ausgewählt werden. Es existieren zwei grundsätzliche Vorsätze, die es hierbei zu beachten gilt, siehe Abbildung 3:

1. Wenn die Fuzzy-Mengen der linguistischen Terme einer linguistischen Variable graphisch übereinandergelegt werden, dann sollte die Summe der Zugehörigkeitswerte an einem Punkt nicht den Wert 1 überschreiten. Dies ermöglicht weichere Übergänge in den Entscheidungen.
2. Wenn die Fuzzy-Mengen der linguistischen Terme einer linguistischen Variable übereinandergelegt werden, dann sollte jede einzelne Zugehörigkeitsfunktion andere Zugehörigkeitsfunktionen maximal an 2 oder wenigen Stellen schneiden.

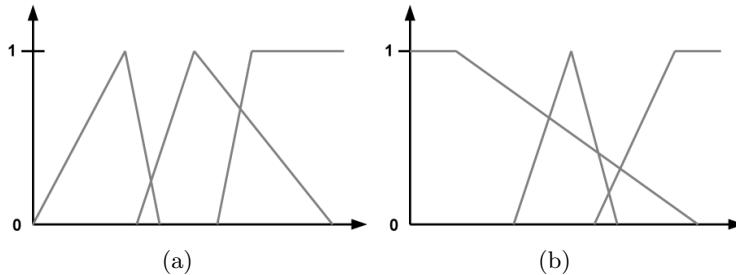


Abbildung 3. Zwei Beispiele, die die Vorsätze bei der Modellierung von Fuzzy-Mengen missachten

4.1 Modellierung der Fuzzy-Mengen

Das Waffenauswahl-Beispiel nutzt die drei linguistischen Variablen *Distance to Target*, *Ammo Status* und *Desirability*. Zu jeder Variablen werden im Folgenden Fuzzy-Mengen graphisch modelliert.

Die linguistische Variable *Desirability* soll einen Bereich von 0 bis 100 abdecken. Deswegen ist es erforderlich, dass dieser Bereich auch von den zugehörigen Termen abgedeckt wird. In dem Beispiel werden drei Terme benutzt: *Undesirable*, *Desirable* und *VeryDesirable*. In der Abbildung 4 ist eine Möglichkeit, die die genannten Vorsätze beachtet, dargestellt.

Die nächste linguistische Variable *Distance to Target* wird ähnlich modelliert. Es werden wieder drei linguistische Terme *Target_Close*, *Target_Medium* und *Target_Far* verwendet, die gut genug geeignet sind, um eine Spielsituation hinsichtlich der Entfernung eines Gegners zu beschreiben. Ein nahes Ziel sollte auch nur als nah gelten, wenn es in der unmittelbaren Umgebung erscheint. Mit diesen Erwartungen wird die Fuzzy-Menge *Target_Close* so modelliert, in dem nur ein kleiner Bereich Zugehörigkeitswerte von 1 aufweist und anschließend stark abfällt. Etwas breiter gestreut wird der Term *Target_Medium* gewählt, was sich

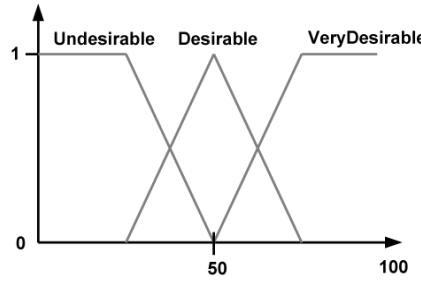


Abbildung 4. Graphische Darstellung der Fuzzy-Mengen zu den Termen *Undesirable*, *Desirable* und *VeryDesirable*

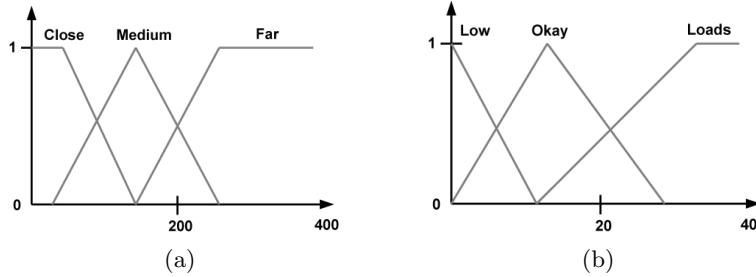


Abbildung 5. (a) Beschreibung der Fuzzy-Mengen zu *Distance to Target* (b) Beschreibung der Fuzzy-Mengen zu *Ammo Status*.

sehr gut mit einer Dreiecksfunktion beschreiben lässt. Ab einer Entfernung von 300 Einheiten werden Ziele als weit entfernt deklariert, siehe Abbildung 5(a). Als letzte linguistische Variable wird *Ammo Status* mit drei Fuzzy-Mengen *Ammo_Low*, *Ammo_Okay*, *Ammo_Loads* modelliert, siehe Abbildung 5(b). Im Gegensatz zu den anderen Variablen unterscheidet sich *Ammo Status* darin, dass die Fuzzy-Mengen der Terme für jede Waffe unterschiedlich beschrieben werden müssen.

Das Beschreiben von Variablen bzw. Termen ist ein intuitiver Prozess, der einfach durchgeführt werden kann. Fuzzy-Mengen werden nach Ermessen des Entwicklers modelliert und können schnell angepasst werden, wenn beim Testen sich ein Verhalten einstellt, dass nicht gefordert ist. An dieser Stelle zahlt sich die einfache verbale Beschreibung von Expertenwissen aus, um trotzdem komplexe Regelwerke zu erzeugen.

4.2 Aufstellen des Regelwerkes

Im ersten Schritt wurden linguistische Terme definiert und geeigneten Fuzzy-Mengen zugeordnet. Nun ist es an der Aufgabe Fuzzy-Regeln aufzustellen. Um alle möglichen Kombinationen zu berücksichtigen, die durch die Terme entstehen, wird für jede Kombination an Termen aus der Prämisse eine Regel erstellt.

Die Prämisse der Regeln wird aus den Variablen *Ammo Status* und *Distance to Target* bestehen. Die Konklusion wird die Variable *Desirability* bilden. Da jede der beiden Variablen aus der Prämisse drei mögliche Terme besitzt, ergeben sich neun zu erzeugende Regeln.

Bei der Regelerzeugung wird wie bei dem Definieren der Fuzzy-Mengen Expertenwissen abverlangt. Die Anzahl der Regeln und deren Prämisse sind durch die Anzahl der Kombinationen fest vorgeben, so dass nur jeder Regel eine passende Konklusion zugewiesen werden muss. Zum Beispiel mag es als sinnvoll erscheinen, dass ein Raketenwerfer eine gute Waffe für die Distanz ist, jedoch auf kurzer Distanz sehr gefährlich für den Spieler selbst sein kann. Weiterhin fliegen die Raketen eines Raketenwerfers nur sehr langsam, so dass auf großer Distanz feindliche Gegenspieler der Rakete einfach ausweichen können. Mit solchen Expertenwissen und Erwartungen können die Regeln einfach aufgestellt werden. Im Folgenden werden neun Regeln präsentiert, die für den Raketenwerfer einen Wert für die *Desirability* der Waffe erzeugen sollen. Natürlich können für andere Waffentypen auf analoge Art und Weise Regelsätze erzeugt werden.

- Regel 1: **IF** *Target_Far AND Ammo_Loads THEN Desirable*
- Regel 2: **IF** *Target_Far AND Ammo_Okay THEN Undesirable*
- Regel 3: **IF** *Target_Far AND Ammo_Low THEN Undesirable*
- Regel 4: **IF** *Target_Medium AND Ammo_Loads THEN VeryDesirable*
- Regel 5: **IF** *Target_Medium AND Ammo_Okay THEN VeryDesirable*
- Regel 6: **IF** *Target_Medium AND Ammo_Low THEN Desirable*
- Regel 7: **IF** *Target_Close AND Ammo_Loads THEN Undesirable*
- Regel 8: **IF** *Target_Close AND Ammo_Okay THEN Undesirable*
- Regel 9: **IF** *Target_Close AND Ammo_Low THEN Undesirable*

4.3 Fuzzy-Inferenz

Mit den vorangegangen Abschnitten sind alle Voraussetzungen gelegt worden, um eine Inferenz auf ein Regelwerk durchzuführen. Theoretisch wurde das Prinzip der Inferenz im Abschnitt 2.2 beschrieben. Das Regelwerk wird bei der Inferenz mit scharfen Eingabewerten gefüttert und ausgewertet, um diejenigen Regeln zu ermitteln, die feuern. Eine Regel feuert, wenn der Wahrheitswert der Prämisse größer als 0 ist. Das Schema einer Inferenz gliedert sich grundlegend beim FITA-Ansatz in folgende drei Punkte:

1. Für jede Regel
 - (a) Errechne die Zugehörigkeitswerte der Terme in der Prämisse
 - (b) Berechne den Wahrheitswert der Prämisse
2. Aggregiere alle errechneten Fuzzy-Mengen
3. Für scharfe Lösungswerte muss eine Defuzzifizierung durchgeführt werden

Wir wollen nun das Beispiel der Waffenauswahl schrittweise durchgehen und eine Lösung evaluieren. Als scharfe Eingabewerte wählen wir eine Distanz zum Ziel von 200 und eine Menge an Munition von 8 Raketen.

Die Regeln werden nun nacheinander betrachtet (siehe FITA-Prinzip 2.2).

- Regel 1: **IF** *Target_Far AND Ammo_Loads THEN Desirable*

Der Zugehörigkeitswert von *Target_Far* beim Wert 200 ist 0.33, siehe Abbildung 6(a). Der Zugehörigkeitswert von *Ammo_Loads* beim Wert 8 ist 0, siehe Abbildung 6(b). Der AND-Operator resultiert in dem Minimum der beiden Werte, so dass der Wahrheitsgehalt der Prämisse für die scharfe Eingabe 0 entspricht. Mit anderen Worten feuert diese Regel nicht. Als Implikation wird die Mamdani-Implikation verwendet, die dem Minimum aus dem Wahrheitsgehalt der Prämisse und dem Zugehörigkeitswert der Konklusion entspricht.

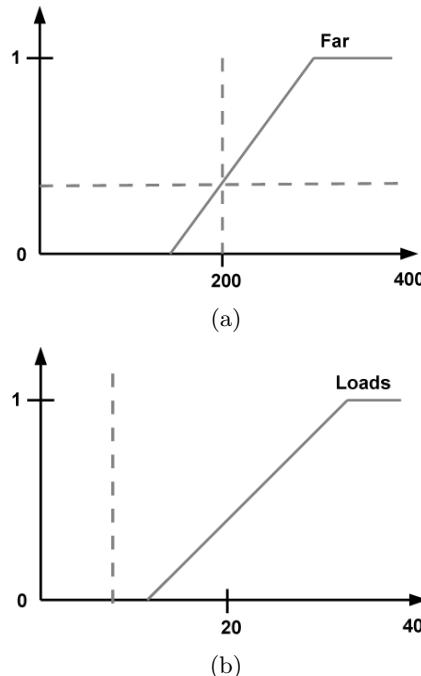


Abbildung 6. Graphische Auswertung des Wahrheitsgehalts der Prämisse von Regel1.

- Regel 2: **IF** *Target_Far AND Ammo_Okay THEN Desirable*

Für die zweite Regel ergibt sich der Zugehörigkeitswert von *Target_Far* beim Wert 200 wie in Regel 1 zu 0.33, siehe Abbildung 4.3. Der Zugehörigkeitswert von *Ammo_Okay* beim Wert 8 ist jedoch 0.78. Das Minimum der beiden Werte ergibt 0.33 und somit ist der Wahrheitsgehalt der Regel 0.33 und der Wert von *Desirable* maximal 0.33. Diese Regel feuert also ein wenig.

Diese einfache Vorgehensweise wird nun weiter für alle übrigen Regeln durchgeführt. Die grafische Veranschaulichung der Vorgehensweise ist eine große Hilfe für das Verständnis des Vorgangs. Die Implementierung eines entsprechenden

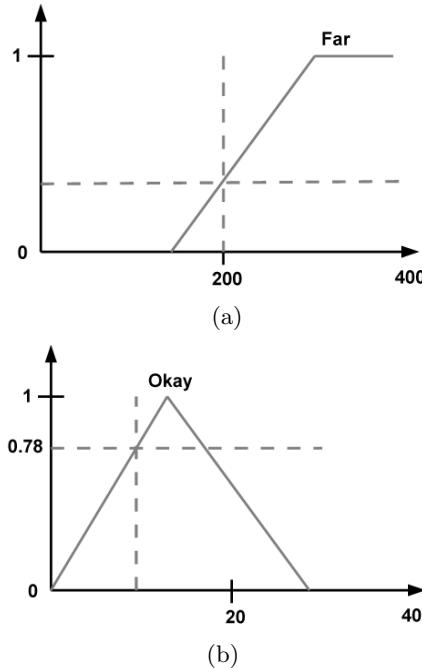


Abbildung 7. Graphische Auswertung des Wahrheitsgehalts der Prämisse von Regel2.

Frameworks zur Fuzzy-Inferenz stellt jedoch auch keine große programmiertechnische Herausforderung dar, siehe dazu [Buc05]. Nachdem die Inferenz für jede Regel durchgeführt worden ist, können die Ergebnisse in einer Matrix aufgezeigt werden. Diese Matrix wird auch *fuzzy associative matrix* (FAM) genannt. In unserem einfachen Beispiel ergibt sich die Matrix aus den Kombinationen der Prämisse zu einer 3×3 -Matrix, siehe Abbildung 8.

In dem betrachteten Beispiel hat jeweils eine Regel für *VeryDesirable* mit dem Wert 0.67 und eine Regel für *Desirable* mit dem Grad 0.2 gefeuert. Häufig besitzen aber verschiedene Regeln dieselbe Konklusion und es kommt nicht selten vor, dass diese Regeln mit unterschiedlichen Werten feuern. Für den Term *Undesirable* zum Beispiel haben zwei Regeln gefeuert, zum einen mit dem Wert 0.2 und zum anderen mit den Wert 0.33. Es gibt mehrere Techniken wie verschiedene Wahrheitsgehalte für gleiche Terme in Konklusionen behandelt werden. Eine Möglichkeit besteht darin die Werte zu addieren, wobei die Summe eine obere Grenze von 1 nicht überschreiten soll. Eine andere Möglichkeit besteht darin, dass eine Veroderung der Werte durchgeführt wird und nur der jeweils größte Wert für die weitere Aggregierung genommen wird. In diesem Beispiel präferieren wir die zweite Variante und nehmen immer den höchsten Wahrheitsgehalt.

Es ergeben sich nun durch die Implikation resultierende Fuzzy-Mengen für die Terme in den Konklusionen. Graphisch betrachtet wird in den Fuzzy-Mengen

		Target_Close	Target_Medium	Target_Far
		Undesirable	Desirable	Undesirable
Ammo_Low	Undesirable	0	0.2	0.2
	Ammo_Okay	0	0.67	0.33
	Ammo_Loads	0	0	0

Abbildung 8. Die *fuzzy associative matrix* (FAM) fasst die Ergebnisse der Inferenz zusammen. Zu den möglichen Kombinationen der Prämisse wird der Wahrheitsgehalt der entsprechenden Regel eingetragen.

ein horizontaler Schnitt an der Stelle an der Y-Achse durchgeführt, die dem Wahrheitsgehalt der Prämisse entspricht, siehe Abbildung 9.

Nach dem FITA-Prinzip wird nun eine Aggregierung der resultierenden Fuzzy-Mengen durchgeführt. Als Aggregierungsfunktion nehmen wir die Maximumfunktion. Graphisch lässt sich dieser Schritt wieder sehr anschaulich darstellen. Die durch die Inferenz erzeugten Fuzzy-Mengen werden einfach übereinandergelegt und zu einer Lösungsfuzzymenge verschmolzen, siehe Abbildung 10. Durch die Maximierung entsteht kein Schnitt der Fuzzy-Mengen, sondern eine Vereinigung. Diese Fuzzy-Menge repräsentiert die Konklusion aller Regeln des Regelwerkes. Ist ein scharfer Lösungswert gefordert – wie in unserem Beispiel – so muss als letzter Schritt eine Defuzzifizierung stattfinden.

4.4 Defuzzifizierung

Einige der üblichen Techniken zur Defuzzifizierung wurden in Abschnitt 2.3 skizziert. Wir wollen an dieser Stelle jedoch etwas andere Methoden benutzen, die sehr schnell zu berechnen sind. Bei den bisher vorgestellten Techniken ist der Rechenaufwand zur Berechnung von Schwerpunkten der Fuzzy-Mengen sehr gross. Eine weitere Methode ist die *Centroid* Methode, welche eine ausgewählte Menge S von Samples aus der Domain der linguistischen Variable aufsummiert und nicht über diese integriert:

$$\hat{y} = \frac{\sum_{s \in S} s \cdot B'(s)}{\sum_{s \in S} B'(s)}$$

Die Anzahl der Samples kann frei ausgewählt werden, so dass die Genauigkeit beliebig eingestellt werden kann, die jedoch in einem Tradeoff mit einer längeren

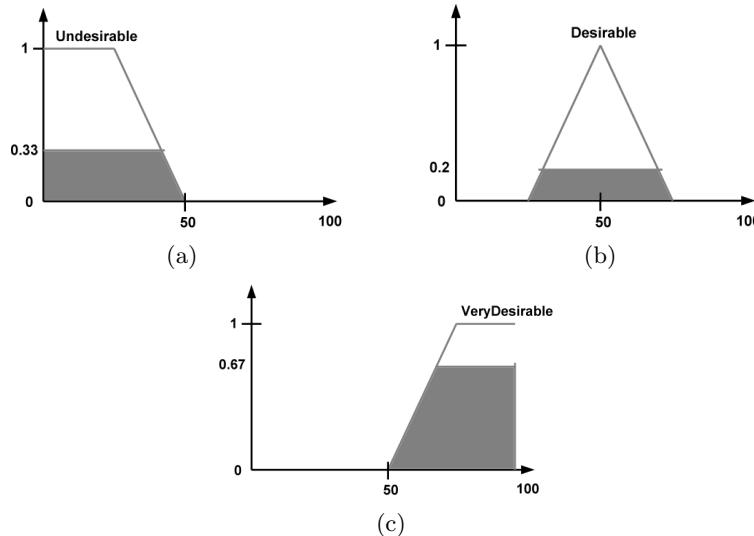


Abbildung 9. Fuzzy-Mengen nach der Inferenz der Regeln. Der ausgegraute resultierende Bereich ergibt sich durch einen horizontalen Schnitt an der Stelle der Y-Achse, die sich durch den Wahrheitsgehalt der am stärksten feuernden Regel ergibt.

Laufzeit steht. Als Daumenregel sollten etwa 10 bis 20 Samples gewählt werden. Wenden wir diese Methode auf unsere Lösungs-Fuzzymenge an, so erhalten wir folgende für die Berechnung notwendigen Werte mit 10 Samples:

s	Undesirable	Desirable	VeryDesirable	Summe
10	0.33	0	0	0.33
20	0.33	0	0	0.33
30	0.33	0.2	0	0.53
40	0.33	0.2	0	0.53
50	0	0.2	0	0.2
60	0	0.2	0.4	0.6
70	0	0.2	0.67	0.87
80	0	0	0.67	0.67
90	0	0	0.67	0.67
100	0	0	0.67	0.67

Werden die Werte nun in die Formel eingesetzt, so erhalten wir einen scharfen Ausgabewert von 62. Zum Vergleich wird die *Average of Maxima* Methode herangezogen. Diese berechnet für alle beteiligten Fuzzy-Mengen einen repräsentativen Wert r . Dieser Wert ist beispielsweise für eine Dreiecksfunktion die Spitze und bei Fuzzy-Mengen mit einem Plateau, ist es der Wert in der Mitte des Plateaus.

$$\hat{y} = \frac{\sum r \cdot B'(r)}{\sum B'(r)}$$

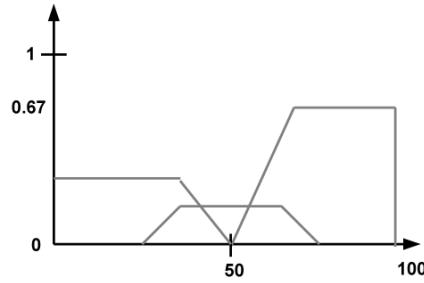


Abbildung 10. Durch die Vereinigung der Fuzzy-Mengen aus Abbildung 9 ergibt sich die Lösungs-Fuzzymenge.

Mit dieser einfachen Methode erhalten wir einen ähnlichen Ausgabewert von 60.625. Die beiden Ergebnisse unterscheiden sich nicht gross, so dass die schnell zu berechende *Average of Maxima* Methode durchaus einzusetzen ist. Trotzdem sei darauf hingewiesen, dass die Ergebnisse sich entsprechend der benutzten Methoden und möglichen Ausgabeszenarien stark unterscheiden können. Das Beispiel sollte aufzeigen, dass es verschiedene Techniken zur Defuzzyfizierung gibt und das für eine konkrete Anwendung abgewägt werden muss, ob eine hohe Genauigkeit gewünscht ist und ob der verursachte Rechenaufwand vertretbar ist. Abschließend ist ein scharfer Wert damit für die *Desirability* eines Raketenwerfers ermittelt worden. Um eine Waffenauswahl durchzuführen, wäre ein solcher Ergebniswert für jede vorhandene Waffe erforderlich. Die Waffe mit dem höchsten Ergebniswert sollte dann im Spiel gewählt werden.

5 Kombinatorische Explosion von Regelwerken

Fuzzy-Regelwerke können schnell aufgestellt werden und bieten durch die umgangssprachliche Beschreibung enorme Vorteile, die bereits aufgezeigt wurden. Allerdings gibt es ein großes Problem, dass bei komplexen Problemstellungen entsteht. Je komplexer ein Problem ist, desto mehr beschreibende Variablen bzw. Fuzzy-Mengen werden benötigt. Dies lässt aber die Anzahl der benötigten Regeln für das Regelwerk explosionsartig ansteigen. Das relativ einfache Beispiel der Waffenselektion benötigt schon neun Regeln, die sich aus der Kombination der möglichen Terme aus der Prämisse ergeben. Wird eine weitere linguistische Variable mit drei Termen hinzugefügt, so wächst das Regelwerk auf 27 Regeln an. Oft wird in bestimmten Problemstellungen ein sehr genauer Detailgrad benötigt und die Variablen erhalten sehr viele linguistische Terme, so dass auch hier die Anzahl der Regeln kombinatorisch in die Höhe steigt. Dieses Problem wird als kombinatorische Explosion bezeichnet. Es sei darauf hingewiesen, dass die Problematik der kombinatorischen Explosion auch bei klassischen scharfen Logiken gegeben ist.

Jede Regel bringt weiteren Rechenaufwand mit sich, so dass ein großes Regelwerk nicht mehr in annehmbarer Zeit gelöst werden kann. Gerade im Kontext

von Computerspielen ist die Performance sehr ausschlaggebend. Die folgende Tabelle zeigt auf, wieviele Regeln mit steigender Anzahl von Variablen für ein vollständiges Regelwerk benötigt werden.

Variablen	Terme pro Variable	Anzahl der Regeln
2	5	$5^2 = 25$
3	5	125
4	5	625
5	5	3125

Um Regelwerke schlank zu halten, werden häufig Regelwerke automatisch datengetrieben erzeugt. Speziell werden Regeln hinsichtlich ihrer Relevanz bewertet und nur bei einem entsprechenden Relevanzwert dem Regelwerk hinzugefügt [Teu95]. Insbesondere werden dabei andere CI-Methoden wie beispielsweise evolutionären Algorithmen eingesetzt, um Regeln zu erzeugen. Ein Individuum repräsentiert eine Regel und wird hinsichtlich der Relevanz bewertet. Der benötigte Rechenaufwand sollte bei diesen Methoden nicht unterschätzt werden und eine Güte kann auch nicht im Vorfeld gewährleistet werden. Einen ganz anderen Ansatz verwendet die Combs-Methode. Sie optimiert keine vorhandenen Regelwerke, sondern stellt eine Methodik vor, wie Regelwerke manuell aufgebaut werden sollen. Interessant hierbei ist, dass ein entsprechendes Regelwerk beim Hinzufügen von Variablen linear anstatt exponentiell wächst.

5.1 Die Combs-Methode

Die nach dem Erfinder William Combs benannte Combs-Methode ist eine Methodik, die es ermöglicht Regelwerke so aufzustellen, dass durch Hinzufügen von Variablen keine kombinatorische Explosion ausgelöst wird, siehe dazu [Com97]. Normalerweise wird für jede Kombination der Eingabeveriablen eine Regel dem Regelwerk hinzugefügt. Williams Combs leitete eine andere Vorgehensweise her, die aus der logischen Umformung von Regeln motiviert ist. Betrachten wir eine Regel aus dem in Kapitel 2 benutzten Regelwerk:

- Regel 1: **IF** *Target_Far* **AND** *Ammo_Loads* **THEN** *Desirable*

Diese Regel ist logisch äquivalent zu der Veroderung der beiden folgenden Regeln:

IF *Target_Far* **THEN** *Desirable*
OR
IF *Ammo_Loads* **THEN** *Desirable*

Der Beweis kann über eine Wahrheitstafel geführt werden [Com97].

Nach der Idee der Combs-Methode soll nur noch für jeden linguistischen Term eine Regel aufgestellt werden. Als Konklusion wird ein Ausgabeterm gewählt, der für die Prämisse am geeignetsten erscheint. In dem Beispiel könnten wir nun sagen, dass bei hoher Entfernung es nicht wünschenswert ist, dass der Raketenwerfer gewählt werden soll und erstellen diese Regel unabhängig von der

Anzahl der verfügbaren Munition. Des Weiteren soll es sehr wünschenswert sein, dass bei viel Munition die Waffe gewählt wird. Auch hierfür wird eine Regel erstellt. Insgesamt ergibt sich nun für jeden Eingabeterm eine Regel, womit ein linearer Zusammenhang zwischen Variablen- bzw. Termanzahl und Regelanzahl hergestellt ist.

Als Voraussetzung wird vorgeschlagen, dass jede Eingangsvariable genauso viele Terme besitzen soll wie die Ausgangsvariable. Diese Voraussetzung ist nicht zwingend notwendig für die Regelgenerierung, ermöglicht es aber, dass jeder Eingangsterm mit genau einem Ausgangsterm gepaart werden kann.

Das Beispiel aus Kapitel 2 kann mit der Combs-Methode mit 6 Regeln aufgestellt werden:

- Regel 1: **IF Target_Far THEN Undesirable**
- Regel 2: **IF Target_Medium THEN Verydesirable**
- Regel 3: **IF Target_Close THEN Undesirable**
- Regel 4: **IF Ammo_Loads THEN VeryDesirable**
- Regel 5: **IF Ammo_Okay THEN Desirable**
- Regel 6: **IF Ammo_Low THEN Undesirable**

Zwar hat sich die Anzahl der Regeln in diesem Beispiel nicht schlagartig reduziert, aber bei vielen Variablen macht sich der Vorteil stark bemerkbar wie die folgende Tabelle zeigt.

Variablen	Terme pro Variable	Anzahl der Regeln	Anzahl der Regeln nach Combs
2	5	$5^2 = 25$	10
3	5	125	15
4	5	625	20
5	5	3125	25

Der große Nachteil der Combs-Methode ist direkt aus diesem Beispiel ersichtlich. Das Regelwerk ist nicht mehr sehr intuitiv und das Problemwissen nicht mehr gut ablesbar. Gerade bei vielen Variablen werden Regelwerke sehr unübersichtlich. Eine grosse Frage steht allerdings im Raum. Verhält sich ein solches Regelwerk genauso wie ein klassisches Regelwerk mit allen Kombinationen? Diese Frage ist nicht explizit zu beantworten, aber es hat sich empirisch gezeigt, dass die Regelwerke sich sehr ähnlich verhalten. Wir wollen das an unserem Beispiel testen mit den gleichen Eingaben wie beim klassischen Regelwerk. Zum Vergleich erhalten wir die in Abbildung 11 dargestellte Lösungsfuzzymenge.

Nach der Defuzzifizierung erhalten wir einen scharfen Wert von 57.16, welcher dem Wert des klassischen Regelwerks sehr nahe kommt. Das gezeigte Beispiel deutet an, dass bei komplexen Aufgabenstellungen, die die Gefahr der kombinatorischen Explosion in sich birgen, die Combs-Methode eine sinnvolle Alternative darstellen kann.

Ein Vorteil der Combs-Methode sei noch erwähnt. Die Combs-Methode kann auf jedes Fuzzy-Framework angewendet werden. Die Inferenz wird wie bisher durchgeführt und die Implementierungen müssen nicht angepasst werden. Nur bei der Regelgenerierung wird ein anderer Weg gewählt.

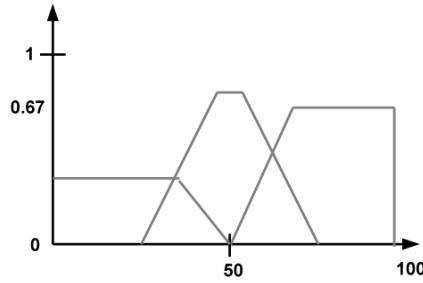


Abbildung 11. Fuzzy-Menge, die durch die Inferenz eines mit der Combs-Methode entstandenen Regelwerks errechnet worden ist.

6 Fazit

Der Seminarbeitrag hat an einem einfachen Beispiel aufgezeigt, dass Fuzzy-Logik zur Erzeugung von künstlicher Intelligenz in Computerspielen gut geeignet ist. Dies ist auch nicht überraschend, denn viele Aufgabenstellungen in Computerspielen sind in ähnlicher Form auch in realen Anwendungen zu finden. Ein weiterer großer Vorteil ist, dass durch die unscharfe umgangssprachliche Integration von Expertenwissen ein natürliches Verhalten durch das Regelwerk erzeugt wird. Dieser Vorteil mündet in mehr Spielspaß in den Computerspielen. Die Spieleindustrie hat den Nutzen der Fuzzy-Logik erkannt und setzt seit Jahren bereits diese Technik ein.

Literatur

- [Buc05] BUCKLAND, Mat: *Programming Game AI by Example*. Wordware Publishing, 2005
- [Com97] COMBS, William E.: *The Combs Method For Rapid Inference*. 1997.
– http://gaia.ecs.csus.edu/~hellerm/EEE222/Atricles/Combs_Fuzzy_Logic/Combs_Rapid_Inference.htm
- [DeL01] DELOURA, Mark: *Game Programming Gems 2*. B&T, 2001
- [DMB04] DAVID M. BOURG, Glenn S.: *AI for Game Developers*. O'Reilly, 2004
- [GJK95] GEORGE J. KLIR, Bo Y.: *Fuzzy Sets and Fuzzy Logic*. Prentice Hall, 1995
- [JK93] JÖRG KAHLERT, Hubert F.: *Fuzzy-Logik und Fuzzy-Control*. Vieweg, 1993
- [MB75] MAMDANI, H. ; BAAKLI, N.: Prescriptive methods for deriving control policy in a fuzzy logical controller. In: *Electron. Lett.* 11 (1975), S. 625–626
- [Rud07] RUDOLPH, Günter: *Fundamente der Computational Intelligence*. Wintersemester 2006/07. – <http://ls11-www.cs.uni-dortmund.de/people/rudolph/teaching/lectures/FCI/WS2006-07/lecture.jsp>
- [Teu95] TEUBER, Peter: *Genetische Algorithmen zur Generierung einer optimalen Regelmenge anhand des ROSA-Relevanzindex*. 1995
- [Zad65] ZADEH, L.A: Fuzzy Sets. In: *Information and Control* 8 (1965), S. 338–353

Neuronale Netze und selbstorganisierte Karten

-Seminararbeit- **Computational Intelligence in Computerspielen**

Andreas Thom

Andreas.Thom@uni-dortmund.de
TU Dortmund

1 Einleitung

Seitdem der Computer auch zur Freizeitgestaltung verwendet wird erhalten Computerspiele eine stetig wachsende Bedeutung, so dass versucht wird, das Verhalten der Computergegner (englisch Non-Player-Character NPC) durch künstliche Intelligenz zu verbessern. Dem Computergegner soll ermöglicht werden, ein „intelligentes“ Verhalten abhängig von der momentanen Situation zu simulieren.

Ein mögliches Konzept zur Verbesserung dieses Verhaltens bieten die neuronalen Netze (NN) oder die selbstorganisierten Karten, auch Self-Organizing Feature Maps (SOM) genannt. NN sind, in Analogie zum menschlichen Gehirn, informationsverarbeitende Systeme, die aus einer Vielzahl von Einheiten (Neuronen) bestehen, welche mittels Verbindungen, gerichteter Kanten, Informationen übertragen und anschließend einen von der Eingabe abhängigen Output berechnen [1]. Dieser Output ist für die Aktion des NPCs verantwortlich. Der klare Vorteil NN im Vergleich zu traditionellen Methoden zur Verbesserung des Verhaltens der NPCs in einem Computerspiel, z.B. State Machines, liegt in ihrer starken Generalisierbarkeit. Damit ist gemeint, dass NN kein komplettes Vorwissen über auftretende Situationen im Spiel und den daraus resultierenden Aktionen im Spiel benötigen. Daher eignet sich die Methode der NN besonders für Probleme, für die gar kein a priori Modell oder zumindest kein exaktes a priori Modell existiert.

1.1 Überblick

Ziel dieser Seminararbeit ist es, eine Einführung in das Gebiet der NN und der Selbstorganisierten Karten (SOM) zu geben. Hierzu wird in Kapitel 2 ein Blick auf das biologische Vorbild der NN, das menschliche Gehirn, gelegt, um anschließend in Kapitel 3 auf die einzelnen Bestandteile und die Struktur eines NN einzugehen. Kapitel 4 befasst sich mit der Lernfähigkeit von NN. Diese besitzen die Fähigkeit selbstständig aus Trainingsbeispielen zu lernen, ohne dass das Netz dazu neu programmiert werden muss. Auf die Selbstorganisierten Karten (SOM), dass heißt

auf ihren Aufbau, die Art zu lernen und auf Unterschiede zu den vorher besprochenen NN wird in Kapitel 5 eingegangen. Nachdem die Methoden NN und SOM durch die vorherigen Kapitel genau erklärt worden sind, wird in Kapitel 6 der Bezug zum eigentlichen Projekthema „Computational Intelligence bei Computerspielen“ hergestellt. Wo bieten sich Einsatzmöglichkeiten der NN/SOM und welchen Problemen muss man sich bei der Realisierung NN/SOM im Bereich der Computational Intelligence in Computerspielen stellen.

2 Biologische Grundlagen

Bevor die künstlichen NN betrachtet werden, sollte zum besseren Verständnis des technischen Modells das biologische Vorbild im Vorfeld erörtert werden. Künstliche neuronale Netze sind den Nervenzellen und Zellverbänden von Tieren und Menschen nachempfunden. Das Gehirn, speziell die Neuronen, ist für die Informationsverarbeitung von entscheidender Bedeutung. Durch die verschiedenen sensorischen Systeme werden die Informationen gesammelt und an das Gehirn weitergeleitet. Abbildung 1 zeigt den schematischen Aufbau einer Nervenzelle mit Synapsen.

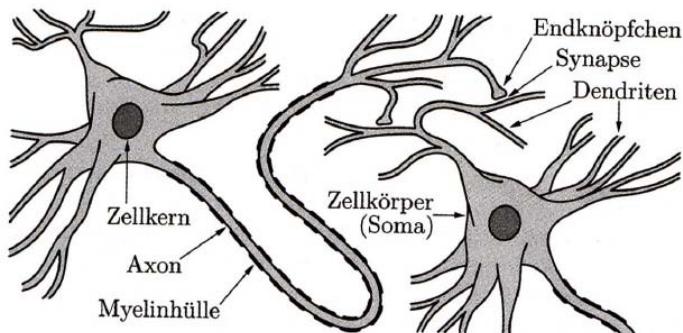


Abbildung 1. Struktur eines biologischen Neurons und Kommunikation zwischen Neuronen. Bild entnommen aus [2].

Eine Nervenzelle oder Neuron besteht aus einem Zellkörper, in welchen der Zellkern eingebettet ist, dem Axon (Nervenfaser), welches für die Weiterleitung der elektrischen Impulse zuständig ist. Dieses Axon kann eine Länge von einigen Millimetern bis zu fast einem Meter besitzen. Am Ende eines Axons bilden sich Verdickungen, sogenannte Endknöpfchen. Die Eingangssignale der Zelle werden durch die Dendriten, sehr dünne röhrenförmige Verästelungen der Zelle, aufgenommen. Eine „Verbindung“ zwischen den Dendriten und dem Axon wird als Synapse bezeichnet. Über die Synapsen werden die Nervensignale an die nachfolgende Nervenzelle weitergeleitet [1].

Die Signalverarbeitung zwischen zwei Neuronen läuft genauer betrachtet wie folgt ab:

1. Signalweitergabe: Die Endknöpfchen der Axone umfassen kleine Bläschen, welche Neurotransmitter enthalten. Erreicht ein Nervenimpuls (elektrischer Impuls) ein Endknöpfchen, so folgt darauf eine Ausschüttung der Neurotransmitter.
2. Signalaufnahme: Die von den Endknöpfchen freigesetzten Neurotransmitter diffundieren durch den schmalen Spalt zwischen Endknöpfchen und Dendriten und bewirken eine Strukturänderung auf Dendritenseite. Hierdurch verändert sich das elektrische Potential der Dendriten. Je nach Potentialänderung kann hier zwischen exzitatorischen (erregend) oder inhibitorischen (hemmend) Synapsen unterscheiden.
3. Signalverarbeitung: Die Potentialänderungen werden am Zellkörper akkumuliert, und falls ein gewisser Schwellenwert überschritten wird, leitet das Axon dieses elektrische Potential weiter (feuert).

Es kann also festgehalten werden, dass sich die Informationsverarbeitung im menschlichen Gehirn aus der Veränderung des elektrischen Potentials und von der Anzahl der Nervenimpulse, die ein Neuron pro Sekunde weiterleitet, zusammensetzt.

Da das menschliche Gehirn aber schätzungsweise aus 10-500 Milliarden [3] Nervenzellen besteht, sollte verständlich sein, dass NN weniger komplex sind. Sie werden meist zur Lösung eines Problems geschaffen, wodurch die Komplexität deutlich abnimmt. Das menschliche Gehirn hingegen hat die Fähigkeit, verschiedene Probleme gleichzeitig zu lösen, unter Verwendung verteilter Bereiche im Gehirn. „We still have a long way to go...“ [3].

3 Grundlagen neuronaler Netze

Nachdem im vorangegangenen Abschnitt das biologische Vorbild der NN erläutert wurde, sollen im folgenden Kapitel die formalen Aspekte der NN untersucht werden. Hierzu wird zuerst ein Blick auf die wesentliche Komponente eines NN, das Neuron, gelegt, um anschließend aus einer Vielzahl von zusammenhängenden Neuronen ein komplettes NN zu entwickeln.

3.1 Das Neuron

Das künstliche Neuron ist der wesentliche Bestandteil eines künstlichen neuronalen Netzes. Es hat die Aufgabe, ankommende Informationen zu verarbeiten und abhängig von der Eingabe zu entscheiden, ob das Neuron aktiviert wird (feuert)

oder nicht. Formal kann ein Neuron als ein Tupel $(\vec{x}, \vec{w}, f_a, f_{net}, net, f_o, o)$ definiert werden bestehend aus einem Eingabevektor $\vec{x} = (x_1, \dots, x_n)$ mit $x_i \in \mathbb{R}$, dem Gewichtsvektor $\vec{w} = (w_1, \dots, w_n)$ mit $w_i \in \mathbb{R}$, der Aktivierungsfunktion f_a mit $f_a : \mathbb{R} \rightarrow \mathbb{R}$, der Netzberechnung $f_{net} : (\mathbb{R} \times \mathbb{R})^U \rightarrow \mathbb{R}$ und einer Funktion zur Ausgabeberechnung $f_o : \mathbb{R} \rightarrow \mathbb{R}$. Das exakte Zusammenspiel der oben beschriebenen Komponenten wird in Abbildung 2 dargestellt.

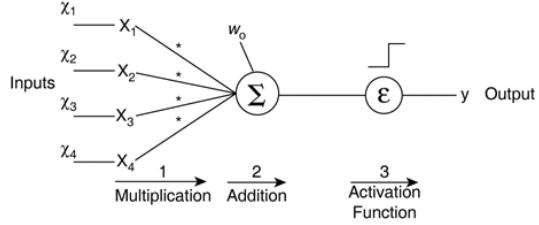


Abbildung 2. Graphische Darstellung eines künstlichen Neurons mit vier Eingängen, Gewichtungen, Netzberechnung, Aktivierungsfunktion und einem Ausgang [4].

Das Neuron, die Verarbeitungseinheit eines NNs, kann als einfacher Prozessor, der abhängig von seinem aktuellen Zustand (der Aktivierung) und der aktuellen Eingabe einen neuen Zustand berechnet, betrachtet werden. Die Arbeitsweise eines Neurons wird im folgenden genauer beschrieben:

- Vektor von Eingangssignalen: \vec{x} mit $x_i \in \mathbb{R}$ stellt den Vektor der Eingangssignale für das Neuron dar.
- Gewichtsvektor: \vec{w} mit $w_i \in \mathbb{R}$ stellt den Gewichtsvektor der eingehen Kan ten dar. Jedes Inputsignal x_i wird mit einem Gewicht w_i assoziiert, welches das anliegende Signal verstärken oder abschwächen soll.
- Netzfunktion: f_{net} berechnet aufgrund der gewichteten Eingaben die eigent liche Netzeingabe für das jeweilige Neuron. Hier wird meist zwischen zwei Arten der Netzeingabeberechnung unterschieden.
 - summation unit (SU): Das Standardverfahren zur Berechnung des Net zinputs summiert die gewichteten Eingaben aller Eingangssignale auf.

$$net = \sum_{i=1}^I x_i w_i \quad (1)$$

- product units (PU): Eine alternative Berechnung des Netzinputs potenziert die Eingangssignale mit den korrespondierenden Gewichtungen und multipliziert dies zum eigentlichen Netzinput auf.

$$net = \prod_{i=1}^I x_i^{w_i} \quad (2)$$

- Aktivierungsfunktion: Jedem Neuron kann generell eine eigene Aktivierungsfunktion zugewiesen werden. In der Regel besitzen jedoch alle Neuronen dieselbe Aktivierungsfunktion. Durch die Aktivierungsfunktion f_a wird bestimmt, ob das Neuron aktiviert ist, also feuert oder nicht. Diese Aktivierung ist abhängig von der Netzeingabe und einem Schwellenwert (threshold Θ).

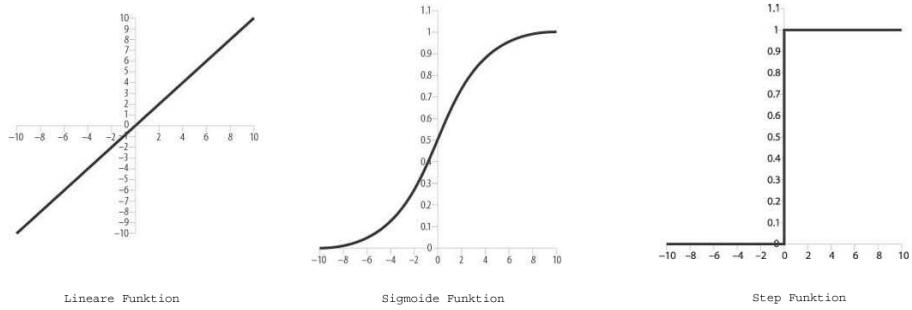


Abbildung 3. lineare, sigmoide und step Aktivierungsfunktion [4].

- Lineare Funktion mit $\Theta = 0$:

$$f_a(\text{net} - \Theta) = \text{net} - \Theta \quad (3)$$

- sigmoide Funktion mit $\Theta = 0$:

$$f_a(\text{net} - \Theta) = \frac{1}{1 + \exp^{-\lambda(\text{net}-\Theta)}} \quad (4)$$

Die am meisten verwendete Aktivierungsfunktion ist die *sigmoide* Funktion. Die Ausgabe der sigmoiden Funktion liegt im Intervall $[0, 1]$ für alle Eingabewerte. Der Parameter λ kontrolliert die Schrittweite der Funktion.

- Step Funktion mit $\Theta > 0$:

$$f_a(\text{net} - \Theta) = \begin{cases} \beta_1 & \text{falls } \text{net} \geq 0 \\ \beta_2 & \text{falls } \text{net} < 0 \end{cases} \quad (5)$$

Ein Neuron, welches die *step* Funktion verwendet „feuert“, falls die gewichteten Eingangssignale einen bestimmten Wert, den Threshold, erreichen oder übersteigen.

- Ausgabeberechnung: Jedes Neuron kann eine eigene Ausgabefunktion verwenden, die die Aktivierung des Neurons in seine Ausgabe überführt. Üblicherweise sind diese jedoch in einem NN gleich. Die einfachste Ausgabefunktion $f_o : \mathbb{R} \rightarrow \mathbb{R}$ ist die Identitätsfunktion, welche also das Ergebnis der Aktivierungsfunktion direkt an den Ausgang weiterleitet. Eine weitere einfache Ausgabefunktion realisiert eine binäre Zuweisung $f_o : \mathbb{R} \rightarrow \{0, 1\}$. Falls das Neuron feuert wird also eine 1 ausgegeben, ansonsten eine 0.

Der Begriff der *externen Eingabefunktion* wird nicht in allen Modelldefinitionen explizit angegeben. Da jedoch im folgenden Definitionen und Beschreibungen verschiedener NN den Begriff der *externen Eingabefunktion* verwenden, wird an dieser Stelle eine kurze Erklärung aufgeführt.

Die externe Eingabefunktion *ex* stellt die Schnittstelle der NN mit der *Umwelt* oder dem System, in welches diese eingebettet sind dar. Hier wird also die *externe* Eingabe für das Netz erzeugt, welche für die Zustandveränderungen der Neuronen verantwortlich ist. Diese Eingabefunktion steht meist nur den Neuronen auf der Eingabeschicht zur Verfügung.

3.2 Neuronale Netze

Nach [4] schränkte Rosenblatt das Modell eines NN auf eine Schicht ein. Durch die Anzahl der Neuronen in solch einem *Single-Layer-Perceptron* wurde der Ergebnisraum bestimmt. Jedoch führte diese Einschränkung auf eine Schicht zum Problem der linearen Separierbarkeit. Einfache Probleme wie das *XOR-Problem* konnten nicht gelöst werden. Daraufhin wurde die Forschungsförderung auf dem Gebiet der NN für 15 Jahre praktisch eingestellt.

Der Weg aus dieser Sackgasse sind die mehrschichtigen neuronalen Netze, Multilayer-Perceptrons, welche im Folgenden genauer beschrieben werden.

Nach [1] setzt sich im allgemeinen ein Neuronales Netz aus folgenden Komponenten zusammen.

Definition 1. *Allgemeines Neuronales Netz in Anlehnung an [1]:*

- *Neuronen (Zellen, Elemente, units): Die Neuronen setzen sich wie in Kapitel 3.1 dargestellt zusammen.*
- *Verbindungsnetzwerk der Neuronen: Ein Neuronales Netz kann als gerichteter, gewichteter Graph angesehen werden, wobei die Kanten die gewichteten Verbindungen zwischen den Neuronen darstellen. Das Gewicht der Verbindung von Neuron i nach Neuron j wird durch w_{ij} bezeichnet.*
- *Netzfunktion (f_{net}): Siehe hierzu Kapitel 3.1.*
- *Lernregel: Die Lernregel ist ein Algorithmus, gemäß dem das Neuronale Netz lernt, für eine vorgegebene Eingabe eine gewünschte Ausgabe zu produzieren. Die am häufigsten verwendete Methode, die zum Einlernen eines KNN verwendet wird, ist die Modifikation der Gewichte an den Kanten. Hierzu wird im Allgemeinen das Verfahren der Backpropagation (Fehlerrückgabe) verwendet.*

Ein Neuronales Netz setzt sich wie in Abbildung 4 dargestellt aus mehreren Single-Layer-Perceptrons zusammen, einer Eingabeschicht (input layer), welche die Eingabesignale an die nachfolgenden Schichten weiterleitet (meist unverfälscht), eine oder mehreren inneren Schichten (hidden layers) und einer Ausgabeschicht (output layer), welche die berechneten Ergebnisse zurückliefert. Die

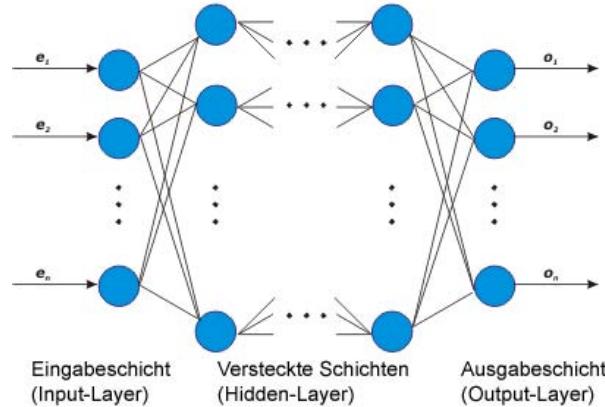


Abbildung 4. Aufbau eines neuronalen Netzes mit Eingabeschicht, mehreren versteckten Schichten und der Ausgabeschicht aus [5].

Kommunikation mit der Umwelt¹ findet nur über die Ein- bzw. Ausgabeschicht statt. Die grundlegende Arbeitsweise des Netzes lässt sich nach [2] in zwei Abläufe unterteilen:

Ruhephase: Die Aktivierung aller Neuronen bleibt konstant. Besonders bei rückgekoppelten Netzen müssen die Zustände der Neuronen solange neu berechnet werden, bis das Netz in einen Ruhezustand konvergiert ist.

Eingabephase: Die Eingabephase, Lernphase, schließt sich der Ruhephase an. Durch die externe Eingabefunktion ex werden die Aktivierungen für die Neuronen auf der Eingabeschicht ermittelt. Sie ist abgeschlossen, wenn alle Eingabeneuronen die externe Eingabe ausgewertet und ihre Aktivierung neu berechnet haben.

Ausgabephase: Schließt sich der Eingabephase an. Hier werden die Ausgaben der einzelnen Neuronen über die durch die eingehenden Kanten und deren Gewichten ermittelten. Alle Neuronen berechnen unabhängig voneinander ihre Netzeingabe, Aktivierung und Ausgabe. Die Ausgabe der Neuronen auf der Ausgabeschicht codieren die Ausgabe des NNs.

Neuronale Netze lassen sich in die in Abbildung 5 dargestellten Netztypologien unterteilen.

3.3 Feedforward Netze (MLP)

In Feedforward-Netzen existiert kein Pfad von einem Neuron direkt, oder über zwischengeschaltete Neuronen, zum Ausgangsneuron zurück. Es kann also als

¹ das System, in welches das Neuronale Netz eingebettet ist

azyklischer Graph verstanden werden. Dieser Netztyp kann zusätzlich in *ebenenweise* verbundene Feedforward-Netze und *allgemeine* Feedforward-Netze mit *shortcut connections* partitioniert werden. Die ebenenweise verbundenen Netze lassen nur Verbindungen von Neuronen direkt benachbarter Schichten zu. Allgemeine Feedforward-Netze lassen hingegen auch Verbindungen zwischen Neuronen zu, bei denen eine odere mehrere Schichten übersprungen werden [1].

Als formales Beispiel eines Feedforward-Netzes wird im folgenden das Multilayer-Perceptron, eines der bekanntesten neuronalen Netzen vorgestellt.

Definition 2. *Multilayer-Perceptron in Anlehnung [2]: Ein Multilayer-Perceptron ist ein Neuronales Netz $MLP = (U, W, A, O, NET, ex)$, das die folgenden Charakteristika aufweist:*

- $U = U_1 \cap \dots \cap U_n$ ist eine Menge von Verarbeitungseinheiten, wobei $n \geq 3$ vorausgesetzt wird. Es gilt dabei, $U_i \neq \{\}$ für alle $i \in \{1, \dots, n\}$ und $U_i \cap U_j = \emptyset$ für $i \neq j$. U_1 heißt Eingabeschicht und U_n Ausgabeschicht. Die U_i mit $1 < i < n$ werden innere (versteckte) Schichten genannt.
- Die Netzwerkstruktur ist durch die Abbildung $W : U \times U \rightarrow \mathbb{R}$ festgelegt, wo- bei nur Verbindungen zwischen direkt aufeinanderfolgenden Schichten exis- tieren. Formal gilt also $W(u, v) \Rightarrow u \in U_{i+1}$ für alle $i \in \{1, \dots, n-1\}$.
- A ordnet jeder Einheit $u \in U$ eine Aktivierungsfunktion $f_a : \mathbb{R} \rightarrow [0, 1]$ zur Berechnung der Aktivierung a_u zu, mit

$$a_u = A_u(ex(u)) = ex(u)$$

für alle $u \in U_1$ und

$$a_u = A_u(net_u) = f(net_u)$$

für alle $u \in U_i$, ($i \in \{2, \dots, n\}$). $f : \rightarrow [0, 1]$ ist dabei eine für alle Einheiten fest gewählte nicht-lineare Funktion.

- O ordnet jeder Einheit (Neuron) $u \in U$ eine Ausgabefunktion $O_u : \mathbb{R} \rightarrow [0, 1]$ zur Berechnung der Ausgabe o_u zu, wobei $o_u = O_u(a_u) = a_u$ für alle $u \in U$ gilt.
- NET ordnet jeder Einheit $v \in U_i$, mit $2 \leq i \leq n$ eine Netzeingabefunktio- on (Propagierungsfunktion) $NET_v : (\mathbb{R} \times \mathbb{R}^{U_{i-1}} \rightarrow \mathbb{R})$ zur Berechnung der Netzeingabe net_v zu, mit

$$net_v = \sum_{u \in U_{i-1}} W(u, v) * o_u + \Theta$$

$\Theta \in \mathbb{R}$ ist der Bias der Einheit v .

- $ex : U_1 \rightarrow [0, 1]$ ordnet jeder Eingabeeinheit $u \in U_1$ ihre externe Eingabe $ex_u = ex(u)$ zu.

3.4 Rückgekoppelte Netze (Rekurrente Netze)

Bei den rückgekoppelten Netzen wird eine Unterteilung in drei verschiedene Klassen vorgenommen: die Klasse der Netze mit direkten Rückkopplungen (*direct feedback*), der indirekt rückgekoppelten Netzen (*indirect feedback*), die vollständig verbundenen Netze und als letztes in die Klasse der Netze, welche Rückkopplungen innerhalb einer Schicht (*lateral feedback*) erlauben.

Direct feedback: Ein Neuron kann seine Ausgabe über eine Verbindung wieder in seine Eingabe zurückleiten. Die Ausgabe kann hierdurch verstärkt oder geschwächt werden.

Indirect feedback: Neuronen höherer Schichten leiten ihre berechnete Ausgabe in den Eingang von Neuronen in niederen Schichten zurück. Dadurch können bestimmte Bereiche im Netz verstärkt angesprochen werden.

Vollständig verbunden: Hier existieren Verbindungen zwischen allen Neuronen. Sie sind auch unter dem Namen Hopfield-Netze siehe Abschnitt 3.5 bekannt geworden.

Lateral feedback: Netze dieser Form werden auch *winner takes all-Netze* genannt. Jedes Neuron erhält eine hemmende Verbindung zu den anderen Neuronen in der Schicht. Durch diese Verbindungen setzt sich das am stärksten aktivierte Neuron gegen die anderen Neuronen durch. Es wird folglich nur ein Neuron, der Gewinner, aktiviert.

Die Netzstruktur des Gehirns hat eher einen rückgekoppelten Aufbau als eine gerichtete Struktur. Rekurrente Netze sind dem menschlichen Gehirn folglich ähnlicher als die bisher vorgestellten Feedforward-Netzen. Da die Aktivierung wieder in das Neuron geleitet wird, welches diese erzeugt hat, können rekurrente-Netze interne Zustände (Aktivierungen) der einzelnen Neuronen speichern, was an das menschliche Kurzzeitgedächtnis erinnert. Durch diese zusätzlichen Kanten kann die Berechnung des Netzes weniger systematisch erfolgen als in Feedforward-Netzen. Zusätzlich neigen rekurrente Netze bei paralleler Aktivierung zur Oszillation. Bei asynchroner Aktivierung wird jedoch ein stabiler Zustand erreicht. Rekurrente Netze haben den zusätzlichen Nachteil der möglichen Instabilität oder auch der Möglichkeit, ein unvorhersehbares chaotisches Verhalten zu zeigen. Die Vorteile solcher Netze liegt wiederum in der Komplexität gegenüber feedforward-Netzen und dem Modellieren des Systems mit Zuständen[6].

3.5 Hopfield-Netze

Die Hopfield-Netze sind die bekanntesten rekurrenten Netze. Hopfield-Netze kommen in vielen Bereichen z.B. bei Optimierungsproblemen, siehe hierzu [1] Kapitel 17.4, oder als assoziativer Speicher zum Einsatz.

Definition 3. *Hopfield-Netz in Anlehnung an [2] Ein Hopfield-Netz ist ein Neuronales Netz $HN = (U, W, A, O, NET, es)$, wobei gilt:*

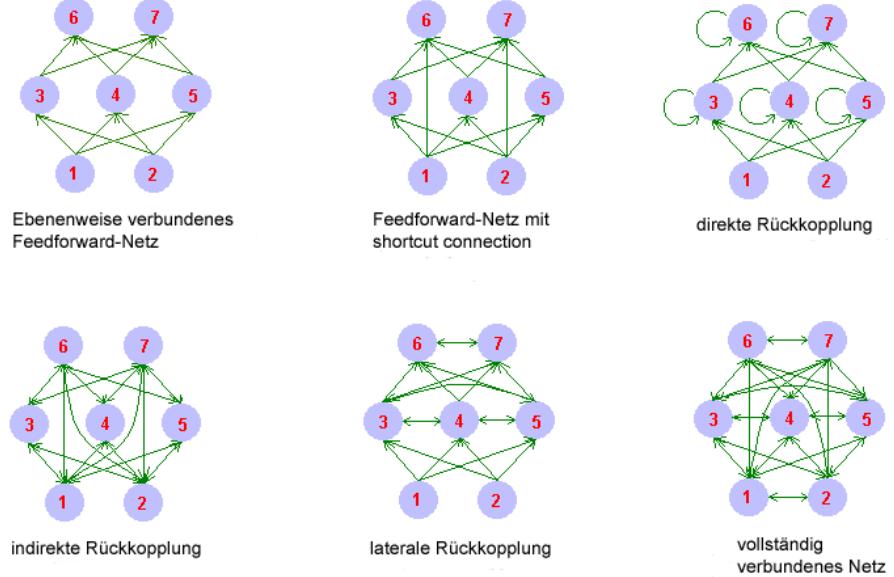


Abbildung 5. Grafische Darstellung der Netztopologien aus [5].

- Die Netzwerkstruktur $W : U \times U \rightarrow \mathbb{R}$ ist symmetrisch und verbindet keine Einheit mit sich selbst, d.h. für alle $u, u' \in U$ gilt $W(u, u') = W(u', u)$ und $W(u, u) = 0$.
- A ordnet jedem Neuron $u \in U$ eine Aktivierungsfunktion $A_u : \mathbb{R} \rightarrow \{-1, +1\}$ zur Berechnung der Aktivierung a_u zu mit

$$a_u = \begin{cases} +1 & \text{falls } \text{net}_u \geq \Theta_u \\ -1 & \text{sonst} \end{cases} \quad (6)$$

wobei der Schwellenwert $\Theta_u \in \mathbb{R}$ und für jedes $u \in U$ fest vorgegeben ist.

- O ordnet jedem Neuron $u \in U$ eine Ausgabefunktion $O_u : \mathbb{R} \rightarrow \mathbb{R}$ zur Berechnung der Ausgabe o_u zu, wobei O_u die Identität ist, d.h. es gilt $o_u = a_u$.
- NET ordnet jeder Einheit $u \in U$ eine lineare Netzeingabefunktion (Propagierungsfunktion) $\text{NET}_u : (\mathbb{R} \times \mathbb{R})^U \rightarrow \mathbb{R}$ zur Berechnung der Netzeingabe net_u zu, es gilt

$$\text{net}_u = \sum_{u' \in U} W(u', u) * o_{u'} = \sum_{u' \in U \setminus \{u\}} W(u', u) * o_{u'}$$

für alle $u \in U$.

- $\text{ex} : U \rightarrow \{-1, +1\}$ ordnet jedem Neuron $u \in U$ seine externe Eingabe $\text{ex}(u)$ zu.

Das Hopfield-Netz ist also ein einfaches NN, welches bestimmte Daten oder Muster in ähnlicher Weise wie das menschliche Gehirn speichern kann. Ein vollständiges Muster kann selbst dann noch abgerufen werden, wenn dem Netz nur Teile des Musters vorgelegt werden. Das Netz wird mit einem bestimmten Muster von feuernden und nicht feuерnden Neuronen gestartet. Durch die hohe Anzahl an Verbindungen und Rückkopplungen löst schon die Aktivierung eines Neurons eine Kettenreaktion aus. Weitere Neuronen werden, angeregt durch das vorherige Neuron, im nächsten Iterationsschritt feuern. Diese Neuronen erregen dann im nächsten Iterationsschritt weitere Neuronen solange, bis ein stabiler Ruhezustand erreicht wird. Das Erreichen dieses Zustandes ist eine zentrale Eigenschaft der Hopfield-Netze, die es überhaupt erst für Simulationszwecke nützlich macht. Das Hopfield-Netz wird, nachdem es gestartet wurde, in dem Endzustand landen, der dem Startmuster am ähnlichsten war. Dieser Endzustand kann in die Verbindungen des Netzes „eingebrannt“ werden, was die Erinnerung des Netzes repräsentiert.

4 Lernen in neuronalen Netzen

Nachdem die Struktur und die Arbeitsweise verschiedener Netztypen dargestellt wurde, wird sich das folgende Kapitel mit der interessanten Komponente des Lernens in NN beschäftigen. Die prinzipiellen Möglichkeiten einem NN Wissen beizubringen sind [1]:

- Entwicklung neuer Verbindungen,
- Löschen existierender Verbindungen,
- Modifikation der Stärke w_{ij} von Verbindungen,
- Modifikation des Schwellenwertes von Neuronen,
- Modifikation der Aktivierungs-, Propagierungs- oder Ausgabefunktion,
- Entwicklung neuer Zellen oder
- Löschen von Zellen.

Die zum Lernen am häufigsten angewandte Methode ist die Modifikation der Gewichte der Verbindungen. Dies schließt z.B. das Löschen und Hinzufügen von Verbindungen mit ein, indem die Kantengewichte auf 0 gesetzt werden. Die Anpassung der Aktivierungs-, Propagierungs- oder Ausgabefunktion ist noch nicht sehr verbreitet.

Hebbsche Lernregel: Die Hebbsche Lernregel ist die älteste und einfachste Lernregel. Hierbei werden die Gewichtswerte auf Basis der Beziehung der Aktivierungen von Neuronen angepasst. Wenn Neuron j eine Eingabe von Neuron i erhält und beide Neuronen gleichzeitig stark aktiviert sind, dann erhöht sich das Gewicht auf der Verbindung dieser Neuronen.

$$\Delta w_{ij}(t) = \eta o_i a_j$$

Dabei stellt $\Delta w_{ij}(t)$ die Veränderung des Gewichts w_{ij} in Zeitpunkt t dar, η ist die Lernrate, o_i die Ausgabe von Neuron i und a_j die Aktivierung des Nechfolgeneurons j . Die Gewichte werden dann wie folgt aktualisiert:

$$w_{ij}(t) = w_{ij}(t - 1) + \Delta w_{ij}(t)$$

Aufbauend auf der *Hebb'schen* Lernregel haben sich zahlreiche Spezialisierungen entwickelt.

Delta-Regel: Bei der Delta-Regel verhält sich die Gewichtsänderung proportional zur Differenz δ_j der aktuellen Ausgabe o_j und der erwarteten Ausgabe e_j .

$$\begin{aligned}\Delta_{ij} &= \eta * o_i * \delta_j \\ \delta_j &= (e_j - o_j) \text{ und } w_{i+j}(t + 1) = w_{ij} + \Delta w_{ij}\end{aligned}$$

Es wird ersichtlich, dass ein größerer Fehler eine größere Veränderung des Gewichtes zur Folge hat.

Nachdem die Lernregeln Neuronaler Netze dargestellt wurden, muss anschließend eine kurze Einführung in allgemeine Lernarten und die daraus resultierenden Lernmethoden in NN gegeben werden. Die drei allgemeinen Lernarten sind:

1. Überwachtes Lernen (Supervised learning) (genauer betrachtet in 4.1)
2. Uniüberwachtes Lernen (Unsupervised learning) (genauer betrachtet in 4.2)
3. Bestärkendes Lernen (Reinforcement learning)

4.1 Überwachtes Lernen (Supervised Learning)

Beim überwachten Lernen in NN lernt das Netz durch einen *externen* Lehrer, wie es sich bei einer bestimmten Eingabe zu verhalten hat. Konkret bedeutet das, dass der *externe* Lehrer zu jedem Eingabemuster der Trainingsmenge das bestmögliche Ausgabemuster angibt. Die Aufgabe des Lernverfahrens ist es dann, die Gewichte des Netzes so zu ändern, dass das Netz nach mehrmaliger Vorlage der Trainingspattern die Assoziation selbstständig zuordnen kann. Darüber hinaus sollen unbekannte oder ähnliche Pattern korrekt eingeordnet (klassifiziert) werden. Das Prinzip eines überwachten Lernverfahrens arbeitet grundsätzlich immer die folgenden fünf Schritte ab[1]:

1. Entsprechende Aktivierung der Eingabeneuronen bezüglich des Eingabemusters,
2. Vorwärtspropagierung der angelegten Eingaben durch das Netz, so dass ein Ausgabemuster erzeugt wird,
3. Vergleich der erzeugten Ausgabe mit der erwünschten Ausgabe (Trainingspattern) liefert einen Fehlervektor (Delta),

4. Rückwärtspropagierung der Fehler von der Ausgabeschicht zur Eingabe liefert Änderungen der Verbindungsgewichte, um den Fehlervektor zu verringern,
5. Änderung der Gewichte aller Neuronen des Netzes um die vorher berechneten Werte.

Das bekannteste und am häufigsten verwendete supervised learning Verfahren ist das Backpropagation.

Backpropagation Das überwachte Lernverfahren *Backpropagation* bezeichnet die rückwärtige Ausbreitung eines Fehlersignals durch das Netzwerk.

Definition 4. *Backpropagation nach [2]: Gegeben sei ein Multilayer-Perceptron siehe Definition 2 mit $U = U_1 \cup \dots \cup U_n$ und der sigmoiden Aktivierungsfunktion $A_u(\text{net}_u) = f(\text{net}_u)$ für alle $u \in U_i, i \geq 2$, sowie einer festen Lernaufgabe $\tilde{\mathcal{L}}$. Der überwachte Lernalgorithmus, der die Veränderung der Gewichtsstruktur W von MLP nach der Propagation des Eingabemusters $i^{(p)}, p \in \tilde{\mathcal{L}}$ durch*

$$\Delta_p W(u, v) = \eta * \delta_v^p * a_u^p$$

mit $u \in U_{i-1}, v \in U_i, 2 \leq i \leq n$ und $\eta > 0$ bestimmt, wobei

$$\delta_u^{(p)} = \begin{cases} f'(\text{net}_u^{(p)}) (t_u^{(p)} - a_u^{(p)}) & \text{falls } u \in U_n \\ f'(\text{net}_u^{(p)}) \sum_{v \in U_{j+1}} \delta_v^{(p)} W(u, v) & \text{falls } u \in U_j, 2 \leq j \leq n-1 \end{cases}$$

gilt, heißt verallgemeinerte Delta-Regel oder Backpropagation-Algorithmus. Dabei ist $a_u^{(p)}$ die Aktivierung von Neuron u nach der Propagation des Eingabemusters $i^{(p)}$ und $t_v^{(p)}, v \in U_n$ ist die durch das Ausgabemuster $t^{(p)}$ für eine Ausgabeeinheit u_n vorgegebene Ausgabe (Aktivierung).

Zur genaueren Herleitung der verallgemeinerten Delta-Regel siehe [2]. Die Anwendung des Backpropagation-Algorithmus lässt sich in zwei Phasen gliedern.

Erste Phase: Das Netz erhält eine Eingabe, welche *vorwärts* durch das Netz propagiert wird, um die Ausgabe jedes Neurons zu bestimmen.

Zweite Phase: Die Fehlersignale werden für die Ausabeeinheiten durch den Vergleich der erzeugten Ausgabe mit der erwünschten Ausgabe ermittelt und die Gewichtsveränderung der Ausgabeschicht berechnet. Anschließend werden diese Fehlerwerte über die gewichteten Kanten an den vorgesetzten Neuronen vermittelt. So können die Fehlersignale bestimmt werden, um die Gewichtsveränderungen der zu dieser Schicht führenden Verbindungen zu ermitteln. Die Fehlerpropagierung wird fortgesetzt bis keine Schicht mehr existiert.

Die Gewichtsänderung berechnet sich jedoch nicht exakt nach der in Definition 4 vorgestellten verallgemeinerten Delta-Regel. Bei der Berechnung wird die Lernrate η und das Moment $\beta > 0$ miteinbezogen. So ergibt sich durch die Propagation von Muster q eine Gewichtsänderung $\Delta_p W(u, v)$ bezüglich des auf q folgenden Musters p durch

$$\Delta_p W(u, v) = \eta * \delta_v^{(p)} * a_u^{(p)} + \beta \Delta_q W(u, v).$$

Das Backpropagation-Verfahren ist ein Gradientenabstiegsverfahren, siehe hierzu auch [7]. Mit dem Gradientenabstiegsverfahren wird versucht in der Fehlerfläche² ein globales Minimum zu finden. Es muss also eine Konfiguration der Gewichte gefunden werden, bei der die Fehlersumme über alle Trainingsmuster minimal ist. Das Backpropagation-Verfahren garantiert jedoch nicht, dass das globale Minimum der Fehlerwerte gefunden wird. Die Fehlerfläche kann z.B. viele lokale Minima oder starke Richtungsänderungen enthalten. Um diesem Weg zu folgen, ist die Wahl der Lernrate η , die Schrittweite der Verfolgung des Gradienten, von entscheidener Bedeutung. Bei einem zu großen Wert der Lernrate kann der Fehler plötzlich wieder ansteigen und bei einem zu kleinen Wert kann ein lokales Minimum nicht überwunden werden. Um dieses Verhalten zu verringern wird das Moment β eingeführt. Durch das Moment wird versucht, hochfrequente Änderungen der Fehlerfläche herauszufinden. Durch die Addition eines Anteils der letzten Gewichtsänderung erhält das Lernverfahren zusätzlich eine Trägheit, was das Auffinden enger Schluchten verstärkt.

4.2 Unüberwachtes Lernen (Unsupervised Learning)

Beim unüberwachten Lernen (unsupervised learning) existieren zu den möglichen Eingabemustern keine erwünschten Ausgaben, welche die Trainingsmuster als *falsch* oder *richtig* klassifizieren könnten. Der unüberwachte Lernalgorithmus versucht selbstständig Gruppen (Cluster) ähnlicher Eingabevektoren zu identifizieren und diese auf Gruppen ähnlicher oder benachbarter Neuronen abzubilden. Das bekannteste und am häufigsten verwendete Unsupervised Learning Verfahren sind die selbstorganisierten Karten.

Selbstorganisierte Karten (SOM) Motiviert durch die selbstorganisierten Charakteristiken des menschlichen Großhirns entstand die Idee der selbstorganisierten Karten (self-organizing feature maps, SOM). Diese Idee sieht eine Modifikation der NN, die auf Basis des Wettbewerbslernens arbeiten, vor. Für diese Aufgabe werden Neuronale Netze bestehend aus zwei Schichten verwendet. Die erste Schicht ist die Eingabeschicht. Die Anzahl der Neuronen in der Eingabeschicht korrespondiert mit der Anzahl der Komponenten des zu untersuchenden Datensatzes. Die zweite Schicht legt die Anzahl der Cluster fest, in die die Daten

² Wenn der Fehler eines neuronalen Netzes als Funktion der Gewichte des Netzwerks graphisch aufgetragen wird, erhält man eine Fehlerfläche

unterteilt werden sollen. Durch die Gewichte an den Kanten von den Neuronen der Eingangsschicht mit einem Neuron der Wettbewerbsschicht wird festgelegt, welches Neuron zu welcher Gruppe zugeordnet wird. Somit ist jedes Neuron der Wettbewerbsschicht für die Erkennung jeweils einer Klasse von Daten zuständig. Wichtig ist, dass die zu klassifizierenden Daten als Vektoren im \mathbb{R}^n vorliegen, um ein Ähnlichkeitsmaß für die Daten zu definieren. Abbildung 6 zeigt einen zu klassifizierenden Datensatz. Es wird ersichtlich, dass nicht nur von näher beieinander oder weiter voneinander gesprochen werden kann. Es wird eine Nachbarschaftsbeziehung ersichtlich, die mit normalen NNn auf Basis des Wettbewerbslernens nicht repräsentierbar sind. Selbstorganisierte Karten definieren, um diesem Problem zu begegnen, eine topologische Struktur auf der Wettbewerbsschicht. Das heißt, dass die Neuronen dieser Schicht in Form einer Geraden, eines Rechtecks oder eines Hyperquaders angeordnet werden. Nach dem Training sollen dann benachbarte Cluster durch benachbarte Neuronen repräsentiert werden. Nach der Präsentation eines Eingabevektors werden in SOMs nicht nur die Kanten gewichte des Siegerneurons angepasst, sondern auch die Gewichtsvektoren der Neuronen in seiner näheren Umgebung. Daraus resultiert, dass benachbarte Neuronen nah beieinander liegende Cluster repräsentieren.

SOMs werden typischerweise zur Dimensionsreduktion bei komplexen Daten

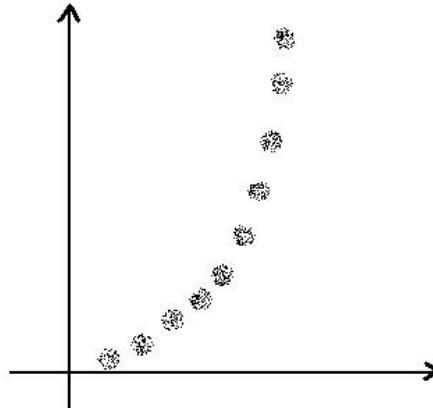


Abbildung 6. Cluster entlang einer parabelförmigen Kurve nach [2].

verwendet. In Abbildung 6 werden also die zweidimensionalen Daten auf eine eindimensionale Struktur reduziert. Formal lassen sich SOMs wie folgt beschreiben. Das Maß für die Übereinstimmung wird über den euklidischen Abstand zwischen der Netzeingabe von Neuron u und der Wettbewerbsschicht bestimmt. Also gilt

$$net_u = \sqrt{\sum_{j=1}^n (W(v_j, u) - o_{v_j})^2}$$

Die Eingabeschicht besteht aus n Neuronen v_1, \dots, v_n , o_{v_j} bezeichnet die Ausgabe von Neuron v_j und liefert die j -te Eingabekomponente des Eingabevektors i . Sei $w^{(u)}$ der Gewichtsvektor von Neuron u auf der Wettbewerbschicht, so lässt sich die Netzeingabe vereinfacht darstellen

$$net_u = \|w^{(u)} - i\|.$$

Das Neuron mit der kleinsten Netzeingabe wird zum Siegerneuron u_s ernannt. Die Gewichtsvektoren eines Neurons u werden dann mittels

$$\Delta w^{(u)} = v(u_s, t) * \sigma(t) * (i - w^{(u)}),$$

angepasst, wobei $\sigma(t)$ die Lernrate mit $0 < \sigma(t) < 1$ ist. Die Lernrate nimmt im allgemeinen im Laufe der Zeit ab, so dass zu Beginn des Lernvorgangs große Änderungen der Gewichtsvektoren erzielt werden können. Gegen Ende des Lernvorgangs nehmen diese Änderungen dementsprechend ab. Also ist der neue Gewichtsvektor von Neuron u

$$w_{neu}^{(u)} = w_{alt}^{(u)} + \Delta w^{(u)}.$$

Bei der Nachbarschaftsfunktion sind verschiedene Arten von Funktionen, wie z.B. die Gauß'sche Glockenkurve oder die Mexican-Hat-Funktion, anwendbar. Hier wird diese durch eine einfache Rechtecksfunktion bezüglich Neuron u_s

$$v(u_s, t) = \begin{cases} 1 & \text{falls } u \in N_{r(t)}(u_s) \\ 0 & \text{sonst} \end{cases}$$

beschrieben. $N_{r(t)}(u_s)$ kennzeichnet die $r(t)$ -Nachbarschaft des Siegerneurons u_s . Diese Nachbarschaft wird erzeugt, indem jedem Neuron u der Wettbewerbschicht seine Position durch einen d -dimensionalen Vektor $p^{(u)}$ zugewiesen wird. Dieser Vektor gibt die Koordinaten von u innerhalb des Gitters \mathbb{Z}^d an. Die Distanz zwischen zwei Neuronen der Wettbewerbschicht wird durch die Supremumsnorm³

$$dist(u, v) = \max_{i \in \{1, \dots, d\}} \{|p_i^{(u)} - p_i^{(v)}|\}$$

berechnet. Die r -Nachbarschaft des Neurons u_s bezüglich dem Radius $r \geq 0$ besteht also aus den Neuronen u , dessen Abstand zum Siegerneuron u_s höchstens r beträgt. Somit folgt

$$N_r(u_s) = \{u | dist(u_s, u) \leq r\}.$$

Der Radius verhält sich äquivalent zur Lernrate $\sigma(t)$, er nimmt also während des Lernvorgangs ab [2]. Somit kann also bestimmt werden, ob Neuron u in der Nachbarschaft vom Siegerneuron u_s liegt und somit die Gewichtsvektoren von u angepasst werden müssen, oder nicht.

³ Abstand entspricht dem Betrag der maximalen Differenz der einzelnen Koordinaten

5 Neuronale Netze und ihre Verwendung in Spielen

Das wesentliche Ziel, welches den Einsatz neuronaler Netze in Computerspielen motiviert, ist das Simulieren menschlichen und intelligenten Verhaltens der NPCs. Diese sollen lernfähig sein und ihr Verhalten gemäß der vorherrschenden Spielsituation anpassen. In folgenden Beispielen werden NN in unterschiedlichen Spielegenren zur Simulierung intelligenten Verhaltens eingesetzt.

5.1 Colin McRae Rally 2.0

Im folgenden Abschnitt soll der Einsatz von NN im Arcade-Modus des bekannten Rennspiels „Colin McRae Rally 2.0“ vorgestellt werden. Im Fokus soll die Entscheidung für die Wahl NN stehen, dass heißt, warum sich NN besonders gut eignen, um das Verhalten der NPCs in diesem Kontext zu verbessern.

In [8] erläutert Jeff Hannan, AI Programmierer von Codemasters, dass die Steue-



Abbildung 7. Screenshot des Fahrverhaltens der Rallywagen in Colin McRae Rally 2. Das滑行 in scharfen Kurven ist gut zu erkennen.

rung eines Rally-Autos auf einer Offroad-Strecke, das heißt einer Strecke die zu großen Teilen von Schlamm und Schotter geprägt ist, eine besondere Herausforderung darstellt. In herkömmlichen Rennspielen muss das Auto (NPCs) „nur“ in die richtige Richtung gelenkt und eventuell die Geschwindigkeit angepasst werden. Auf rutschigen Oberflächen hingegen, soll das Auto frühzeitig in die Kurve einlenken und dann um die Kurve „sliden“, wie in Abbildung 7 dargestellt. Der

Versuch, dieses Verhalten mit einer Menge von Regeln zu simulieren schlug fehl, da nicht alle Spielsituationen durch die Regelmenge abgedeckt werden konnten. Deshalb wurde die Steuerung auf eine einfache Weise mittels eines NNs erzeugt. Die Entscheidung fiel auf ein normales Feedforward-Netz, wie in Definition 2 aufgebaut, welches mit dem Backpropagation Lernverfahren trainiert wurde. Der Schlüssel zum Erfolg war die Wahl der Eingangsvariablen, die jeweils die auftretenden Spielsituationen codieren. Für jede Oberfläche wurde ein eigenes Neuronales Netz trainiert. Die Trainingsdaten, welche zum Training der Netze benötigt wurden, hat Jeff Hannan bei selbst durchgeführten Trainingsspielen gesammelt. Nachdem die Rallywagen eine gute Rundenzeit auf einer vorgegebenen Rennline der Strecken erzielten, zeigten die NN ihr ganzes Potential. Es konnten ohne erneutes Training unterschiedliche Rennlinien zu der bereits trainierten hinzugefügt werden und die Wagen konnten auch auf dieser Rennlinie die Strecke in einer guten Zeit bewältigen. Der NPC (das Auto) kann zwischen verschiedenen Rennlinien wählen, wodurch die Computergegnern ein zufälligeres, menschliches Verhalten simulierten.

5.2 Pacman

Im Zuge der PG511-Computational Intelligence in Games an der Universität Dortmund wurde versucht das Spielverhalten der NPCs in dem Pac-Man Klon „NJam“ durch den Einsatz Neuronaler Netze zu verbessern. In diesem Spiel tritt der Spieler als „Pac-Man“ gegen vier Geister an. In einem Labyrinth versucht er alle Futterpillen zu fressen, ohne sich dabei seinerseits von den Geistern fangen zu lassen. Zusätzlich sind in den Ecken des Spielfeldes sogenannte „Power Pillen“ platziert, durch welche der Spieler für kurze Zeit in der Lage ist, die Geister, NPCs zu fressen.

In [9] wurde der eigentliche Entwurf eines NNs erst nach der Frage der Struktur des Netzes und dem verwendeten Lernverfahren geklärt. Die Entscheidung fiel auf eine einfaches MLP, siehe Definition 2, welches mit dem in Kapitel 4.1 vorgestellten Lernverfahren das Verhalten der NPCs erlernen sollte. Als erster Entwurf wurde das in Abbildung 8 dargestellte Netz entwickelt, welches aus drei Neuronen auf der Eingabeschicht, vier Neuronen auf der Ausgabeschicht besteht und nur eine Schicht in der versteckten Schicht aufweist. Zusätzlich wurde versucht die Komplexität des NNs nicht künstlich zu erhöhen, so dass die Anzahl der Neuronen in der versteckten Schicht möglichst gering gehalten wurde. Die einzelnen Schichten sind wie im normalen Feedforward-Netz (siehe Abschnitt 3.3) vollständig untereinander vernetzt. Wie schon in 5.1 dargestellt wurde, war auch in diesem Fall die Wahl der Eingabevariablen von entscheidener Bedeutung für den Lernerfolg des Netzes. Die drei Eingaben codieren folgende Sensoren:

1. Wandsensor: Nimmt die Umgebung der Geister wahr. Er bestimmt also die Position der Wände.
2. Pacman-Sensor: Teilt den Geistern mit, ob und in welcher Richtung und Entfernung Pacman gesichtet wurde.

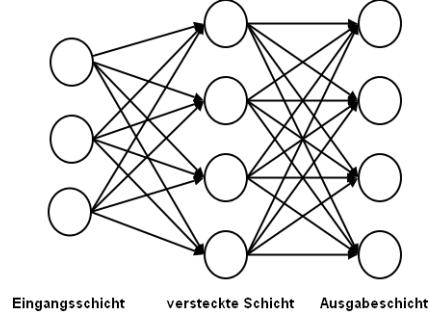


Abbildung 8. Entwurf eines neuronalen Netzes zur Steuerung der Geister im Pacman-klon Njam. Abbildung entnommen aus [9].

3. Pacman-Status: Codiert den Zustand von Pacman, ob er eine Power Pille gefressen hat oder nicht und wie lange diese noch anhält.

Die Ausgabeneuronen repräsentieren die möglichen Bewegungsrichtungen der Geister. Mit dieser Kodierung konnten 263 verschiedene Situationen dargestellt werden, welche nun durch den Backpropagation-Algorithmus erlernt werden mussten. Um die Generalisierbarkeit von NN auszunutzen, wurden die wichtigsten 92 Pattern, mit dem wie oben aufgebauten Netz trainiert. Leider war es nicht möglich ein solches Netz so zu trainieren, dass die nicht trainierten Fälle richtig klassifiziert wurden. Zusätzlich zeigte sich im praktischen Einsatz, dass die Geister mit einem so trainierten Netz dazu neigten, auf geraden Strecken häufig die Richtung zu ändern. Als Konsequenz aus diesem Verhalten wurde die Eingabe angepasst. Konkret heißt das, dass ein neues Eingangsneuron ergänzt wurde, welches die Richtung des Geistes im vorherigen Zug codiert. Die Anzahl der Pattern erhöhte sich auf 308 mögliche Spielsituationen. Um ein besseres Training und eine feinere Abstimmung der Gewichte auf den Kanten zu gewährleisten, wurde zusätzlich die Anzahl der Neuronen in der versteckten Schicht deutlich erhöht. Nach dieser Modifikation des Netzes konnte dieses mit zufällig gewählten 92 Pattern so trainiert werden, dass das Toggeln fast eliminiert wurde und die Geister zudem ein „intelligentes“ Verhalten simulierten. Dieses Netz besteht aus vier Eingangs- und Ausgangsneuronen und besitzt zehn Neuronen auf der versteckten Schicht.

5.3 Bot-Verhalten in Games

Thurau et al. beschreiben in *Combining Self Organizing Maps and Multilayer Perceptrons to Learn Bot-Behavior for a Commercial Game* [10] wie durch die Kombination der in Abschnitt 4.2 dargestellten self-organizing-maps und der in Abschnitt 3.2 vorgestellten MLPs das Verhalten eines Bots⁴ im *First-Person-*

⁴ abgeleitet vom englischen robot

Shooter (FPS) Quake II⁵ modelliert werden kann.

Die zentrale Idee dieses Ansatzes besteht in der Aufzeichnung von durch Menschen gespielten Testspielen, sogenannten *Demos*, um anschließend unter Einsatz von verschiedenen Techniken der Mustererkennung, Data-Mining und der Statistik das menschliche Verhalten der Spieler durch die Bots zu imitieren. Hierzu wird der Zustand eines Spielers zum Zeitpunkt t durch den Vektor \vec{s}_t dargestellt, so dass sich der Zustand in $t + 1$ durch

$$\vec{s}_{t+1} = F(\vec{s}_t, \dots, \vec{s}_{t-k}, \vec{e}_t, \vec{a}_t)$$

beschreiben lässt. Hierbei gibt \vec{e}_t die Umgebungseinflüsse an und \vec{a}_t stellt die Reaktion des Spieler da, welche sich auf seine letzten k Zustände bezieht. Somit kann die Reaktion als Ergebnis von

$$\vec{a}_t = f(\vec{s}_{t+1}, \vec{s}_t, \dots, \vec{s}_{t-k}, \vec{e}_t)$$

verstanden werden. Funktion f sollte also durch die gesammelten Daten erlernbar sein. Die benötigten Daten werden beim Netzwerkverkehr vom Client zum Server mitgeschnitten. In Quake II wird pro Netzpket der Zustand des Spiels, natürlich nur die Daten, die der Spieler auch sehen kann, übermittelt. Hierzu zählen zum Beispiel die Position des Spieler $\vec{o}_{x,y,z}$, sein Blickwinkel $\vec{\delta}_{yaw,pitch,roll}$ und seine Geschwindigkeit $\vec{v}_{x,y,z}$. Graphische Informationen von Gegenständen werden in den Paketen nicht übermittelt. Diese liegen in lokalen Datensätzen vor. Lediglich die Referenzen auf diese Datensätze werden mitgesendet. Da der Spielzustand in diesen Paketen enthalten ist, lässt sich folglich auch die Reaktion des Spielers aus diesen sehr gut extrahieren.

Auf Basis der gesammelten Daten werden nun zwei Schritte zur Modellierung eines menschlichen Verhaltens der Bots durchgeführt:

1. Im ersten Schritt wird eine Dimensionsreduktion der gesammelten Daten mit Hilfe von SOMs durchgeführt. Durch diese werden die Trainingsproben in unterschiedliche *Cluster* unterteilt.
2. Der zweite Schritt weist jedem so entstandenen *Cluster* zwei MLP zu, eins für den Blickwinkel und eins für Geschwindigkeitsanpassung, welche in der Lage sind, mit den vorhandenen Daten des Clusters zu arbeiten. Diese MLPs bilden die Eingabe, also den Zustandsvektor \vec{s}_t , auf eine gültige Reaktion $\vec{a}_t (\vec{s}_t)$ ab.

Wann immer die Reaktion eines Bots in einer Situation \vec{s}_t bestimmt werden muss, wird das MLP, welches dem Gewinnerneuron der SOM zugeordnet ist, zur Reaktionsberechnung aktiviert. Die erste zu trainierende Menge an *Demos* befasst sich mit der Fortbewegung der Bots. Die zweite Trainingsmenge befasst sich mit dem Zielen als Erweiterung des erlernten Fortbewegungsverhalten. Die

⁵ id Software 1997

zugrundeliegenden *Demos* sind auf die zu trainierenden Situationen zugeschnitten. Die verwendeten MLPs werden durch einen Backpropagation-Algorithmus trainiert und enthalten nur 6 Neuronen in der versteckten Schicht.

Während der eigentlichen Testphase werden zunächst die *Demos* bezüglich des Bewegungsverhaltens trainiert. Es liegen 6970 Zustands/Reaktionspaare vor, welche durch 1028 Testproben erlernt werden müssen. In der Praxis zeigt sich, dass durch die SOM unterschiedliche MLPs angesprochen werden, jedoch das Verhalten der Bots keine signifikante Änderung zeigt. Jedes MLP ist für einen bestimmten Zustandsraum spezialisiert. Beim Training konnte festgestellt werden, dass sich mit steigender Anzahl der Wettbewerbsneuronen in der SOM, der gemittelte quadratische Trainingsfehler nur im geringen Maße verringert, wie in Abbildung 1 ersichtlich. Mittlerer quadratischer Fehler:

$$E_d = \frac{1}{n} \sum_{i=1}^n \| \vec{a}_i^{erwünscht} - \vec{a}_i(\vec{s}_i) \|^2$$

mit einer gegebenen Datenmenge d , n Zuständen \vec{s} und gewünschten Reaktionen $\vec{a}_i^{erwünscht}$.

Tabelle 1. Entwicklung des quadratischen Trainingsfehlers während des Trainings und der Offline-Tests bezüglich des Erlernens des Blickwinkels und der Geschwindigkeitsanpassung der Bots aus [10].

#SOM neurons	E_{train} Viewangle	E_{test} Viewangle	E_{train} Velocity	E_{test} Velocity
1	0.340	0.224	0.141	0.058
2	0.173	0.136	0.040	0.025
4	0.105	0.125	0.077	0.015
6	0.149	0.137	0.092	0.010
10	0.088	0.121	0.076	0.081
20	0.203	0.117	0.063	0.064
30	0.106	0.133	0.062	0.017

Als Ergebnis kann festgehalten werden, dass eine Kombination aus SOMs und MLPs mit gesammelten Trainingsproben trainiert werden kann, um menschliches Verhalten zu imitiert.

6 Diskussion

Im Hinblick auf das Seminarthema *Computational Intelligence in Computerspielen* bieten sich eine Vielzahl von Möglichkeiten durch den Einsatz NN die Intelligenz der Computergegner (NPCs) zu verbessern. Diese „moderne“ Methode, die Intelligenz der Gegner zu steuern, wird eher zaghaft eingesetzt. Ursache hierfür

sind der extremem Zeitdruck und die beschränkten finanziellen Mitteln bei der Spieleanwendung. Für die Modellierung der Künstlichen Intelligenz (KI) der Computergegner steht somit nicht unbegrenzte Entwicklungszeit zur Verfügung, so dass die Entwickler dazu neigen, bekannte Methoden zur Steuerung der KI, wie *Finite-state-machines*, einzusetzen. Beim Training der NN und der Kodierung der Testpattern ist der Zeitaufwand nicht zu unterschätzen. Jedoch eignen sie sich besonders gut, um diverse Funktionen zu realisieren, welche fast jedes gewünschte Verhalten modellieren können.

Die in Kapitel 5 aufgeführten Beispiele für den Einsatz NN in Spielen zeigen, dass es sich lohnen kann, neue Methoden zur KI Modellierung einzusetzen. Durch die im Seminar dargestellten Methoden lässt sich ein komplexes NN entwerfen, welches durch die genannten Lernverfahren in der Lage ist, menschliches Verhalten nachzuahmen. Dieses Verhalten ist ein signifikantes Indiz für intelligente Computergegner, denn gerade das Spielen gegen *menschliche* Gegner ist für Computerspieler sehr interessant.

Literatur

- [1] Zell, A.: Simulation Neuronaler Netze. Addison-Wesley, Reading Mass. (1994)
- [2] Nauck, D., Klawonn, F., Kruse, R.: Neuronale Netze und Fuzzy-Systeme. Series Artificial Intelligence. Vieweg, Wiesbaden (1996)
- [3] Engelbrecht, A.P.: Computational Intelligence An Introduction. Wiley-VCH Verlag, Weinheim (2002)
- [4] Champandard, A.J.: AI Game Development: Synthetic Creatures with Learning and Reactive Behaviors. New Riders Publishing (2003)
- [5] Lippe, W.M.: Einführung in Neuronale Netze. Neuro-Fuzzy AG Universität Münster <http://wwwmath.uni-muenster.de/SoftComputing/lehre/material/wwwnnscript/startseite.html>.
- [6] Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice-Hall (1995)
- [7] Heuser, H.: Lehrbuch der Analysis. Teubner, Stuttgart (1995)
- [8] Generation5: Colin McRae Rally 2.0, Interview mit Jeff Hannan AI Programmierer bei Codemasters (2001) <http://www.generation5.org/content/2001/hannan.asp>.
- [9] Projektgruppe 511: Zwischenbericht der Projektgruppe 511, CI in Games (2007) <http://www.ciingames.de>.
- [10] Thurau, C., Bauckhage, C., Sagerer, G.: Combining Self Organizing Maps and Multilayer Perceptrons to Learn Bot-Behaviour for a Commercial Game. In Q. H. Mehdi, N. E. Gough, S.N., ed.: GAME-ON, EUROSIS (2003) 119–
- [11] Rojas, R.: Theorie der neuronalen Netze - Eine systematische Einführung. Springer-Verlag (1996)
- [12] Bourg, D.M., Seemann, G.: AI for Game Developers. O'Reilly Media, Inc. (2004)

Schwarmintelligenz und Computerspiele

Computational Intelligence bei Computerspielen

André Barthelmes

`bathi@nef.wh.uni-dortmund.de,`

Universität Dortmund, 44221 Dortmund, Germany

WWW home page: <http://ls11-www.cs.uni-dortmund.de/>

Zusammenfassung Particle Swarm Optimization und Ant Colony Optimization sind die beiden verbreitetsten Verfahren in dem vergleichsweise jungen Forschungsgebiet der Schwarmintelligenz. Im Bereich klassischer akademischer Probleme wurden diese bereits erfolgreich eingesetzt, während ihre Verwendung für Computerspiele bisher eher selten erfolgt ist. Beide Verfahren werden jeweils vorgestellt und ihre Funktionsweise erläutert. Anschließend werden jeweils einige Einsatzmöglichkeiten dieser Algorithmen im Bereich der Computerspiele beschrieben und die dafür nötigen Anpassungen erläutert. Abschließend wird das in Computerspielen bereits verwendete Verfahren des Flockings beschrieben, welches eine der Schwarmintelligenz ähnlich Technik zur Steuerung von Einheitengruppen verwendet.

1 Einführung

Schwarm Intelligenz (SI) ist ein Teilgebiet der Computational Intelligence (CI) und beschäftigt sich mit dem Studium von kollektivem Verhalten in dezentralisierten, selbst organisierten Systemen. Dabei wird die Vorgehensweise natürlicher Organismen imitiert um intelligente und effiziente Lösungen für Problemstellungen zu entwickeln.

SI Systeme bestehen normalerweise aus einer Population einfacher Agenten, die lokal untereinander und mit ihrer Umgebung interagieren. Es existiert also typischerweise keine zentrale Kontrollstruktur die den Individuen Aktionen vorschreibt und diese koordiniert. Alle Individuen sind gleichwertig und handeln gewissermaßen autonom. Trotzdem entsteht durch die lokalen Interaktionen oft eine Art globales Verhalten. In der Natur können Systeme mit einem solchen Verhalten zum Beispiel bei Ameisenkolonien oder Vogelschwärmen gefunden werden.

Zu den in der Informatik umgesetzten Arten von Schwarmintelligenz gehören vorrangig zwei Verfahren: Die Ant Colony Optimization (ACO) und die Particle Swarm Optimization (PSO). Beide Verfahren wurden in angepassten Varianten bereits erfolgreich zur Lösung klassischer Optimierungsprobleme der Informatik eingesetzt. Die Laufzeiten dieser Verfahren zur Bestimmung einer guten Lösung können jedoch häufig nicht mit hoch spezialisierten Algorithmen, wie sie z.B. für das bekannte Travelling Salesman Problem (TSP) existieren, mithalten. Ihr

Einsatz ist jedoch gerade bei weniger gründlich erforschten Optimierungsproblemen, was auf fast alle Probleme im Bereich der Computerspiele zutrifft, zu empfehlen, da sie leicht an ein beliebiges Problem angepasst werden können und so vergleichsweise schnell gute Lösungen liefern.

2 Ant Colony Optimization

Das Forschungsfeld der Ant Colony Optimization wurde von Ameisenkolonien inspiriert und imitiert das Verhalten realer Ameisen bei der kollektiven Nahrungssuche. Bei der Beobachtung von Ameisenkolonien wurde die interessante Entdeckung gemacht, dass sich die Ameisen auf dem Weg zwischen Nest und Futterquelle mit zunehmender Dauer immer wahrscheinlicher auf dem kürzesten Weg dorthin bewegen.

2.1 Das Double Bridge Experiment

In diesem Experiment wurden Ameisen gegenüber einer Futterstelle angesiedelt, der einzige Weg zwischen dem Nest und dieser Futterstelle führte über eine doppelte Brücke (siehe Abbildung 1). Im Experiment wurde folgendes beobachtet: Wählt man die beiden Wege gleich lang so bevorzugen nach einer gewissen Zeit fast alle Ameisen einen dieser Wege. Nach mehrmaliger Wiederholung des Experiments kann festgestellt werden, dass die Wahl welcher der beiden Wege bevorzugt wird völlig zufällig ist.

Ist die Länge der Wege zur Futterstelle deutlich verschieden, siehe Abbildung 2, so bevorzugen die Ameisen mit zunehmender Zeit immer mehr den kürzeren der beiden Wege. Einzelne zufällige Ameisen benutzen jedoch auch den längeren Weg. Dies geschieht jedoch mit zunehmender Zeit immer seltener.

Die Erklärung für den Ausgang dieser Experimente liegt in der Kommunikation der Ameisen. Ameisen kommunizieren über Pheromone miteinander. Pheromone sind chemische Botenstoffe, die von Ameisen bei der Fortbewegung abgesondert und wahrgenommen werden können. Ameisen beeinflussen ihre Wegwahl nun aufgrund der vorliegenden Pheromonspur vorangegangener Ameisen. Weiterhin verdunsten Pheromone nach einiger Zeit, so dass eine vor langer Zeit gelegte Spur nachfolgende Ameisen nicht mehr beeinflusst.

Zu Beginn befindet sich auf keinem der Wege Pheromon, so dass die Ameisen mit gleicher Wahrscheinlichkeit den linken wie den rechten Weg wählen. Sind beide Strecken unterschiedlich lang, so sind die Ameisen auf dem kürzeren Weg schneller wieder zurück am Nest als die Ameisen auf dem längeren Weg. Da aber gleich viele Ameisen am Anfang den linken wie den rechten Weg gewählt haben, sind die Ameisen auf dem kürzeren Weg schneller und die Pheromonspur übersteigt somit diejenige auf dem längeren Weg. Nachfolgende Ameisen werden somit eher den kürzeren Weg wählen, und dieser Effekt verstärkt sich dann immer weiter, da jede neue Ameise eine weitere Pheromonspur ablegt. Die Konzentration auf dem einen Weg übersteigt irgendwann diejenige des anderen Weges bei weitem, so dass fast alle Ameisen den kürzeren Weg wählen.

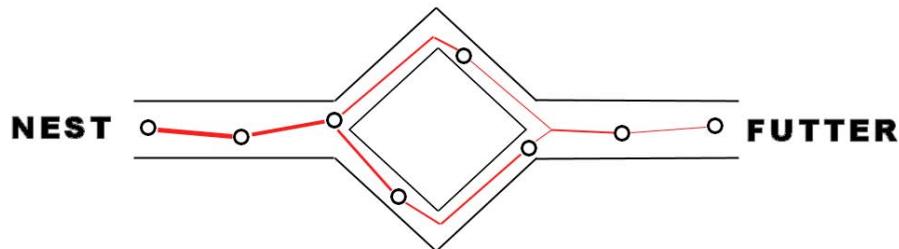


Abbildung 1. Das Double Bridge Experiment mit identischen Wegstrecken. Die schwarzen Kreise stellen Ameisen dar, die roten Linien Pheromonspuren. Die Dicke der Linien gibt die Intensität an. Nach einer gewissen Zeit bevorzugen die Ameisen einen zufällig gewählten Weg.

Wenn beide Strecken gleich lang sind, kann eine geringfügig höhere Konzentration auf dem linken oder rechten Weg dazu führen, dass dieser eher gewählt wird. Da mit der Anzahl der Ameisen auf dem Pfad dieser Effekt noch verstärkt wird übersteigt die Pheromonkonzentration auf dem einen Weg mit fortschreitender Zeit die Konzentration auf dem anderen Weg bei weitem, so dass auf Dauer fast nur noch ein Weg gewählt wird.

2.2 Übertragung in die Informatik

Wie beim biologischen Vorbild erschaffen die künstlichen Ameisen eine Lösung durch eine Reihe probabilistischer Entscheidungen. Sie bewegen sich dabei im Laufe mehrerer Iterationen innerhalb einer künstlichen Struktur die als Graph interpretiert wird. Befindet sich eine Ameise an einem Knoten dieses Graphen und steht vor der Entscheidung über welche Kante sie diesen verlassen soll, so entscheidet sie sich zufällig gemäß der Verteilung, die durch die vorhandenen Pheromonspuren gegeben ist. Dabei sollten alle nicht benutzten Kanten eine geringe Auswahlwahrscheinlichkeit > 0 erhalten. Ist durch eine solche Folge von Entscheidungen schließlich eine Lösung erreicht, so kann in Abhängigkeit von der Qualität der Lösung auf dem gewählten Weg die Pheromonspur entsprechend verstärkt werden. So werden Entscheidungen die zu guten Lösungen führten in den folgenden Iterationen bevorzugt gewählt.

Nach jeder Iteration sollte eine so genannte Evaporation-Phase folgen, in der alte Pheromon-Spuren ähnlich einem Verdunsten beim biologischen Vorbild langsam verschwinden. Auf diesem Weg kann der Einfluss von einmal zufällig schlecht getroffenen Entscheidungen gering gehalten werden. Gute Entscheidungen sind von dieser Verdunstung nur sehr gering betroffen, da sie selbst und alle Entscheidungen, die sich an diesen orientieren, wieder zu guten Lösungen und damit zu

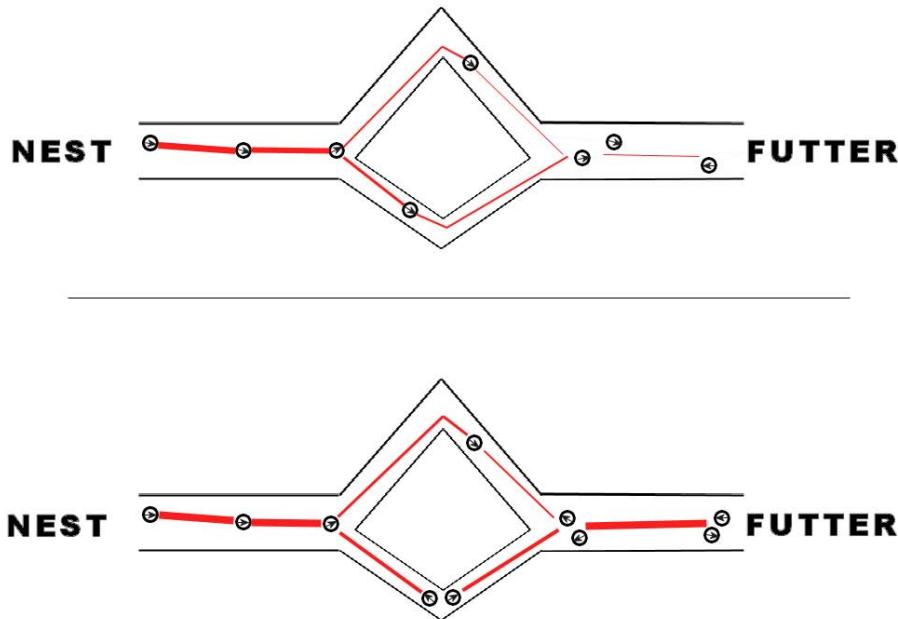


Abbildung 2. Das Double Bridge Experiment mit verschiedenen langen Wegstrecken. Die schwarzen Kreise stellen Ameisen dar, die roten Linien Pheromonspuren. Die Dicke der Linien gibt die Intensität an. Nach einer gewissen Zeit bevorzugen die Ameisen den kürzeren Weg.

frischen und starken Pheromonspuren führen.

Als Abbruchkriterium für den Algorithmus kann eine Kombination der folgenden Ereignisse gewählt werden:

- Die beste gefundene Lösung hat eine gewünschte Qualität erreicht.
- Die beste Lösung würde über eine festgelegte Zeit nicht mehr verbessert.
- Eine bestimmte Iterationszahl wurde erreicht.

Ursprünglich wurde Ant Colony Optimization zunächst zur Lösung kombinatorischer Optimierungsprobleme für Graphen eingesetzt, wie z.B. das Travelling Salesman Problem oder Routing im Netzwerk. Mittlerweile existieren jedoch auch Varianten für binäre oder kontinuierliche Funktionen (vgl. [ACO04], [ACO06]). Die guten Ergebnisse von Ant Colony Optimization Algorithmen bei akademischen Problemen hat dazu geführt, dass sie mittlerweile auch zur Lösung vieler praktischer Probleme in der Industrie eingesetzt werden.

2.3 Einsatzgebiete im Bereich der künstlichen Spielintelligenz

Computer-Go Go ist ein altes chinesisches Spiel mit einer Geschichte von über 3000 Jahren. Jeder Spieler versucht seinen territorialen Einfluss gegenüber seinem Gegenspieler zu vergrößern. Hierzu platzieren beide Spieler abwechselnd schwarze und weiße Steine auf einem Spielfeld der Größe 19 x 19 (siehe Abb. 3). Ziel des Spiels ist es eine möglichst große Fläche des Spielfeldes zu kontrollieren indem man Gebiete mit seinen eigenen Steinen umschließt.

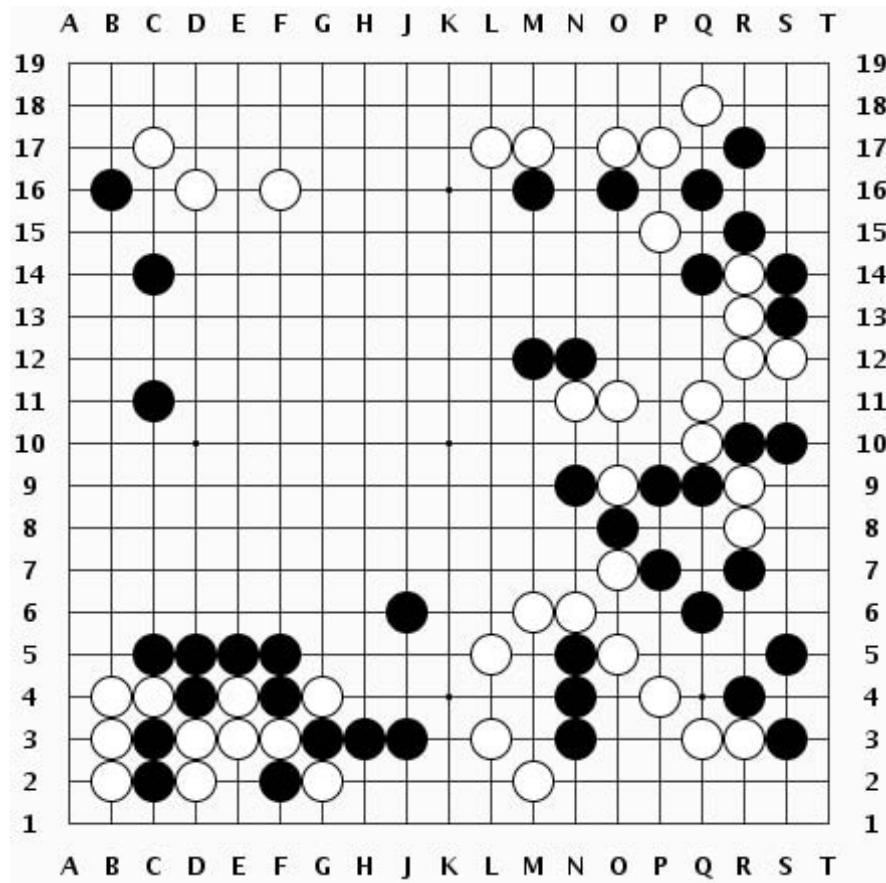


Abbildung 3. Beispiel für den Stand einer Go Partie nach einigen Zügen.

Obwohl Go nach Schach das Spiel ist, für das aktuell die meiste Programmierungsarbeit zur Erstellung künstlicher Computergegner aufgewendet wird, können sich diese lediglich mit durchschnittlichen Go-Spielern messen ([Mue01]). Dies liegt unter anderem daran, dass eine Spielfeldbewertung sehr schwierig ist

und zudem sehr viele mögliche Züge für jede Spielsituation existieren. Dies macht die Analyse mittels Algorithmen, die alle möglichen Züge für eine bestimmt Anzahl von Spielzügen in der Zukunft berechnen sehr schwierig und zeitaufwendig. Im Gegensatz zu Computern können Menschen dank der visuellen Natur des Spielfeldes eine Spielsituation sehr viel besser und schneller beurteilen.

Eine künstliche Intelligenz auf der Basis von Ant Colony Optimization wurde von Tapani Raiko in [Rai04] vorgestellt. Zur Bestimmung wichtiger Spielfeldteile gebiete spielen in jedem Zug eine Menge von Ameisen das Spiel startend mit der aktuellen Spielfeldbelegung zu Ende. Eine einzelne Ameise repräsentiert also eine bestimmte, zufällig gewählte Abfolge von Zügen vom aktuellen Zug bis zum Spielende. Nachdem alle Ameisen ihr Spiel beendet haben werden die Endspielfelder halbtransparent übereinander gelegt. Auf diese Art erhält man eine Spielfeld, in dem jeder Position ein Graustufenwert zugeordnet ist: Felder die dabei in allen von den Ameisen betrachteten Zukunftsvarianten vom schwarzen Spieler gewonnen werden konnten werden komplett schwarz eingefärbt, Felder die immer vom weißen Spieler gewonnen werden konnten komplett weiß. Felder deren Zugehörigkeit zu einem bestimmten Spieler noch nicht feststand, also in den möglichen Zukunftsvarianten von unterschiedlichen Spieler erobert wurden, erhalten eine Graustufung entsprechend ihrem Zugehörigkeitsverhältnis zu den beiden Spielern. Flächen mit einem mittleren Graustufenwert sind dann die Gebiete, die für die künstliche Intelligenz besonders interessant sind, da hier offenbar die Zugehörigkeit zu einem der Spieler in Abhängigkeit der zukünftigen Züge noch leicht verändert werden kann. Anhand der so gesammelten Daten können nun wichtige Spielfeldpositionen bestimmt werden, also Punkte, deren Belegung laut der von den Ameisen gesammelten Daten mit hoher Wahrscheinlichkeit die Graustufeneinfärbung eines großen Gebiets beeinflussen. Die Wahrscheinlichkeit ihrer Signifikanz und die Größe des von ihnen beeinflussten Gebietes geben dann ein Bewertungssystem für den nächsten Spielzug.

In Spielvergleichen mit anderen künstlichen Intelligenzen für Go konnte die von Tapani Raiko erstellte KI zwar mit einigen Algorithmen mithalten, wurde jedoch von der besten aktuellen KI *GNU Go* deutlich geschlagen. Die von Abramson in [Abr90] vorgeschlagene Idee zur Entwicklung einer künstlichen Intelligenz Spiele wiederholt stochastisch zu Ende zu spielen konnte also mit Hilfe von Ant Colony Optimization umgesetzt werden. Sie beruht auf der Annahme dass der Ausgang von zufälligen Spielfortsetzungen ein heuristisches Bewertungssystem liefert, das zumindest für einige Zukunftsmöglichkeiten zutreffend ist.

Strategiespiele Ein nahe liegendes Anwendungsgebiet für den Einsatz von Ant Colony Optimization im Bereich der Computerspiele ist die künstliche Intelligenz von Computergegnern sowohl bei Echtzeit- als auch bei rundenbasierten Strategiespielen.

Ein Grund weshalb sich die Anwendung von Ant Colony Optimization an dieser Stelle anbietet ist, dass von den Computergegner in diesen Spielen eine Reihe von Entscheidungen getroffen werden müssen, die die Grundlage für einen erfolgreichen Ausgang des Spiels haben. Fast alle aktuellen Strategiespiele verfügen über

einen so genannten Tech-Tree (siehe Abb. 4). Jeder Spieler muss verschiedene Gebäude errichten um spezielle Kampfeinheiten ausbilden zu können. Dabei werden zur Errichtung höherwertiger Gebäude, die stärkere Einheiten produzieren können, bestimmte niedrigwertigere Gebäude benötigt. In anderen Gebäuden können meist Verbesserungen für bestimmte Einheiten erforscht werden. Sowohl die Errichtung der Gebäude wie auch das herstellen von Einheiten und die Erforschung der Verbesserungen benötigt Ressourcen und Zeit.

Konzentriert man sich zu sehr auf den Forschungsaspekt des Spiels, um möglichst schnell höherwertigere Gebäude errichten zu können, wird man gegen Strategien die schon von Beginn an sehr viele Einheiten erstellen verlieren. Wählt man jedoch einen Mittelweg aus Forschung und Produktion wird man den ersten Angriff eines Spielers mit Produktionsstrategie abschlagen können und auf Dauer dank des Technologievorsprungs gewinnen. Gegen eine reine Forschungsstrategie wird man aber in den meisten Fällen verlieren, da man den Gegner mit der geringen Anzahl eigener Einheiten nicht ausreichend bei seinen Forschungen stören kann. Eine allgemein gültige optimale Strategie existiert also nicht.

Ein weiterer Grund weshalb sich keine allgemein gültige beste Strategie finden lässt ist, dass in fast allen Spielen dieser Art ein so genanntes Stein-Papier-Schere Prinzip zum Tragen kommt: Wählt man eine feste Strategie lässt sich stets eine Gegenstrategie entwickeln, so gewinnen z.B. Reiter immer gegen Bogenschützen, Bogenschützen immer gegen Fußsoldaten, Fußsoldaten immer gegen Reiter.

Welche Strategie ein Computerspieler verfolgt wird maßgeblich durch seine Entwicklung innerhalb des Tech-Trees bestimmt, also einer Reihe von Entscheidungen wofür eingenommene Ressourcen ausgegeben werden.

Nach einer Betrachtung der Gründe, weshalb viele Spieler Computerspiele spielen (Abwechslung, Herausforderung und Erfolgsgefühl - siehe [Fritz95]), stellen sich also folgende Anforderungen an die künstliche Intelligenz eines Computergegners:

- Sie sollte eine Zufallskomponente enthalten: Der Spieler sollte nie schon vor dem Spiel wissen, was für eine Strategie sein Gegner verfolgen wird, sonst kann er sich darauf einstellen und das Spiel erscheint ihm langweilig.
- Sie sollte sich an die Strategie des Gegners anpassen und diesen zwingen neue Strategien zu entwickeln um Erfolg haben zu können. Dabei darf die KI ab und zu gewinnen, jedoch nicht zu häufig, sonst gibt der Spieler frustriert das Spiel auf.
- Eine optimale Strategie ist nicht verlangt, es reicht hinreichend schnell eine gute Strategie zu entwickeln.

Diese Anforderungen liessen sich vergleichsweise einfach erfüllen, wenn man die KI des Spiels ständig mit CI-Methoden überarbeiten würde. Am Beispiel der Strategiespiele könnte man die Idee verfolgen, Ameisenalgorithmen wie folgt einzusetzen: Man erzeugt einen Hyperbaum (bzw. Graphen) wobei jeder Knoten quasi einen Tech-Tree enthält, also für eine spezielle Folge von Entscheidungen Ressourcen auszugeben steht. Findet nun ein Spiel statt wird die vom Computerspieler gewählte Strategie bzw. Baureihenfolge nach ihrem Erfolg bewertet.

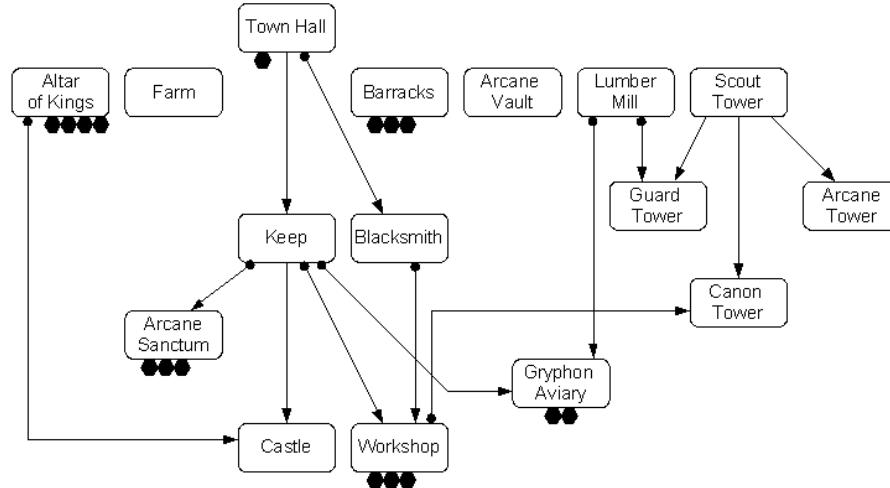


Abbildung 4. Beispiel für einen Tech-Tree (Warcraft III Humans). Die Pfeile geben Gebäude-Upgrades an, Pfeile mit Punkt benötigte Gebäude. Schwarze Sechsecken stehen für unterschiedliche Einheiten die in den Gebäuden hergestellt werden können.

Erfolg bedeutet hierbei, dass der Spieler ein abwechslungsreiches, anspruchsvolles und knapp erfolgreiches Spiel hatte. Jede einzelne Entscheidung der KI im Hyperbaum wird entsprechend in ihrer Auswahlwahrscheinlichkeit verändert und lange zurückliegende Entscheidungsveränderungen langsam vergessen.

Im Bereich der künstlichen Intelligenz für Strategiespiele sind bereits verschiedene CI-Methoden wie das Training neuronaler Netze oder Offline-Learning mittels Evolutionärer Algorithmen erfolgreich zum Einsatz gekommen (vgl. [Pon04]). Die erzeugten Strategien waren denen früherer Spiele, bei denen die Handlung des Computergegner von den Entwicklern fest vorgeschrieben wurde, überlegen. Zur weiteren Verbesserung unter Berücksichtigung der Notwendigkeit, Strategien ständig verändern zu müssen, um in einer vielschichtigen Stein-Papier-Schere Umgebung gute Ergebnisse zu erzielen, bietet der Gedanke der Ant Colony Optimization einen viel versprechenden Ansatz.

3 Particle Swarm Optimization

Das Forschungsfeld der Particle Swarm Optimization (PSO) wurde von Vogel- bzw. Fischschwärm auf Nahrungssuche inspiriert. Zentraler Aspekt dieses Forschungsfeld ist die Koordination einzelner Individuen im Schwarm, welche den zugrunde liegenden, natürlichen Systemen bei der Nahrungssuche enorme Vorteile verschafft.

3.1 Klassische Particle Swarm Optimization

Die klassische Particle Swarm Optimization ist eine mächtige und robuste Methode, um Extrempunkte unbekannter numerischer Funktionen innerhalb eines abgeschlossenen, stetigen Definitionsbereichs zu finden. Die Grundidee beruht darauf, eine Menge sich bewegender Partikel in dem (kontinuierlichen) Suchraum zu platzieren. Dabei hat jedes einzelne Partikel

- eine Position und eine Geschwindigkeit,
- Kenntnis von seiner Position und vom zugehörigen Funktionswert,
- eine Erinnerung an die beste Position die es selbst bis jetzt gefunden hatte,
- Kenntnis der besten (oder alternativ der aktuellen) Positionen und der dazugehörigen Funktionswerte seiner Nachbarn. Hierbei zählt jedes Partikel auch als eigener Nachbar.

Dabei gibt es unterschiedliche Möglichkeiten eine Nachbarschaft zu definieren:

- Physikalisch: Benachbart sind zwei Partikel, wenn ihr Abstand gering ist (teuer, da der Abstand nach jedem Schritt neu berechnet werden muss).
- Sozial: Jedes Partikel hat eine unveränderliche zu Beginn festgelegte Liste seiner Nachbarn.

Mit jedem Berechnungsschritt bewegen sich die Partikel, wobei zur Bestimmung der Geschwindigkeit (Richtung und Betrag) jedes Partikel einen Kompromiss wählt aus (siehe Abb. 5):

- seiner aktuellen Geschwindigkeit.
- einer Bewegung in Richtung seiner besten bisher gefundenen Position.
- einer Bewegung in Richtung der bisher besten von allen Nachbarn gefundenen Position.

Formal:

$$v_{t+1} = c_1 v_t + c_2(p_t - x_t) + c_3(p_{g,t} - x_t) \quad (1)$$

$$x_{t+1} = x_t + v_{t+1} \quad (2)$$

wobei

v_t := Geschwindigkeit zum Zeitpunkt t

x_t := Position zum Zeitpunkt t

p_t := bisher beste Position des jeweiligen Partikels zum Zeitpunkt t

$p_{g,t}$:= beste bisher gefundene Position des besten Nachbarn zum Zeitpunkt t

c_1, c_2, c_3 := soziale / kognitive Vertrauens – Koeffizienten

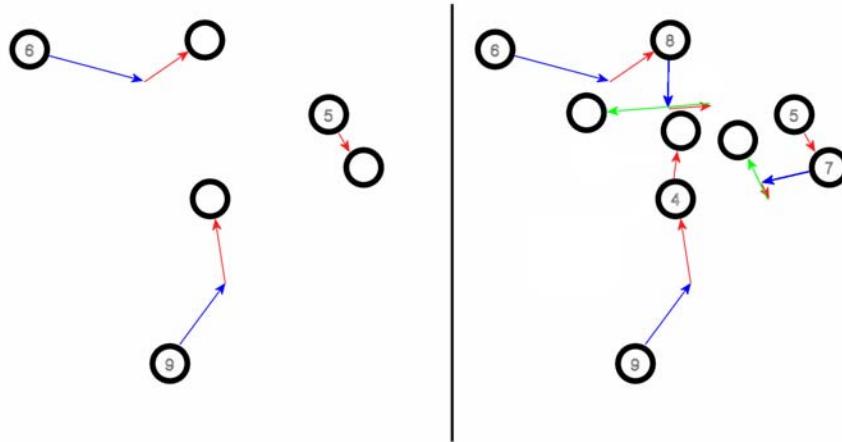


Abbildung 5. Beispiel für einen Berechnungsschritt einer Particle Swarm Optimisierung. Die Zahlen innerhalb der Partikel geben deren Funktionswert an. Die blauen Pfeile stehen für die Teilbewegung in Richtung der besten Position aller Nachbarn, die grünen für die Teilbewegung in Richtung der besten selbst gefundenen Position des Partikels. Die roten Pfeile stehen für die Teilbewegung in Richtung der im letzten Rechenschritt durchgeführten Bewegung.

Durchführung Beim Start des Verfahrens werden zunächst die Positionen der einzelnen Partikel zufällig gewählt. Anschließend bestimmen die Partikel ihre jeweilige Lösungsqualität. Nun betrachtet jedes Partikel die (beste) Position seiner Nachbarn und berechnet seine neue Position (den nächsten Suchpunkt) mittels (1) und (2). Diese beiden Schritte werden solange wiederholt bis ein Abbruchkriterium erreicht ist. Im ersten Schritt sind die Startgeschwindigkeiten v_0 zufällig gewählt. Das Verfahren sorgt also dafür, dass sich die Partikel ein wenig in die Richtung der besten selbst gefundenen Position und ein wenig in die Richtung der bislang besten gefundenen Position bewegen. Die Gewichtung dieser beiden Lösungen wird durch die so genannten sozialen und kognitiven Vertrauens-Koeffizienten festgelegt. Diese werden üblicherweise für jeden Berechnungsschritt innerhalb gewisser Intervalle zufällig gewählt.

Probleme Die Änderung der Geschwindigkeiten besitzt eine Reihe von Parametern, welche maßgeblich die Richtung der Partikel beeinflussen. Die Wahl der richtigen Parameter kann entscheidenden Einfluss auf die Qualität der entstehenden Lösungen haben. So hat sich gezeigt, dass die Geschwindigkeit der Partikel häufig schnell sehr stark anwächst. Oftmals werden im Laufe der Zeit die Partikelgeschwindigkeiten so groß, dass die Partikel-Positionen von den eigentlich interessanten Funktionsstellen weit entfernt sind. Zwar bewegen sich die Partikel immer in Richtung der besten gefundenen Positionen, doch häufig mit einer zu hohen Geschwindigkeit, so dass sie diese weit verfehlten. Um mit fort-

schreitender Laufzeit eine Konvergenz des Verfahrens durch einen möglichst nahe zusammenliegenden Schwarm zu erreicht müssen die Partikel-Geschwindigkeiten also gedämpft werden. Dies darf jedoch nicht zu schnell geschehen, da sonst ein größerer Suchbereich mit potentiellen globalen Optimalstellen nicht mehr erreicht werden kann.

Es existieren mehrere verschiedene Verfahren, die den Schwarm beobachten und manipulieren:

- Additional Chaos : Durch generieren von Zufallszahlen und dem Vergleich mit vorher festgelegten Parametern kann einem Partikel eine komplett neue Geschwindigkeit und Position zugewiesen werden.
- Bouncing : Sollten sich zwei Partikel näher kommen als eine gewisse definierte Distanz, so stoßen sich diese wie Atome mit gleicher Ladung ab.
- Explicit Diversity Control : Der Algorithmus teilt sich in zwei Phasen, eine Attraction Phase und eine Repulsion Phase. Überschreitet die durchschnittliche Abweichung aller Partikel einen gewissen Abstand vom Mittelwert aller Partikel wird zwischen den Phasen umgeschaltet, so dass der Schwarm abwechselnd seine Verschiedenheit erhöht und verringert. Das Umschalten kann durch invertieren der Geschwindigkeitsvektoren erreicht werden.

3.2 Diskrete Particle Swarm Optimization

Auf viele Problemstellungen im Bereich der Computerspiele wie in der Informatik allgemein lässt sich die obigen Kapitel vorgestellte Particle Swarm Optimization nicht anwenden, da sie über keinen stetigen Definitionsbereich verfügen. Viele Probleme, die innerhalb von Computerspielen auftreten, sind als Durchläufe durch spezielle Graphen abbildbar, so z.B. das aufsammeln bestimmter Items innerhalb eines Gangsystems oder die Build-Order von Gebäuden bei Strategiespielen. Das wohl allgemein bekannteste Problem für einen speziellen Graphendurchlauf ist das Travelling Salesman Problem.

Soll ein solches kombinatorisches Optimierungsproblem mittels Particle Swarm Optimization gelöst werden, ist zunächst nicht intuitiv klar, wie sich Begriffe wie Geschwindigkeit und Position von Partikeln übertragen lassen. Der Suchraum ist eine endliche Zustandsmenge und die Kostenfunktion diskret.

Wie man kombinatorische Optimierungsprobleme mit Particle Swarm Optimization lösen kann wird im folgenden kurz anhand des Travelling Salesman Problems erläutert (vgl. [Onwu04]):

Benötigte mathematische Objekte und Operationen Bei einer genaueren Betrachtung von (1) und (2) fällt auf, das zur Durchführung einer Particle Swarm Optimization die folgenden mathematischen Objekte und Operationen benötigt werden:

- Position eines Partikels
- Geschwindigkeit eines Partikels
- Bewegungsfunktion : $(\text{Position}, \text{Geschwindigkeit}) \rightarrow \text{Position}$

- Subtraktionsfunktion : (Position, Position) → Geschwindigkeit
- Additionsfunktion : (Geschwindigkeit, Geschwindigkeit) → Geschwindigkeit
- Skalare Multiplikation : (Skalar, Geschwindigkeit) → Geschwindigkeit

Suchraum und Position Sei $G = \{E_G, V_G\}$ der für das TSP gegebene, gewichtete Graph, mit E_G als Menge der gewichteten Kanten und V_G als Menge der Knoten (numeriert von 1 bis N). Jedes Element aus E_G lässt sich also als Tripel $(i, j, w_{i,j})$ darstellen ($i, j \in V_G, w_{i,j} \in \mathbb{R}^+$). Da nach einem Zyklus gesucht wird, werden Folgen aus $N+1$ jeweils unterschiedlichen Knoten betrachtet, wobei lediglich der letzte Knoten mit dem ersten Knoten übereinstimmen muss. Eine solche Folge heißt N-Zyklus und wird hier als Position betrachtet, womit der Suchraum wie folgt definiert ist : Die endliche Menge aller möglichen N-Zyklen.

Kostenfunktion Betrachtet wird eine gegebene Position $x = (n_1, n_2, \dots, n_N, n_{N+1})$ mit $n_i \in V_G$,

$n_1 = n_{N+1}$, alle Verbindungen (n_i, n_{i+1}) existieren. Um eine Kostenfunktion definieren zu können werden nun die Kosten für alle bis jetzt nicht vorhandenen Verbindungen zwischen zwei Knoten auf einen (genügend großen) Wert L gesetzt, so dass keine Lösung eine mit diesen Kosten versehene Verbindung enthalten kann :

$$L > MAX(w_{i,j}) + (N - 1)(MAX(w_{i,j}) - MIN(w_{i,j})) \quad (3)$$

Die Kostenfunktion ist dann definiert als :

$$f(x) = \sum_{i=1}^{N-1} w_{n_i, n_{i+1}} \quad (4)$$

Die so definierte Funktion hat eine endliche Anzahl an Werten und ihr globales Minimum wird (von einem bestimmten N-Zyklus, der besten Lösung) erreicht.

Geschwindigkeit Benötigt wird ein Operator v, der angewandt auf eine Position eine andere Position ergibt. Es muss hierbei eine Permutation der gegebenen N Elemente geliefert werden. Erforderlich ist also eine Liste von Transpositionen. Sei $\|v\|$ die Länge dieser Liste, dann ist eine Geschwindigkeit definiert als:

$$v = ((i_k, j_k)) \text{ mit } i_k \in V_G, j_k \in V_G, k = 1, \dots, \|v\| \quad (5)$$

Falls zwei unterschiedliche Geschwindigkeiten angewandt auf alle Positionen stets das selbe Ergebnis liefern, heißen sie äquivalent. Die Geschwindigkeit 0 ist äquivalent zur leeren Liste (\emptyset) .

Die entgegengesetzte Geschwindigkeit zu v ist definiert als:

$$\neg v = ((i_{\|v\|-k+1}, j_{\|v\|-k+1})) \quad (6)$$

$\neg v$ führt die selben Transpositionen aus wie v, nur in umgekehrter Reihenfolge. Es ist leicht zu sehen dass $\neg \neg v = v$ und $\neg v + v = \emptyset$

Bewegung (Position plus Geschwindigkeit) Sei x eine Position und v eine Geschwindigkeit so ist $x' = x + v$, wobei man x' durch Anwendung der durch v gegebenen Transpositionen (in der entsprechenden Reihenfolge) auf x erhält.

Subtraktion (Position minus Position) Seien x_1 und x_2 zwei Positionen. Die Differenz $x_1 - x_2 = v$ ist definiert als Geschwindigkeit, also eine *durch einen speziellen Algorithmus* gefundene Liste von Transpositionen. (der Zusatz *durch einem speziellen Algorithmus* ist notwendig, da zwei Geschwindigkeiten äquivalent sein können). Dieser wird so gewählt, dass $x_2 - x_1 = -(x_1 - x_2)$ und $x_1 = x_2 \Rightarrow v = \emptyset$

Addition (Geschwindigkeit plus Geschwindigkeit) Seien v_1 und v_2 zwei Geschwindigkeiten. Um $v_1 \oplus v_2$ zu berechnen wird eine neue Liste von Transpositionen definiert, die zuerst alle Elemente von v_1 gefolgt von den Elementen von v_2 enthält. Optional kann die Liste nun noch verkürzt werden, denn evtl. ist es möglich eine äquivalente Geschwindigkeit mit weniger Transpositionen zu erhalten. Speziell ist diese Operation so definiert, dass $\neg v + v = \emptyset$. Daraus folgt $\|v_1 \oplus v_2\| \leq \|v_1\| + \|v_2\|$ aber nicht unbedingt $v_1 \oplus v_2 = v_2 \oplus v_1$.

Multiplikation (Skalar mal Geschwindigkeit) Sei c ein Skalar aus \mathbb{R} und v eine Geschwindigkeit:

1. Fall : $c = 0$: Es folgt $cv = \emptyset$
2. Fall : $c \in]0, 1]$: Sei $\|cv\|$ die größte Ganzzahl kleiner gleich $c\|v\|$.
So definieren wir : $cv = ((i_k, j_k), k = 1, \dots, \|cv\|)$
3. Fall : $c > 1$: Sei $c = k + c'$ mit $c' \in]0, 1]$ und $k \in \mathbb{N}$
So definieren wir $cv = c'v \oplus \sum_1^k v$
4. Fall : $c < 0$: Da $cv = (-c)\neg v$, kann man sofort einen der Fälle 1-3 betrachten.

Abstand zwischen zwei Punkten Seien x_1 und x_2 zwei Positionen. Der Abstand ist definiert als:

$$d(x_1, x_2) = \|x_2 - x_1\| \quad (7)$$

Anmerkung: Diese Abbildung ist auch ein Metrik, da die folgenden Axiome erfüllt sind :

1. $\|x_2 - x_1\| = \|x_1 - x_2\|$
2. $\|x_2 - x_1\| = 0 \Leftrightarrow x_1 = x_2$
3. $\|x_2 - x_1\| \leq \|x_2 - x_3\| + \|x_3 - x_1\|$

4 Flocking

Ein Anwendungsbereich, das der Particle Swarm Optimization sehr ähnlich ist und sowohl bei Computerspielen als auch in Filmen Anwendung findet ist das so genannte Flocking (vgl. [Bourg04]). Dabei geht es um die Bewegung einer Menge von Spielfiguren oder computeranimierter Charakteren in einer Gruppe. Sowohl die Figuren als auch deren gemeinsames Ziel können hierbei sehr unterschiedlich sein:

- Eine Herde von Schafen die eine Wiese abgrast.
- In Echtzeitstrategiespielen eine Gruppe von militärischen Einheiten die zu einer Position vorrückt.
- In einem First-Person-Shooter die Bewegung einer kleinen Kampfeinheit.

Der erste Flocking Algorithmus wurde 1987 von Craig Reynolds vorgestellt (siehe [Rey87]), in dem der Begriff Boids für die simulierten Flocks (dt. Herden-Mitglieder) geprägt wurde. Diese Implementierung kommt ohne Anführer aus, d.h. kein einzelner bestimmter Boid bestimmt die Bewegungen des Flocks. Der Algorithmus beruht auf drei einfachen Regeln:

- Cohesion (dt. Zusammenhalt): Jede Einheit bewegt sich zu einem Teil in Richtung der durchschnittlichen Position seiner Nachbarn.
- Alignment (dt. Ausrichtung): Jede Einheit bewegt sich zu einem Teil so wie die durchschnittliche Bewegungsrichtung seiner Nachbarn.
- Separation (dt. Trennung): Jede Einheit bewegt sich so, dass sie nicht mit benachbarten Einheiten kollidiert.

Aus diesen Regeln folgt offensichtlich, dass jede Einheit in der Lage sein muss seine Bewegung selbst zu steuern und durchzuführen. Weiterhin muss jede Einheit ihre direkte Umgebung kennen, also wo sich Nachbareinheiten befinden, in welchem Abstand und wohin diese sich bewegen. In wie weit jede Einheit sich über seine Nachbarn, also deren Position, Bewegungsrichtung und Abstand, bewusst ist hängt von ihrem Sichtfeld ab, das durch zwei Parameter bestimmt ist: Dem Kreisradius r und dem Winkel θ (siehe Abb. 6). Allgemein erlaubt es ein großer Sichtradius einen größeren Teil des Flocks zu beobachten und führt zu einer zusammenhängenderen Gruppe, während ein kleinerer Radius die Wahrscheinlichkeit der Aufteilung in mehrere kleinere Gruppen erhöht. Ein großer Sichtwinkel θ führt zu annähernd kreisförmigen Flocks, während die Individuen bei einem kleinen Sichtwinkel dazu tendieren eine Kette zu bilden (siehe Abb. 7). Je nach gewünschtem Ergebnis kann so das Flockverhalten gesteuert werden.

Viele Flocking Algorithmen verwenden Partikel um die zu bewegenden Einheiten zu repräsentieren, wobei jedoch bedacht werden sollte, dass in Computerspielen fast alle Einheiten massive Körper besitzen, d.h. sie können sich nur in eine Richtung bewegen und müssen sich ggf. vorher durch eine Rotation ausrichten. Die Orientierung jeder Einheit muss also immer berücksichtigt werden während sie sich bewegt.

Viele Flocking Algorithmen verwenden Partikel um die zu bewegenden Einheiten zu repräsentieren, wobei jedoch bedacht werden sollte, dass in Computerspielen fast alle Einheiten massive Körper besitzen, d.h. sie können sich nur in eine Richtung bewegen und müssen sich ggf. vorher durch eine Rotation ausrichten. Die Orientierung jeder Einheit muss also immer berücksichtigt werden während sie sich bewegt.

Wichtig für die Steuerung der Einheiten ist außerdem, ein gutes Verhältnis der drei Bewegungskomponenten Cohesion, Alignment und Separation zu finden. Je

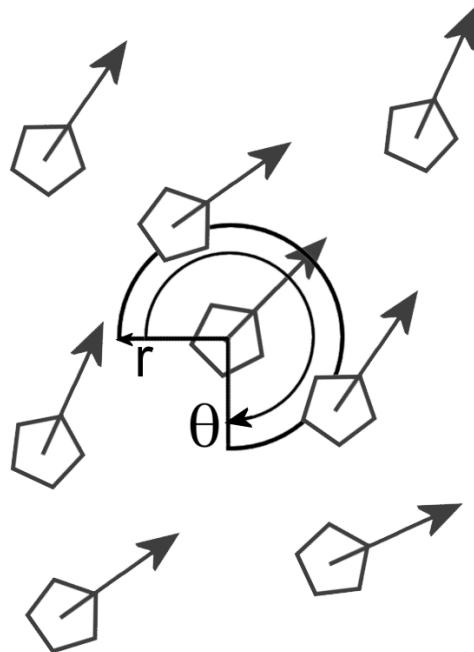


Abbildung 6. Sichtfeld einer Einheit. Jede Einheit kennt nur die Position, die Bewegungsrichtung und den Abstand der Einheiten in ihrem Sichtfeld. Dieses ist durch den Radius r und den Winkel θ festgelegt.

nach Terrain durch das sich der Flock bewegt ist dieses Verhältnis aber unterschiedlich zu wählen, um z.B. beim Umlaufen eines Hindernisses eine Aufspaltung des Flocks zu vermeiden. Eine Erweiterung dieser Algorithmus um zu vermeiden, dass der Flock gegen Objekte der Spielwelt prallt lässt sich vergleichsweise einfach durchführen. Zunächst benötigt jedes Individuum einen Mechanismus um Hindernisse innerhalb einer gewissen Entfernung in seiner Bewegungsrichtung zu sehen. Wird ein Hindernis gefunden muss lediglich eine Bewegungskomponente zur Kollisionsvermeidung zur normalen Bewegung bestehend aus Cohesion, Alignment und Separation addiert werden.

Betrachtet man Particle Swarm Optimization und den Flocking Algorithmus im Vergleich, lassen sich also viele Gemeinsamkeiten feststellen. Beide Verfahren bewegen eine Menge von Individuen innerhalb einer Umgebung und überlassen die Steuerung der einzelnen Partikel jeweils den Individuen selbst. Die einzelnen Bewegungen setzen sich jeweils aus mehreren Komponenten zusammen die dafür sorgen, dass die Partikelmenge ein gewünschtes Verhalten zeigt.

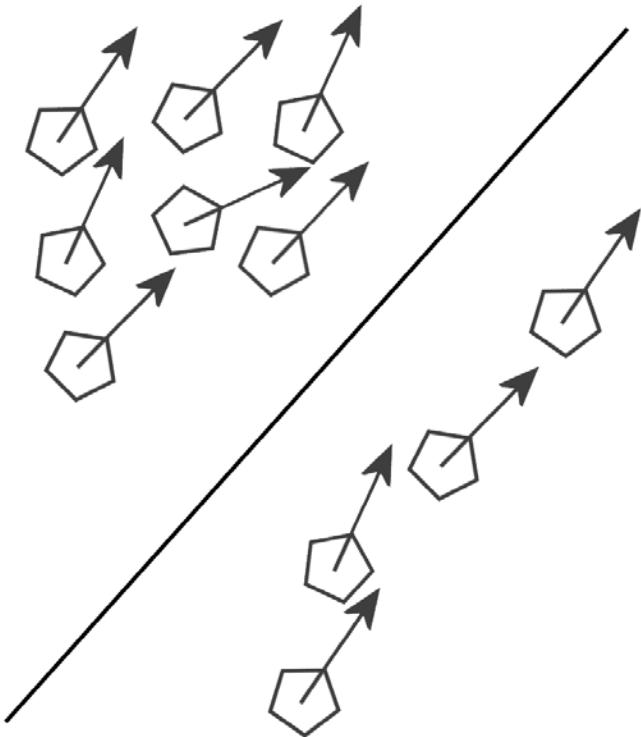


Abbildung 7. Formation des Flocks in Abhängigkeit vom Sichtwinkel θ . Ein großer Winkel begünstigt einen kompakten Flock (links oben), ein kleiner Winkel eine Kettenbildung (rechts unten).

5 Zusammenfassung

Die beiden Verfahren der Ant Colony Optimization und Particle Swarm Optimization als Beispiele für Schwarmintelligenz-Algorithmen wurden inklusive möglicher bzw. notwendiger Anpassungen für deren Einsatz im Bereich der Computerspiele vorgestellt. Bisher werden beide Verfahren jedoch, im Gegensatz zu anderen Methoden der Computational Intelligence, in diesem Gebiet kaum eingesetzt. Es wurden jedoch Einsatzmöglichkeiten im Bereich der Entwicklung (dynamischer) künstlicher Intelligenzen für Computergegner dargestellt, die von den Fähigkeiten der Schwarmintelligenzverfahren profitieren könnten. Schließlich wurde ein der Particle Swarm Optimization sehr ähnliches Verfahren zur Steuerung von Einheitengruppen, das Flocking, beschrieben.

Literatur

[Kenn01] James Kennedy; Russell C. Eberhart: Swarm Intelligence. Morgan Kaufmann, San Francisco (2001)

- [Rey87] Reynolds, C. W.: Flocks, Herds, And Schools: A Distributed Behavioral Model. *Computer Graphics* 21(4) (1987) 25–34
- [Dor06] Marco Dorigo, Mauro Birattari, Thomas Stützle: Ant Colony Optimization. *IEEE Computational Intelligence Magazine* (2006) 28–39
- [Champ03] Alex J. Champandard: AI Game Development. New Riders, Indianapolis (2003)
- [Bourg04] David M. Bourg, Glenn Seemann: AI For Game Developers. O'Reilly Media, Beijing (2004)
- [Buck05] Mat Buckland: Programming Game AI By Example. Wordware Publishing Inc., U.S. (2005)
- [Onwu04] Godfrey C. Onwubolu, B.V. Babu: New Optimization Techniques In Engineering. Springer, Berlin (2004) 219–238
- [ACO04] Marco Dorigo (Hrsg.): Ant Colony Optimization And Swarm Intelligence. Springer, Berlin (2004)
- [ACO06] Marco Dorigo (Hrsg.): Ant Colony Optimization And Swarm Intelligence. Springer, Berlin (2006)
- [Mue01] M. Müller: Computer Go. Special issue on games of Artificial Intelligence Journal (2001)
- [Rai04] Tapani Raiko: The Go-Playing Program Called Go81. FAIC, Helsinki (2004) 197–206,
- [Abr90] B. Abramson: Expected-outcome: A General Model Of Static Evaluation. *IEE Transactions on Pattern Analysis and Machine Intelligence* (1990) 182–193
- [Fritz95] Jürgen Fritz: Warum Computerspiele Faszinieren. Juventa (1995)
- [Pon04] Marc Ponsen, Pieter Spronck: Improving Adaptive Game AI With Evolutionary Learning Lehigh University, Bethlehem (2004)

KI in Ego Shootern

Computational Intelligence in Computerspielen

Jan Quadflieg

TU Dortmund

1 Einleitung

Das Genre der sogenannten *Ego-* oder auch *First-Person-Shooter* (FPS) ist mittlerweile 15 Jahre alt¹ und erfreut sich großer Beliebtheit². Zentraler und kontroverser Inhalt, aber nicht unbedingt Ziel dieser Actionspiele ist immer das Töten möglichst vieler Gegner, seien es vom Computer gesteuerte *Nicht-Spieler-Charaktere* (englisch Non-Player-Character, NPC) oder die Figuren anderer Spieler. Diese Seminararbeit soll einen Überblick darüber geben, welche Verfahren die Spieleentwickler für die Implementierung der künstlichen Intelligenz (KI) der Computergegner einsetzen und an welchen Stellen Methoden der Computational Intelligence (CI) Anwendung finden. Dabei wird auch immer wieder der Begriff des *Agenten* als Bezeichnung für die NPCs fallen:

Ein Agent ist ein in eine *Umgebung* eingebettetes Computersystem, welches fähig ist, *autonome* Handlungen innerhalb seiner Umgebung durchzuführen, um seine Zieltvorgabe zu erfüllen. [3]

Dabei ist die Umgebung natürlich die virtuelle Spielwelt. Eine andere geläufige Bezeichnung für die Computergegner, speziell im Kontext der *Multiplayer Shooter*, ist der Begriff des *Bot* (Abkürzung von englisch robot (Roboter)). Wie Paul Tozour feststellt, finden sich auch Elemente der Robotik in der Spiele-KI wieder, speziell die Wegfindung und Navigation [4].

Charakteristisch und namensgebend für FPS ist die Sichtweise des Spielers auf die Spielwelt: Er sieht die Umgebung aus der Perspektive des Charakters, den er verkörpert, wie in Abbildung 1 zu sehen. Diese Darstellung fand sich zuerst im Arcadespiel Battlezone³. Das Genre der FPS begründete die Firma id Software mit ihren Spielen Wolfenstein3D und Doom [1].

Seit der Veröffentlichung von Quake⁴ ist die Spielwelt von FPS durchgängig dreidimensional. Überhaupt stand die Verbesserung der Grafikqualität, angetrieben von immer leistungsfähigerer Hardware, jahrelang im Vordergrund der

¹ Gemessen an der Veröffentlichung von Wolfenstein3D am 5. Mai 1992 [1].

² Der Multi Player Shooter *Counterstrike* kommt auf ca. 13,3 Millionen Stunden Spielzeit pro Monat. *Half Life 2 - Episode One* kommt im Zeitraum Juni 2006 bis Oktober 2007 auf 3,8 Millionen Stunden Spielzeit [2].

Siehe auch c't 22/07 vom 15.10. 2007, Seite 80-83.

³ Atari, 1980

⁴ id Software, 1995



Abbildung 1. Typisches Bild eines First Person Shooters, hier aus dem Spiel Far Cry (Crytek, 2004). Gut zu sehen ist die Sichtweise aus der Perspektive der Figur, die der Spieler steuert. Unten rechts sieht man die Waffe, die der Spieler gerade benutzt. Außerdem sind unten rechts Informationen visualisiert, die in jedem FPS von entscheidender Bedeutung für den Spieler sind: Als roter Balken die noch verfügbaren Lebenspunkte (hier etwa die Hälfte) und rechts daneben die vorhandene Munition, für die aktuell gewählte Waffe.

Weiterentwicklung. Unsere menschliche Wahrnehmung ist sehr stark vom Sehen geprägt, schließlich ist das Auge unser primäres Sinnesorgan. Entsprechend leicht lassen sich Verbesserungen bei der Grafik in Screenshots zeigen und vermarkten, die schönere Darstellung fällt buchstäblich sofort ins Auge. Änderungen, die sich nicht direkt im optischen Erscheinungsbild niederschlagen werden dagegen oft übersehen und bleiben unbeachtet, nicht nur bei Computerspielen [5]. Ein anderer Grund ist die Tatsache, dass Techniken für die Verbesserung der Grafikqualität schon lange erforscht sind und nur aus dem Bereich des Offline Rendering übernommen werden müssen, während die künstliche Intelligenz ein Feld aktiver Forschung darstellt⁵.

Erst in den letzten Jahren rückte auch die Verbesserung der Computergegner in den Fokus der Spieleentwickler. Die durch die Verlagerung der Grafikberechnung auf dedizierte Hardware frei gewordene Rechenleistung machte dies möglich und die Erwartung der Spieler machte dies nötig: Die durch das Internet geschaffene Möglichkeit, direkt gegen andere menschliche Spieler anzutreten, steigerte auch die Erwartungshaltung gegenüber der Intelligenz und dem Verhalten computergesteuerter Gegner (vgl. [4]). Bei der Verbesserung der KI gehen die Entwickler jedoch äußerst konservativ vor, gerade bei den FPS sind es überwiegend einfache endliche Automaten, die das Verhalten der Computergegner steuern. Die Spieleindustrie ist ein hoch riskantes Geschäft; ein Entwicklerteam besteht aus 20 oder mehr Personen, jede Verzögerung des Zeitplans kostet entsprechend viel Geld [6].

⁵ John Carmack, Keynote Speech QuakeCon 2005, ein Transkript findet sich unter <http://forum.beyond3d.com/showpost.php?p=543166>

Da die KI im Rahmen des Spiels mit einem Menschen interagiert, dessen Verhalten sich nicht vorhersehen lässt, kostet es viel Zeit eine KI zu entwickeln, die robust genug ist in allen Situationen ein glaubhaftes Verhalten zu zeigen [7], [8]. Die Entwickler greifen deshalb auf Methoden zurück, die sich deterministisch verhalten und einfach zu debuggen sind.

Ein anderer aktueller Trend ist die immer bessere Simulation physikalischer Gesetze für die Objekte in der Spielwelt.

First-Person-Shooter lassen sich in zwei Gruppen einteilen: *Single Player* Spiele, die von einem Spieler alleine gespielt werden und *Multi Player* Spieler, bei denen man gegen andere menschliche Spieler über lokale Netze oder über das Internet spielt. Häufig bieten FPS beide Modi an, reine Multi Player Shooter enthalten Bots, die ein Spielen alleine ermöglichen. Der einfachste Spielmodus eines Multi Player Shooters ist das *Deathmatch*, bei dem derjenige Spieler (oder beim *Team Deathmatch* das Team von Spielern) gewinnt, der am häufigsten andere Spieler getötet hat. Bei anderen Spielmodi ist das Töten der Gegner nur Mittel zum Zweck, es gewinnt, wer am häufigsten die gegnerische Flagge erobert hat (*Capture the Flag*, CTF) oder andere vorgegebene Ziele erfüllt.

Single Player Spieler enthalten eine Rahmenhandlung, die die Aufgaben, die der Spieler zu erfüllen hat, mehr oder weniger überzeugend motiviert. Dabei trifft der Spieler nicht mehr nur ausschließlich auf feindlich gesinnte NPCs.

1.1 Überblick

Abschnitt 2 stellt eine allgemeine von Paul Tozour [9] vorgeschlagene Architektur für einen Agenten eines FPS vor und geht detailliert auf die Aufgaben und den Aufbau der einzelnen Teilmodule wie Wegfindung und Verhaltenssteuerung ein. Abschnitt 3 geht auf konkrete Implementierungen ein und stellt die Computergegner der Spiele Quake 3 Arena, Far Cry, den Counterstrike Bot JoeBot und einen alternativen Bot für Quake 3 Arena vor. Abschnitt 4 stellt die Intelligenz der Computergegner dem Spielspaß gegenüber.

2 KI Architektur für FPS

Paul Tozour hat in [9] eine allgemeine Architektur für die KI eines FPS vorgestellt. Dabei teilt er das KI-Modul in vier Komponenten ein, wie in Abbildung 2 zu sehen ist: *Bewegung*, *Animation*, *Kampf* und *Verhaltenssteuerung*. Die einzelnen Komponenten werden mit Ausnahme des Animationsmoduls in den folgenden Abschnitten detailliert vorgestellt. Natürlich hat jedes Spiel andere Anforderungen an die KI der Computergegner; die vorgestellte Architektur ist jedoch allgemein genug gehalten, um als Grundgerüst in jedem FPS Einsatz zu finden. Auch in den in Abschnitt 3 vorgestellten Beispielen findet sich die hier vorgestellte Architektur wieder. Sie ist auch nicht allein auf den Einsatz in FPS festgelegt, ebenso gut ist ein Einsatz in Spielen aus der *third-person* Perspektive denkbar. Die hier vorgestellte Architektur modelliert jedoch nur das Verhalten eines einzelnen Agenten, während in letzter Zeit Gruppentaktiken und die koordinierte Zusammenarbeit mehrerer Agenten an Bedeutung gewinnt.



Abbildung 2. Grafische Darstellung der vorgestellten Beispielarchitektur. Der obere Block steuert das Verhalten des Agenten, wobei Paul Tozour das Kampfverhalten etwas abgrenzt. *Animation* und *Wegfindung* sind Dienstleistungsmodule, die vom Verhaltensmodul aufgerufen werden.

2.1 Bewegung

Die Komponente zur Bewegung des Agenten dient der Wegfindung und der Ausführung der Bewegung entlang des gefundenen Pfades, sie entscheidet jedoch nicht über das Ziel der Bewegung. Diese Ziele bekommt sie von anderen Komponenten und stellt dann nur sicher, dass der Agent das Ziel erreicht - bzw. meldet einen Fehler zurück, wenn kein Pfad zum angegebenen Ziel gefunden wurde.

Aus Gründen der Effizienz wird zwischen *globaler* und *lokaler* Wegfindung unterschieden. Die globale Wegfindung sucht einen Pfad zu einem vorgegebenen Ziel. Der dabei eingesetzte Standardalgorithmus A* (gesprochen „A Stern“) arbeitet auf einem Graphen, der keine Informationen über dynamische Objekte (andere NPCs, bewegliche Gegenstände) der Spielwelt enthält. Es ist deswegen Aufgabe der lokalen Wegfindung, bei der Navigation entlang des gefundenen Pfades eine Kollision mit dynamischen Objekten zu verhindern.

Wegfindung mit A* A* ist der Standardalgorithmus für die Wegfindung in Spielen. Er vereint ein gutes Laufzeitverhalten mit einem akzeptablen Speicherverbrauch für die benötigten Datenstrukturen und ist in der Lage, für zwei beliebige Punkte auf einer Karte den *billigsten* Pfad zwischen diesen Punkten zu finden, sofern ein Pfad existiert. *Billigster* Pfad deshalb, weil A* die Kosten aufsummiert, die bei jedem Teilschritt des Pfades entstehen. Was diese Kosten sind, kann abhängig von der jeweiligen Anwendung beliebig festgelegt werden, was A* sehr flexibel macht. Für die genauere Funktionsweise des Algorithmus sei an dieser Stelle auf [10] verwiesen. Trotz des guten Laufzeitverhaltens von A* lohnt es sich, seinen Einsatz zu vermeiden („The fastest code is code that doesn't run.“), z.B. indem vorher geprüft wird, ob die Sichtlinie zwischen Start- und Endpunkt frei ist. Ist das der Fall, gibt es einen direkten Weg zum Ziel und auf die Wegfindung mit A* kann verzichtet werden [11].

A* arbeitet auf einem Graphen⁶, dessen Knoten Positionen in der Spielwelt

⁶ Der zugrunde liegende Suchraum muß nicht unbedingt ein Graph sein, sich aber als solcher beschreiben lassen. Gerade in Strategiespielen bietet sich ein zweidimensio-

repräsentieren, auch Wegpunkte genannt. Sind zwei Knoten über eine Kante verbunden, besteht für einen Agenten die Möglichkeit von dem einen Wegpunkt zum Anderen zu kommen. Dieser Graph wird während der Entwicklung des Spiels erzeugt, entweder automatisch oder manuell. In der Praxis wird immer eine Mischform eingesetzt, bei der ein möglichst großer Teil des Graphen automatisch erzeugt wird, um Zeit zu sparen. Dieser Graph dient dann als Ausgangspunkt für weitere Verbesserungen durch den Level Designer, der die Möglichkeit hat, den Graphen nach Belieben zu verändern. Die Knoten enthalten Informationen darüber, welche Fläche in der Spielwelt sie repräsentieren. Die Kanten enthalten mindestens Informationen über die Kosten, die bei ihrer Traversierung entstehen, darüber hinaus häufig auch weitere Informationen, z.B. welche Art der Bewegung nötig ist (Gehen, Schwimmen) oder wie viel Platz der Pfad bietet (zwingt große Agenten zur geduckten Fortbewegung).

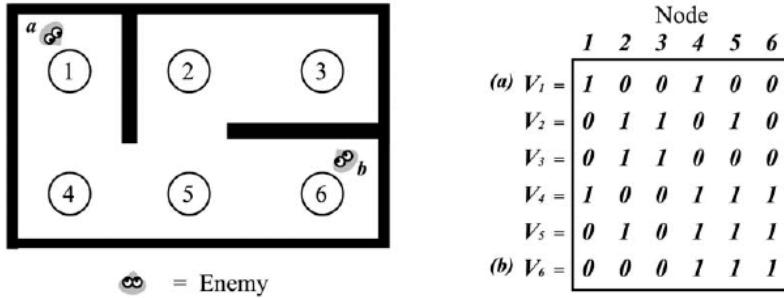


Abbildung 3. Annotation des Navigationsgraphen mit Sichtbarkeitsinformationen. Links ein kleines Level, dessen Navigationsgraph 6 Knoten enthält. Rechts die Bitvektoren der einzelnen Knoten. Grafiken aus [13].

Annotation des Graphen Der Graph kann jedoch noch mit weiteren Informationen versehen werden. So kann ein Level Designer zum Beispiel spezielle Knoten mit Hinweisen versehen, die zur Laufzeit von der KI ausgewertet werden können. Natürlich bedeutet dies mehr Arbeit für den Level Designer, jedoch bieten die zusätzlichen Informationen die Möglichkeit, auf einfache Weise die Planung taktischer Manöver zu unterstützen.

Lars Lidén schlägt in seinem Artikel *Using Nodes to Develop Strategies for Combat with Multiple Enemies* [13] vor, die Sichtbarkeit der Knoten untereinander in einem Preprocessing Schritt vorzuberechnen, um damit taktische Entschei-

nales Raster aus Rechtecken oder Sechsecken an. Einzelne Zellen des Rasters sind die Knoten des Graphen, die Kanten ergeben sich implizit aus den Nachbarschaftsbeziehungen der einzelnen Zellen. Für eine ausführliche Diskussion von möglichen Suchräumen siehe [12].

dungen des Agenten zu unterstützen. Der Vorteil ist, dass diese Informationen vollautomatisch berechnet werden können. Da die Knoten des Graphen Flächen repräsentieren, ist die Aussage, ob ein Knoten von einem anderen aus sichtbar ist, nicht sehr exakt (siehe Abbildung 4). Dieser Nachteil wird jedoch durch die schnelle Auswertung zur Laufzeit aufgewogen. Bei geschickter Verteilung der Wegpunkte lässt sich eine Situation, wie sie in Abbildung 4 gezeigt ist auch ganz vermeiden.

Für einen vorliegenden Navigationsgraphen mit n Knoten wird für jeden Knoten i ein Bitstring Λ_i der Länge n gespeichert. Ist das $j - te$ Bit im Bitstring Λ_a eines Knoten a gesetzt, ist der $j - te$ Knoten b von Knoten a aus zu sehen. Diese Darstellung ist sehr kompakt und Berechnungen zur Auswertung der Sichtbarkeitsinformationen beschränken sich auf einfache boolsche Operation wie *or*, *and* oder die Negation. Auf die gleiche Art wird für jeden Knoten i ein Bitstring Ψ_i vorberechnet, der angibt, mit welchen anderen Knoten er über eine Kante verbunden ist (ein solcher Bitstring ist also ein Zeilen- bzw. Spaltenvektor der Adjazenzmatrix des Graphen). Bereits diese sehr einfache Darstellung lässt die Planung vieler interessanter Manöver zu.

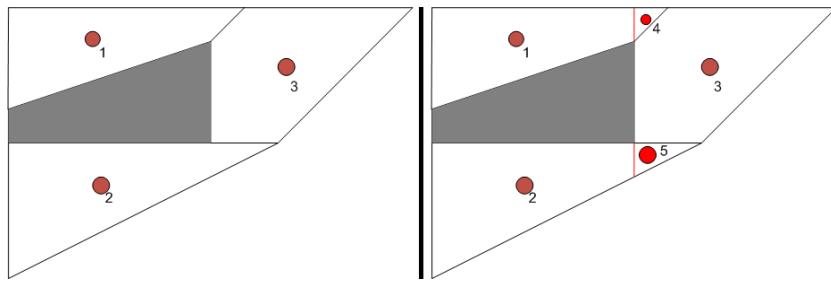


Abbildung 4. Ungenauigkeiten bei der Berechnung der Sichtbarkeit: Die graue Fläche repräsentiert ein Hinderniss, das die Sicht blockiert, außerdem sind drei Knoten des Navigationsgraphen und die dazu gehörenden Flächen zu sehen. Knoten 1 kann in der linken Grafik Knoten 2 sehen, obwohl eigentlich nur kleine Teile der jeweils anderen Fläche sichtbar sind und der Großteil hinter dem Hinderniss verborgen ist. Die rechte Grafik zeigt, wie diese Ungenauigkeit behoben werden kann, indem die Flächen geteilt und neue Knoten 4 und 5 eingefügt werden.

Sichere Positionen und Pfade Für jeden Knoten des Graphen lässt sich sehr einfach berechnen, ob er von der Position des Spielers aus zu sehen ist. Dafür muß nur der zur Position des Spielers gehörende Knoten e berechnet werden, dessen Bitstring Λ_e gibt direkt alle Knoten an, die von der Position des Spielers aus zu sehen sind. Kämpft ein Agent gegen mehrere Gegner (z.B. wenn der Spieler von verbündeten NPCs unterstützt wird), lässt sich der Bitstring V , der alle von den Gegnern sichtbaren Positionen enthält, durch eine einfache *or*-Verknüpfung der zu jedem Gegner j gehörenden Knoten V_j

berechnen:

$$V = \bigcup_{j=0}^{j=k} V_j.$$

Die Menge der sicheren Knoten ist dann die Negation von V .

Diese neu gewonnene Information lässt sich dafür nutzen, dass Agenten bei der Bewegung in Deckung bleiben: Als gefährlich identifizierten Knoten werden einfach sehr hohe Kosten zugeordnet, der A* Algorithmus bevorzugt dann bei der Wegfindung automatisch die sicheren Knoten.

Flankieren *Flankieren* beschreibt ein Manöver, bei dem der NPC versucht den Spieler zu umlaufen, möglichst ohne dabei gesehen zu werden, um ihn dann von hinten anzugreifen. Ausgangspunkt ist der Knoten F , bei dem sich der Spieler befindet. Dann werden im ersten Schritt die Knoten F_a bestimmt, von denen der Spieler gesehen werden kann (über den Bitstring von F), die er selber aufgrund seiner Orientierung aber nicht sieht. Dies lässt sich einfach mit Hilfe des Skalarprodukts des Vektors zwischen F und $f \in F_a$ und des Vektors, der die Blickrichtung des Spielers angibt, berechnen. Die verbliebene Menge von Knoten kommt als Ziel für das Manöver in Frage.

Sniper Positionen Als *Sniper Positionen* bezeichnet Lidén Positionen, die eine Sichtlinie zum Gegner bilden und in deren direkter Umgebung Punkte liegen, die Deckung bieten (zum Beispiel zum Nachladen der Waffe). Mögliche Schußpositionen entnimmt man wieder dem Bitstring Λ_e des Knoten e , an dem sich der Gegner befindet, wie beim Manöver „Flankieren“. Für jeden der gefundenen Knoten werden dann mit Hilfe dessen Bitstrings Ψ die Nachbarn bestimmt. Ist einer der Nachbarknoten für den Gegner nicht sichtbar, bietet der Nachbarknoten Deckung und der ursprüngliche Knoten bildet zusammen mit seinem Nachbarn eine *Sniper Position*.

In *Strategic and Tactical Reasoning with Waypoints* [14] stellt Lidén eine weitere Möglichkeiten vor, taktische Informationen aus dem Navigationsgraphen zu extrahieren. Die Idee beruht auf der Identifikation sogenannter *Pinch Points*, welche Engpässe in der Spielwelt markieren. An solchen Stellen ergeben sich dann für die KI Möglichkeiten überraschend aus dem Hinterhalt anzugreifen. Pinch Points lassen sich algorithmisch im Graphen identifizieren:

Für jeden Punkt N des Graphen, der nur zwei Nachbarknoten hat, tue das Folgende:

1. Entferne vorübergehend den Knoten N und speichere die Nachbarn als Knoten A und B .
2. Falls A und B mit großen Flächen verbunden sind, ist N kein Pinch Point. Wähle den nächsten Knoten N .
3. Versuche, einen Pfad zwischen A und B zu finden. Wenn ein Pfad existiert, ist N kein Pinch Point. Wähle den nächsten Knoten N .
4. Der Nachbarknoten, der mit der größeren Fläche verbunden ist, wird O genannt, der Nachbarknoten, der mit der kleineren Fläche verbunden ist, wird I genannt.

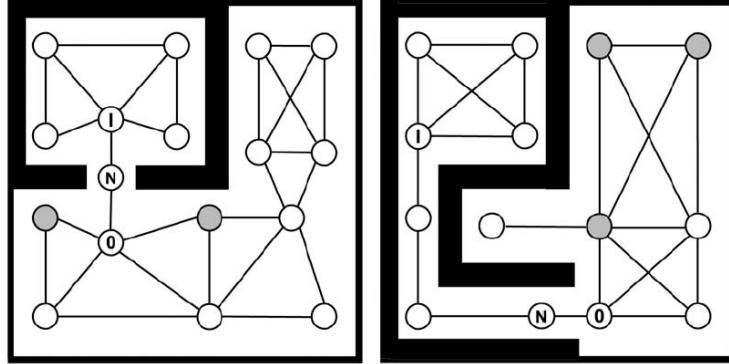


Abbildung 5. *Pinch Points* in Navigationsgraphen. N ist der Pinch Point, I der innere Knoten und O der äußere Knoten. Positionen, die für einen Überraschungsangriff in Frage kommen, sind grau unterlegt. Die rechte Grafik verdeutlicht die Handhabung langer Gänge bei der Berechnung von Pinch Points. Grafiken aus [14].

Mögliche Positionen A für einen Hinterhalt sind dann alle Knoten, die eine Sichtlinie zu O , nicht aber zu N haben. Das lässt sich leicht mit den Sichtbarkeitsinformationen berechnen. Sei V_O die Menge der Knoten, die von O aus sichtbar sind, und \bar{V}_N die Menge der Knoten, die von N aus nicht sichtbar sind, dann ist

$$A = V_O \cap \bar{V}_N.$$

Abbildung 5 zeigt in der linken Grafik die Lage des Pinch Points, des inneren und des äußeren Knoten und grau unterlegt die Positionen, die für einen Überraschungsangriff in Frage kommen. Die rechte Grafik zeigt eine Situation mit einem langen, schmalen Gang, für die der oben beschriebene Algorithmus erweitert werden muß, um die Lage der drei Punkte korrekt zu bestimmen. Dafür muß der oben beschriebene Algorithmus um folgenden Schritt ergänzt werden:

5. Solange O neben N nur einen anderen Nachbarn V hat
 - Setze $N = O$
 - Setze $O = V$

2.2 Verhaltenssteuerung und Kampf

Die Verhaltenssteuerung und das darin eingebettete Kampfmodul bestimmen das Verhalten des Agenten, d.h. seine Reaktion auf die Ereignisse in seiner Umgebung (der Spielwelt). Dabei setzen die Spieleentwickler überwiegend auf *endliche Automaten* (*Finite-State-Machines*, FSM)[9]. Die Aktionen des Computergegners in einem bestimmten Zustand sowie bei Zustandsübergängen sind dabei fest im Code des Spiels verankert. Die Computergegner in FPS verfügen mindestens über einen *Idle* Zustand, in dem sie nichts tun und einen Zustand für den Kampf gegen den Spieler. Dazu kommen andere Zustände wie das Suchen

nach dem Spieler, wenn der NPC diesen aus den Augen verloren hat. Zustandsänderungen werden über Ereignisse ausgelöst, die von der Game Engine berechnet werden. Auch die Wahrnehmung des NPC (Sehen, Hören) wird über solche Ereignisse modelliert.

Ergänzt wird die Verhaltenssteuerung durch *Skripte* und *Triggersysteme*. Skripte geben dem Designer die Möglichkeit, das Verhalten eines Agenten genau vorzugeben. *Trigger* sind unsichtbare Schalter in der Spielwelt, die unbemerkt vom Spieler aktiviert werden. Das können z.B. unsichtbare Flächen sein, die auf die Berührung durch den Spieler reagieren. Durch die Aktivierung des Schalters wird dann die Ausführung eines Skripts aktiviert.

Ein Beispiel soll dieses Zusammenspiel verdeutlichen: Der Spieler betritt einen Raum durch eine Tür, dabei durchläuft er die Fläche eines Schalters, die sich in der Türöffnung befindet. Dieser Schalter startet die Ausführung eines Skripts, das einen NPC veranlasst, den gleich Raum durch eine andere Tür zu betreten. Für den Spieler entsteht dann der Eindruck, der NPC (z.B. ein Wachmann) habe ihn gehört und sich intelligenterweise auf die Sache nach Spieler gemacht. In Wirklichkeit ist das scheinbar intelligente Verhalten strikt vom Spiel Designer vorgegeben. Das Beispiel verdeutlicht auch die zwei größten Nachteile von Skripten und Triggersystemen: Das Spielerlebnis ist immer das gleiche, jedes mal wenn der Spieler den Raum betritt, wird der NPC gleich reagieren (den Raum durch die andere Tür betreten). Beim wiederholten spielen sind die Computergegner leicht zu durchschauen. Darüber ist es schwierig sicherzustellen, dass der Spieler überhaupt das Skript auslöst (vielleicht findet er einen anderen Weg in den Raum?) oder dass Skripte in der richtigen Reihenfolge vom Spieler ausgelöst werden. Letzteres ist vor allem in Spielen mit großen Außenarealen, in denen sich der Spieler frei bewegen kann, problematisch.

3 Beispiele

In diesem Abschnitt werden die konkreten KI Implementierungen zweier kommerzieller First Person Shooter und zweier Bots vorgestellt. Das Spiel *Farcry* steht dabei für eine klassische FPS KI auf Basis einer FSM. Die anderen drei Beispiele setzen dagegen Methoden der Computational Intelligence ein: Die Bots von *Quake 3 Arena* benutzen Fuzzy Relationen zur Auswertung der Eingabe, die alternativen Quake 3 Arena Bots von Priesterjahn et al. basieren auf Regelbasen, die mit Hilfe eines evolutionären Algorithmus erzeugt werden und *Joebot* setzt neuronale Netze zur Entscheidungsfindung ein.

3.1 Quake 3 Arena

Quake 3 Arena wurde am 2. Dezember 1999 von id Software veröffentlicht und ist im Gegensatz zu den Vorgängern Quake⁷ und Quake 2⁸ als reiner Multiplayer Shooter konzipiert, d.h. Gegenstand des Spiels ist der Kampf gegen andre menschliche Spieler über lokale Netzwerke oder das Internet. Um dennoch das

⁷ id Software, 1996

⁸ id Software, 1997

Spiel alleine spielen zu können, verfügt Quake 3 Arena über Bots, die sich ähnlich wie ein menschlicher Spieler verhalten und als Trainingspartner dienen. Die Entwicklung der Bots wurde lange Zeit vom Entwicklungsteam vernachlässigt. Als erste eigene Versuche nicht die gewünschten Resultate brachten, wurde diese Aufgabe im März 1999 an den Niederländer Jan Paul van Waveren abgegeben, der sich bereits zuvor in der Mod⁹ Szene mit Bots für Quake und Quake 2 einen Namen gemacht hatte [1]. Van Waveren beschreibt die Funktionsweise der Bots ausführlich in seiner Masterarbeit [15]. In der Zwischenzeit wurde auch der Quelltext von Quake 3 Arena veröffentlicht¹⁰.

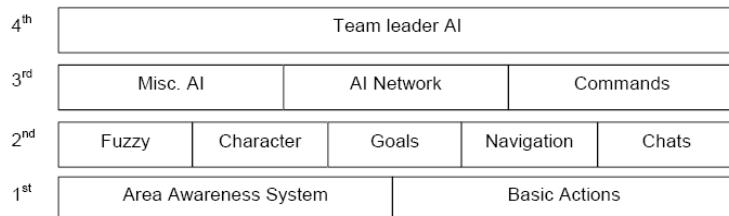


Abbildung 6. Die vier Schichten der Quake 3 Arena Bot KI. Grafik aus [15]

Aufbau der Bot KI Die KI eines Quake 3 Arena Bots besteht aus vier Schichten, wie in Abbildung 6 zu sehen. Auf der höchsten Ebene befindet sich die Team Leader KI, die nur dann zum Einsatz kommt, wenn der Bot Teil eines Teams ist und dieses anführt. Unter dieser Schicht liegt die Verhaltenssteuerung des einzelnen Bots, das *AI Network*, eine FSM. Außerdem befinden sich in dieser Schicht das *Command Module*, das ggf. Kommandos anderer Spieler oder des Teamführers verarbeitet und ein *misc. AI* genannter Teil, der Verhaltensmuster enthält, die vom AI Network verwendet werden. Die unterste Schicht bildet die Ein- und Ausgabe des Bots. Über das *Area-Awareness-System* nimmt der Bot seine Umgebung wahr. Das Ausgabemodul *Basic Actions* ahmt über elementare Aktionen wie *Attack*, *Move*, *Jump* und andere eine Schnittstelle nach, wie sie auch ein menschlicher Spieler über die Eingabemöglichkeiten von Maus und Tastatur hat. Zwischen der Ein-/Ausgabeschicht und der Verhaltenssteuerung liegt eine Schicht, die Eingaben fuzzyfiziert und Ziele (siehe unten) festlegt. Ob diese Ziele wirklich verfolgt werden, wird in der dritten Schicht entschieden. Die KI arbeitet mit einer Frequenz von 10Hz, wird also 10 mal pro Sekunde ausgeführt, und der Bot kann die Eingaben neu auswerten und ggf. sein Ziel neu festlegen. Da das Ausführen der Bot KI die Spielsimulation unterbricht, darf die benötigte Rechenzeit nur wenige Millisekunden betragen.

⁹ Abkürzung für Modifikation. Modifikationen sind von Fans erstellte Erweiterungen eines Spiels, die zum Beispiel neue Level, Waffen und Gegner enthalten.

¹⁰ <ftp://ftp.idsoftware.com/idstuff/source/quake3-1.32b-source.zip>

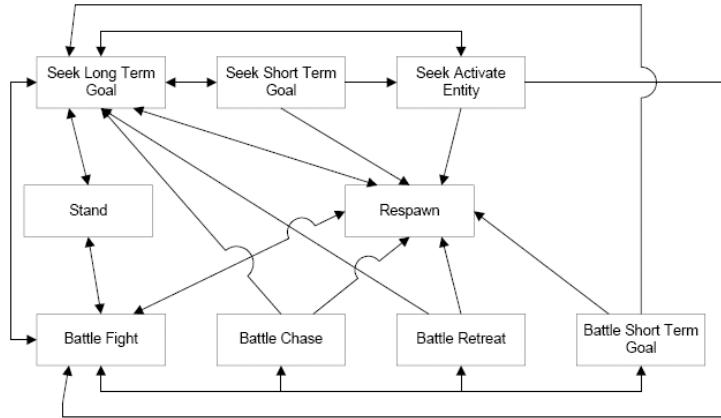


Abbildung 7. Das AI Network der Quake 3 Bots. Zu sehen sind Knoten, die die Zustände repräsentieren, in denen sich ein Bot befinden kann. Die Kanten visualisieren die möglichen Zustandsübergänge. Es fehlen die Zustände *Intermission* und *Observer*. Grafik aus [15]

AI-Network Das *AI-Network* bestimmt das Verhalten des Bots. Es besteht aus Knoten, die Zustände repräsentieren und Übergängen zwischen diesen Knoten. Ein Bot befindet sich immer in genau einem Zustand. Das Netzwerk ist in Abbildung 7 zu sehen. Es fehlen zwei Zustände, in denen der Bot jedoch nicht aktiv am Spiel teilnimmt: *Intermission* (beim Wechsel des aktuellen Level) und *Observer*.

Ein Bot hat immer das Ziel das Spiel zu gewinnen. Je nach Spielmodus bedeutet das, möglichst viele Gegner zu töten (Deathmatch und Team Deathmatch) oder möglichst oft die gegnerische Flagge zu erbeuten (CTF).

CI Methoden der Q3A Bots In der zweiten Schicht der Bot KI werden Fuzzy Relationen eingesetzt. Diese dienen dazu festzulegen, wie sehr ein Bot einen Gegenstand haben, ein Ziel verfolgen oder einen Gegenstand einsetzen möchte. Gespeichert werden sie in einer baumartigen Datenstruktur.

Die Fuzzy Relationen der Bots wurden mit Hilfe eines EA erzeugt. Dazu kämpfen 10 Bots mit anfangs identischen Relationen in Duellen gegeneinander. Die Fitness eines Bots ergibt sich direkt aus der Anzahl gewonnener Duelle, die Wahl der Eltern für die Rekombination erfolgt proportional zu den Fitnesswerten. Bei der Rekombination werden die Fuzzy Zugehörigkeitswerte in den Baumstrukturen der beiden Eltern gemittelt¹¹. Danach werden die Werte noch durch die Addition einer Zufallszahl mutiert. Leider liefert die Beschreibung des EA in van Waverens Masterarbeit keine weiteren Details, wie konkrete Wahrscheinlichkeiten für die Mutation, die Selektion der Folgepopulation, etc.

¹¹ Das hat natürlich bei der ersten Iteration, wenn beide Eltern noch identische Werte haben keinen Effekt

3.2 Alternative Bots für Quake 3 Arena

Steffen Priesterjahn et al haben einen alternativen Bot für Quake 3 Arena entwickelt. Grundlage ihres Bots ist eine Regelbasis, die mit Hilfe eines evolutionären Algorithmus (EA) erzeugt wird [16]. Ziel war es jedoch nur, eine Regelbasis für das Kampfverhalten zu erzeugen.

Regeln Eine *Regel* $R : G \rightarrow C$ bildet ein Raster G auf einen Befehl C ab. Das Raster G ist die Wahrnehmung der Umgebung des Bots. Dabei wird ein 15m*15m großes Quadrat, in dessen Mitte sich der Bot befindet, auf eine Matrix $G = (g_{i,j})_{1 \leq i,j \leq n} \in \mathbb{N}_0^{n \times n}$, $n \in \mathbb{N}$, $n \equiv 1 \text{ mod } 2$, $g_{i,j} \in \{0, 1, 20\}$ abgebildet. Dabei bedeutet eine 0, dass das Feld $g_{i,j}$ nicht passierbar ist, eine 1, dass das Feld leer ist und eine 20, dass das Feld von einem Gegner belegt ist. Ein solches Raster repräsentiert also eine bestimmte Spielsituation aus der Sicht des Bots.

Ein Befehl C ist ein Tupel $C = (f, r, \varphi, a)$ mit $f, r \in \{-1, 0, 1\}$, $a \in \{0, 1\}$ und $\varphi \in [-180^\circ, 180^\circ]$. f ist die Bewegung geradeaus ($0 = \text{keine Bewegung}$, $-1 = \text{Bewegung nach hinten}$, $1 = \text{Bewegung nach vorne}$), r die Bewegung zur Seite ($0 = \text{keine Bewegung}$, $-1 = \text{Bewegung nach links}$, $1 = \text{Bewegung nach rechts}$), a bestimmt, ob der Bot angreift ($0 = \text{kein Angriff}$, $1 = \text{Angriff}$) und φ bestimmt die Änderung des horizontalen Blickwinkels.

Für eine bestimmte als Raster / Matrix G kodierte Spielsituation legt eine Regel R also fest, wie der Bot in dieser Situation reagieren soll. Dabei steht dem Bot jedoch nicht die gleiche Menge an Informationen zur Verfügung, wie den oben beschriebenen Quake 3 Arena Bots. Das Raster kann z.B. nicht ausdrücken, wo sich bestimmte Gegenstände befinden.

Regelbasis und Auswertung der Regeln Die Regelbasis des Bot ist eine Teilmenge aller möglichen Regeln. Priesterjahn et al haben mit verschiedenen großen Regelbasen mit unterschiedlichen Rastergrößen experimentiert. Dabei hat sich eine Anzahl von 100 Regeln als 'optimal' herausgestellt, eine kleinere Regelbasis führte zu einem wesentlich schlechteren Verhalten des Bots, gemessen am Schaden, den er seinem Gegner zugefügt hat. Das gleiche gilt für eine deutlich größere Regelbasis mit 400 Regeln. Die Größe des Rasters hat einen ähnlichen Einfluß auf die Stärke des Bots: Ein 15×15 Felder großes Raster brachte die besten Ergebnisse, bei kleineren und größeren Rastergrößen nahm die Spielstärke des Bots ab.

Bei der Ausführung der Bot KI wird für die aktuelle Spielsituation das Raster G erzeugt, das diese repräsentiert. Das erzeugte Raster wird dann mit allen Rastern in der Regelbasis verglichen und die Regel ausgeführt, deren Raster G am ähnlichsten ist. Als Maß für die Ähnlichkeit dient der euklidische Abstand, beide Raster werden vorher mit einem Gauss Filter geglättet.

Erzeugung der Regelbasis Die Bots und ihre Regelbasen wurden mit einem $(\mu + \lambda)$ EA erzeugt, dessen Module im Folgenden beschrieben werden. Die Anzahl der Individuen μ betrug 10, die Anzahl der Nachkommen λ betrug 50. Der EA wurde nach 72 Generationen abgebrochen.

Rekombination Für die Rekombination werden zwei Eltern zufällig gleichverteilt gewählt. Die Regelbasis des Nachkommen wird dann aus den Regelbasen der Eltern zusammengesetzt, wobei die i -te Regel des Nachkommen zufällig gleichverteilt aus den i -ten Regeln der Eltern gewählt wird. Die Rekombination ändert also noch nichts an einzelnen Regeln sondern stellt nur aus den Regeln der Eltern eine neue Regelbasis zusammen.

Mutation Die Mutation verändert einzelne Regeln des Nachkommen, wobei sowohl das Raster G als auch der Befehl C einer Regel der Mutation unterliegen. Alle Änderungen werden mit der gleichen Wahrscheinlichkeit $p_m = 0,1$ vorgenommen. Im Raster G kann ein leeres Feld zu einem vollen Feld mutieren (und umgekehrt) oder die Position des Gegners kann von einem Feld zu einem benachbarten Feld verschoben werden.

In einem Befehl können f , r und a durch eine Mutation auf einen der möglichen Werte gesetzt werden. Der Wert des Blickwinkels φ wird bei einer Mutation durch die Addition eines zufälligen Winkels $\delta \in [-5^\circ, 5^\circ]$ verändert.

Fitnessfunktion Für die Bewertung der Fitness spielt der Bot 1 Minute gegen einen der in Quake 3 Arena enthaltenen Bots. Die dabei verlorenen Lebenspunkte des Individuums h_{own} , multipliziert mit einem Faktor w_{own} , werden von den verlorenen Lebenspunkten des Gegners h_{opp} , wiederum multipliziert mit einem Faktor w_{opp} , abgezogen. Damit ergibt sich die Fitnessfunktion $f = w_{opp}h_{opp} - w_{own}h_{own}$.

Die Gewichtsfaktoren wurden eingeführt, weil die einfache Fitnessfunktion $f = h_{opp} - h_{own}$ (entspricht $w_{own} = 1$, $w_{opp} = 1$) dazu führte, dass die erzeugten Bots einfach vor ihrem Gegner weg rannten, statt diesen anzugreifen. Mit den Gewichten $w_{own} = 1$ und $w_{opp} = 2$ griffen die Bots dagegen an, ohne an die eigenen Verluste zu denken. Die endgültig eingesetzte Fitnessfunktion verwendet deshalb $w_{own} = 1$ und startet mit $w_{opp} = 2$, senkt w_{opp} jedoch durch die Multiplikation mit einem Regressionsfaktor $q = 0.98$ bei jedem Generationsschritt, bis w_{opp} den Wert 1 erreicht.

Tests haben gezeigt, dass die erzeugten Bots in der Lage sind, die Quake 3 Arena Bots zu dominieren. Ihre Regelbasis deckt jedoch nur das Verhalten im Nahkampf ab, während die Quake 3 Bots langfristige Ziele verfolgen können.

3.3 Farcry

Farcry wurde im März 2004 von Crytek veröffentlicht und setzte mit seinen Computergegnern neue Maßstäbe. Die Gegner sind in der Lage, in den großen Außenarealen des Spiels zu navigieren, den Spieler koordiniert in Gruppen anzugreifen und ihn dabei zu umlaufen und einzukreisen. Ihre Implementierung ist nicht so gut dokumentiert wie die der Quake 3 Arena Bots, das Crytek AI Manual [17] liefert jedoch Hinweise auf ihre Funktionsweise.

Navigation Die Agenten von Farcry navigieren mit Hilfe eines ungerichteten Graphen, siehe auch Abschnitt 2.1. Der Graph kann weitestgehend (zu ca. 90%)

automatisch im Level Editor erzeugt werden; der Level Designer hat dann die Möglichkeit den erzeugten Graphen zu verändern. Die automatische Erzeugung des Graphen basiert auf der Triangulierung des Levels. Ein Knoten des Graphen repräsentiert eine dreieckige Fläche, die frei von Hindernissen ist. Für die Wegfindung wird der A* Algorithmus eingesetzt. Der gefundene Pfad wird jedoch noch begradiert, um ihn realistischer aussehen zu lassen. Das dabei eingesetzte Verfahren wird nicht beschrieben, eine mögliche Lösung findet sich in [18].

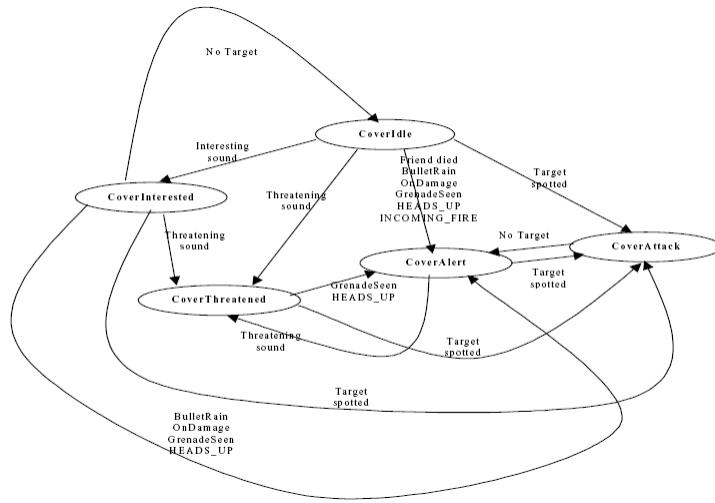


Abbildung 8. Finite State Machine eines Gegners aus Far Cry, hier für den Typ *Cover*. An den Kanten finden sich die *Tactical Events*, die den jeweiligen Zustandsübergang auslösen können. Grafik aus [17].

Verhaltenssteuerung Für die Verhaltenssteuerung der Agenten in Far Cry werden Finite State Machines eingesetzt, Abbildung 8 zeigt die FSM des Gegnertyps *Cover*. Übergänge zwischen den verschiedenen Zuständen können nur von sogenannten *Tactical Events* ausgelöst werden. Die Game Engine berechnet jedoch bei jedem Simulationsschritt *Game Events* genannte Ereignisse. Jeder Gegnertyp in Far Cry besitzt ein Regelwerk das angibt, welches Game Event welches Tactical Event auslöst. Game Events sind alle Ereignisse, die in der Spielwelt auftreten können (Geräusche, optische Wahrnehmung, etc). Die Reaktion auf einen Zustandsübergang ist fest im Code des Spiels integriert, alle Agenten eines Typs zeigen deshalb bei dem gleichen Zustandsübergang die gleiche Reaktion. Auch in Far Cry kommen darüber hinaus die in Abschnitt 2.2 beschriebenen Trigger- und Scriptingsystem in Form der Sprache LUA zum Einsatz.

Die Gegner können in Gruppen agieren, auch dieses Verhalten wird über die FSM und Tactical Events modelliert: Ein Agent kann Mitteilungen seiner Kameraden

in Form von Tactical Events bekommen oder selber solche Events erzeugen. Welche Events er an die anderen Mitglieder der Gruppe weitergibt, ist wieder fest als Reaktion auf einen Zustandsübergang vorgegeben.

3.4 JoeBot

JoeBot ist ein Bot von Lampel ([19]) für die beliebte Half Life Modifikation *Counterstrike*¹² (*CS*), der künstliche Neuronale Netze (KNN) für die Kollisionsvermeidung und für die Bewegungssteuerung in Kampfsituationen einsetzt. Beide Netze wurden offline mit dem Backpropagation Verfahren trainiert.

Kollisionsvermeidung Das KNN für die Kollisionsvermeidung ist ein ebenenweise verbundenes Feedforward Netz mit drei Eingabeneuronen, drei Neuronen in der verdeckten Schicht und einem Ausgabeneuron. Die Eingabe des Netzes besteht aus den Daten dreier Sensoren, die den Abstand zum nächsten Hinderniss angeben. Einer der Sensoren blickt direkt nach vorn, die beiden anderen Sensoren blicken um 35° nach links, bzw. nach rechts. Die Sichtweise ist auf eine Distanz von ca. 2m begrenzt. Die Konsequenz davon ist, dass das Netz nur in engen Korridoren zum Einsatz kommt, nicht aber im offenen Gelände. Das Ausgabeneuron liefert die nötige Richtungsänderung, um eine Kollision zu vermeiden: Drehung nach links, bei einem Ausgabewert kleiner -0.5, Drehung nach rechts bei einem Wert größer 0.5, sonst keine Reaktion.

Bewegung in Kampfsituationen Auch das KNN für die Bewegung ist ein ebenenweise verbundenes Feedforward Netz. Es besteht aus sechs Eingabeneuronen, zwei verdeckten Schichten mit je sechs Neuronen und fünf Neuronen in der Ausgabeschicht. Die sechs Eingaben sind: Anzahl der Lebenspunkte des Bots, Entfernung des Gegners, Kampfstärke der gegnerischen Waffe, Stärke der eigenen Waffe, verfügbare Munition und die aktuelle Situation¹³. Die Ausgabeneuronen entsprechen den Bewegungsmöglichkeiten des Bots: Springen, Ducken, Verstecken, Links / Rechts und Gehen / Laufen.

4 KI vs. Spaß

Es ist scheinbar offensichtlich, dass Computergegner *intelligent* sein sollen. Das muss aber nicht heißen, dass die Computergegner unbesiegbar sein sollen, schließlich gibt ein Spieler heutzutage ca. 40€ für ein aktuelles Computerspiel aus und

¹² Die Entwickler von Counterstrike wurden mittlerweile von Valve Software eingestellt. Neuere Versionen von Counterstrike sind als eigenständige Spiele erschienen und nicht mehr kostenlos.

¹³ Die Anzahl der Gegner, Anzahl der Mitspieler im eigenen Team und die Tatsache, ob der Bot eher defensiv oder offensiv aggieren soll. Wie diese Werte verrechnet werden, um die Eingabe für das Neuron zu erzeugen ist leider nicht beschrieben.

erwartet dafür, gut unterhalten zu werden [4]. Es ist also weniger ein besonders intelligentes Verhalten der Agenten gefragt, sondern ein überzeugendes, das die Glaubwürdigkeit der Spielwelt erhält und gleichzeitig eine zu bewältigende Herausforderung darstellt.

Auch Lidén stellt fest, dass zu intelligente Gegner im Sinne des Spielspaßes kontraproduktiv sein können, schließlich soll der Spieler immer in der Lage sein, das Spiel zu gewinnen. Er hat deshalb zehn Verhaltensmuster für eine FPS KI veröffentlicht, die teilweise darauf beruhen, die KI bewusst Fehler machen zu lassen, um den Spielspaß zu erhöhen [20]:

Move before firing: Dies bezieht sich auf die Situation, dass ein Spieler eine neue Umgebung, zum Beispiel einen Raum, betritt und sofort von einem Computergegner gesehen und angegriffen wird. Das kann äußerst frustrierend sein, vor allem, wenn der Spieler dabei sein (virtuelles) Leben verliert. Ein möglicher Ausweg ist, den NPC niemals sofort angreifen zu lassen, wenn er den Spieler wahrnimmt. Stattdessen sollte der Gegner zuerst einer anderen Tätigkeit nachgehen, zum Beispiel dem Suchen einer Deckungsmöglichkeit oder dem Scharfmachen seiner Waffe. Das gibt dem Spieler die Gelegenheit, auf die Bedrohung zu reagieren.

Miss the first time: Ein anderer Ausweg aus der unter "move before firing" beschriebenen Situation: Der Gegner greift sofort an, schießt aber absichtlich erst einmal daneben. Auch das gibt dem Spieler die Gelegenheit zu reagieren, bevor er Schaden nimmt. Der Einsatz dieses Verhaltensmusters bietet sich auch in anderen Situationen an, z.B. wenn ein Gegner von hinten angreift. Das Verhalten lässt sich weiter ausbauen, indem man den Gegner nicht daneben schießen lässt, sondern bewusst ein Objekt in der Spielwelt zerstören lässt. Dieses sollte sich in der Nähe des Spielers befinden und auf möglichst spektakuläre Weise kaputt geht, zum Beispiel eine Vase, die in 1000 Scherben zerspringt. Wenn es das Spieldesign zulässt, kann man Waffen mit Hinweisen versehen, die auf die Position des Schützen schließen lassen, zum Beispiel der Strahl eines Laserpointers (siehe Abbildung 9, links) oder Projektils, die Spuren in der Luft hinterlassen (Leuchtspurmunition oder auch die Luftverwirbelung der Railgun in Quake 3 Arena).

Be Visible: Im Gegensatz zur Realität, in der man die eigene Sichtbarkeit bei Kampfhandlungen möglichst vermeiden möchte, macht es in Spielen wenig Spaß, den Bildschirm Pixel für Pixel abzusuchen, um einen Gegner zu finden. Die Texturen der Gegner sollten also einen möglichst großen Kontrast zum Hintergrund haben.

Dieses Prinzip wird zum Beispiel von Team Fortress 2¹⁴ angewendet, bei dem sich die beiden Teams farblich deutlich unterscheiden (rot und blau) und von der Umgebung abheben. Außerdem sind die 3D-Modelle der Spieler so gestaltet, dass schon aus den Konturen hervorgeht, von welchem Typ

¹⁴ Valve Software, Veröffentlichung geplant für den 12. Oktober 2007



Abbildung 9. Links: Visuelle Hinweise, um die Position eines Heckenschützen zu erkennen. Hier ist es der Strahl eines Laserpointers, der den Gegner verrät. Screenshot aus Half Life 2 - Episode 2 (Valve Software, 2007). Rechts: Die weißen Tarnanzüge der Soldaten in Half Life.

ein anderer Spieler ist [21]. Im Spiel Half-Life¹⁵ tragen die Soldaten zwar Tarnkleidung, diese ist jedoch weiß (siehe Abbildung 9, rechts):

Wearing a clever form of anti-camo, these soldiers were designed to give both the impression of being camouflaged, but at the same time be highly visible on low-resolution displays.

(Chuck Jones in [22])

Have horrible aim: Eine weitere Möglichkeit, die Spannung zu erhöhen, ist es, den Spieler ständig unter Beschuß zu nehmen. Das dabei entstehende Risiko, dass der Spieler schnell Schaden nimmt oder stirbt, lässt sich reduzieren, indem die Computergegner sehr schlecht zielen. FPS setzen für die Schüsse der NPCs häufig Streuwinkel von bis zu 40° ein.

Warn the player: Eine weitere Möglichkeit, dem Spieler Gelegenheit zu geben, auf einen Angriff zu reagieren, ist es, ihn vor dem Angriff zu warnen, zum Beispiel durch das Abspielen einer passenden Animation (Scharfmachen der Waffe, o.ä.) oder durch eine akustische Äußerung des NPC ("Nimm das!").

Attack "Kung-Fu" Style: "Kung-Fu" im Sinne eines Bruce Lee Films: Der Spieler wird von mehreren Gegnern bedroht, ggf. sogar eingekesselt, von diesen greifen aber nur eine begrenzte Anzahl an, während der Rest anderen Tätigkeiten nachgeht. Das vermittelt dem Spieler das Gefühl, trotz einer großen Übermacht von intelligenten Gegnern bestehen zu können. Die Soldaten in Half Life bilden Gruppen, von einer Gruppe dürfen aber nur zwei Soldaten gleichzeitig angreifen. Die anderen Soldaten laden dann ihre Waffe nach oder ändern ihre Position.

¹⁵ Valve Software, 1998

Tell the player what you are doing: Es kann passieren, dass das clevere Verhalten von Computergegnern (Suchen von Deckung, Flankieren des Spielers) gar nicht wahrgenommen oder falsch verstanden wird ("Was rennt der NPC denn da so blöd durch die Gegend?"). Auch hier bieten akustische Äußerungen der NPCs einen Ausweg ("Gib mir Deckung!"). Ein angenehmer Nebeneffekt dieser Technik ist, dass ein Spieler den Computergegnern auch dann ein intelligentes Verhalten attestiert, wenn dieses womöglich gar nicht existiert.

React to mistakes: Es ist unvermeidbar, dass einer KI im Rahmen einer dynamischen Spielwelt Fehler unterlaufen, zum Beispiel beim Werfen einer Granate, die von einem dynamischen Objekt abprallt und zum Computergegner zurückrollt. Lässt sich dieser einfach in die Luft sprengen, ist der Fehler für den menschlichen Spieler offensichtlich und die KI wirkt dumm. Ist die KI jedoch in der Lage auf ihren Fehler zu reagieren, zum Beispiel durch das Einnehmen einer schützenden Haltung oder einem passenden Ausspruch ("Oh nein!"), bleibt ihre Glaubwürdigkeit erhalten.

Pull back at the last minute: Wie schon mehrfach erwähnt, ist es der Schlüssel zu einem spannenden Spielerlebnis, den Spieler permanent an seine Grenzen zu bringen und ihn gewinnen zu lassen. Dazu kann die KI den Zustand des Spielers überwachen und den Schwierigkeitsgrad senken (geringere Genauigkeit der abgefeuerten Schüsse), wenn die Spielerfigur des Spielers kurz davor ist, ihre Lebenspunkte vollständig zu verlieren. Dieses Verhalten darf jedoch nicht zu offensichtlich sein, durchschaut es der Spieler, geht der Reiz des Spiels verloren.

Intentional Vulnerabilities: Schwächen lassen Computergegner natürlicher wirken. Das können zum einen Nachteile sein, wie zum Beispiel eine kurze Pause, die ein Computergegner nach einem Sprint einlegen muß, bevor er den Spieler wieder angreifen kann. Die NPCs können jedoch auch absichtlich gelegentlich Fehler machen.

Es ist außerdem wichtig, dass die KI nicht offensichtlich betrügt ("cheatet", von englisch "to cheat"). Für einen KI Programmierer, der Zugriff auf alle internen Datenstrukturen eines Spiels hat, ist es einfach, die Computergegner allwissend zu machen. Spieler erkennen solche Tricks jedoch sehr schnell und fühlen sich betrogen. Selbst wenn Spieler nicht genau feststellen können, wie die KI betrügt, nehmen sie doch das unnatürliche Verhalten der KI wahr.

5 Ausblick

Bisher dominieren einfache deterministische Verfahren wie endliche Automaten bei der Verhaltenssteuerung von Computergegnern in FPS. Grund dafür ist die enorme Komplexität heutiger Spiele und die damit verbundenen Kosten ihrer Entwicklung, weswegen die Entwickler wenig Risikobereitschaft zeigen, andere

Verfahren auszuprobieren.

Wie jedoch die Beispiele in Abschnitt 3 zeigen, lassen sich verschiedenste Methoden der Computational Intelligence erfolgreich für die Entwicklung von Computergegnern für FPS einsetzen. Neue Ideen kommen dabei oft aus der Mod Szene (JoeBot, in gewisser Weise auch Quake 3 Arena), in der Fans der Spiele in ihrer Freizeit Erweiterungen für ihre Lieblingsspiele schaffen, einschließlich neuer Computergegner. Dabei unterliegen die Mod Entwickler nicht den Zeit- und Budgetbegrenzungen der professionellen Spieleentwickler und können so leichter neue Ideen ausprobieren.

Bei aller Finesse der Computergegner darf aber nie der Spielspaß zu kurz kommen. Die in Abschnitt 4 vorgestellten Möglichkeiten bieten interessante Verhaltensmuster, die eine Herausforderung für den Spieler darstellen, ihm aber dennoch erlauben, diese zu meistern.

Literatur

- [1] Kushner, D.: Masters of Doom: How Two Guys Created an Empire and Transformed Pop Culture. Random House (2003)
- [2] Valve: Statistiken der onlinevertriebsplattform steam von valve software. Online Article (09 2007) <http://www.steampowered.com/v/index.php?area=stats>.
- [3] Wooldridge, M.: Introduction to Multiagent Systems. John Wiley and Sons (2002)
- [4] Tozour, P.: 1.1 The Evolution of Game AI. In: AI Game Programming Wisdom. Volume 1. Charles River Media (2002) 3–14
- [5] Chen, R.: 14 When you change the insides, nobody notices. In: The old new thing. Addison Wesley (2006) 336–337
- [6] Sterret, J.: Reasons for the fall: A post-mortem on looking glass studios. Online Article (05 2000) <http://www.ttlg.com/articles/lgsclosing1.asp>.
- [7] B. Pelletier, M. Gummelt, J.M.: 4.2 Raven Software's Star Trek: Voyager Elite Force. In: Postmortems from Game Developer. CMP Books (2003) 237–250
- [8] E. Biesman, R.J.: 4.3 Raven Software's Soldier of Fortune. In: Postmortems from Game Developer. CMP Books (2003) 259–271
- [9] Tozour, P.: 8.1 First-Person Shooter AI Architecture. In: AI Game Programming Wisdom. Volume 1. Charles River Media (2002) 387–396
- [10] Matthews, J.: 3.1 Basic A* Pathfinding Made Simple. In: AI Game Programming Wisdom. Volume 1. Charles River Media (2002) 105–113
- [11] Rabin, S.: 3.5 A* Speed Optimizations. In: Game Programming Gems. Volume 1. Charles River Media (2000) 272–287
- [12] Tozour, P.: 2.1 Search Space Representations. In: AI Game Programming Wisdom 2. Volume 1. Charles River Media (2004) 85–102
- [13] Lidén, L.: Using nodes to develop strategies for combat with multiple enemies. Artificial Intelligence and Interactive Entertainment: Papers from the 2001 AAAI Symposium (2001)
- [14] Lidén, L.: 5.1 Strategic and Tactical Reasoning with Waypoints. In: AI Game Programming Wisdom. Volume 1. Charles River Media (2002) 211–220
- [15] van Waveren, J.: The quake iii arena bot. Master's thesis, University of Technology Delft (2001)
- [16] S. Priesterjahn, O. Kramer, A.W.A.G.: Evolution of human-competitive agents in modern computer games. Proceedings of the IEEE World Congress on Computational Intelligence (2006)

- [17] Crytek: Crytek AI Manual. (2004) Teil des CryEngine Software Development Kit.
- [18] Rabin, S.: 3.4 A* Aesthetic Optimizations. In: Game Programming Gems. Volume 1. Charles River Media (2000) 264–271
- [19] Lampel, J.: Einsatz von neuronalen netzen in einem bot für counterstrike. Facharbeit am Friedrich Schiller-Gymnasium Preetz (11 2001)
- [20] Lidén, L.: 1.4 Artificial Stupidity: The Art of Intentional Mistakes. In: AI Game Programming Wisdom 2. Volume 1. Charles River Media (2004) 41–48
- [21] Onyett, C.: Team fortress 2 interview mit charlie brown, doug lombardi und robin walker von valve software. Online Article (09 2007) <http://pc.ign.com/articles/779/779677p1.html>.
- [22] Valve: Half-Life 2 - Raising the bar. Prima Games, Division of Random House, Inc. (2004)

Computational Intelligence in Echtzeitstrategiespielen

Seminar Computational Intelligence bei Computerspielen

Simon Wessing
simon.wessing@uni-dortmund.de

Technische Universität Dortmund,
Fachbereich Informatik,
Lehrstuhl 11

Zusammenfassung Die Qualität von Computergegnern spielt eine große Rolle für den Spielspaß in Computerspielen. Dies gilt besonders für Echtzeitstrategiespiele, die vor allem die Intelligenz des Spielers fordern müssen. Leider wurde dieses Thema von den Spieleproduzenten seit Jahrzehnten vernachlässigt. Die Forschung liefert seit einigen Jahren neue Impulse mit Computergegnern, die Methoden der Computational Intelligence verwenden. Diese Ausarbeitung soll zeigen, wie sich das Geschehen informationstechnisch aufbereiten lässt und einen Überblick über den aktuellen Stand der Forschung geben.

1 Einleitung

Echtzeitstrategiespiele werden in der Regel auf einem zweidimensionalen Spielfeld ausgetragen (Abbildung 1). Sie haben keinen rundenbasierten sondern einen kontinuierlichen zeitlichen Verlauf. Zu einem Zeitpunkt können theoretisch beliebig viele nebenläufige Ereignisse auftreten. Das Ziel des Spiels ist meist, den Gegner vernichtend zu schlagen. Jeder Spieler befehligt dutzende bis hunderte Einheiten, die er zum Erreichen des Spielziels einsetzen muss. Der Zustandsraum eines Spiels ist somit sehr groß. Zusätzlich verfügen die Spieler üblicherweise nicht über vollständiges Wissen, da Teile des Spielfeldes, die nicht von eigenen Einheiten überwacht werden, nicht einsehbar sind. Diese für mehr Realismus sorgende Spieloption wird „Nebel des Krieges“ genannt. Komplexere Echtzeitstrategiespiele haben zusätzlich zum militärischen einen wirtschaftlichen Aspekt, in dem es um Ressourcengewinnung und Produktionskapazitäten geht. Gebäude müssen dann aufgebaut und verteidigt werden. Noch weiter gehen Spiele mit Technologiebäumen, in denen der Spieler die Wahl hat, Ressourcen in die Vermehrung der vorhandenen Gebäude und Einheitentypen oder in die Erforschung fortgeschrittenener Technologien zu investieren.

Dieses vielschichtige Szenario verlangt dem Spieler ebenso vielseitige Fähigkeiten ab. Buro [1] nennt als Aufgaben in Echtzeitstrategiespielen unter anderem

- Planen mit unvollständigem Wissen,



Abbildung 1. Das Echtzeitstrategiespiel Glest [2]. Links findet ein Kampf statt, rechts sind die Gebäude eines Spielers zu sehen. In der Ecke links oben befindet sich unerforschtes Terrain. Rechts unten beutet ein Arbeiter Goldressourcen aus.

- räumliches und zeitliches Denken,
- Ressourcenmanagement,
- Lernen und
- Gegnereinschätzung.

Um einen Agenten in die Lage zu versetzen, diese Aufgaben bewältigen zu können, gibt es prinzipiell viele verschiedene Möglichkeiten. Bisher wurden zu diesem Zwecke in kommerziellen Spielen vor allem erprobte Methoden der künstlichen Intelligenz (KI), wie endliche Automaten und regelbasierte Systeme, eingesetzt. Im Bereich der Computerspiele werden diese Verfahren allgemein unter dem Begriff *Scripting* zusammengefasst. Leider stagniert der Fortschritt der Intelligenz mit diesen Methoden seit geraumer Zeit. Das Verhalten so ausgestatteter Agenten wird von Menschen meist als vorhersehbar und einfallslos empfunden. Daher beruhen viele neuere Versuche, intelligentes Verhalten zu erzeugen, auf Methoden der *Computational Intelligence* (CI).

Leen und Fyfe [3] stellen als Unterschied zwischen KI und CI heraus, dass in der CI nicht versucht wird, ein Problem direkt zu lösen. Vielmehr wird versucht, ein Programm zu finden, das die Lösung selbstständig erlernen kann:

„Perhaps the definitive difference between traditional Artificial Intelligence and the studies which comprise contemporary Artificial Intelligence (which we take to encompass Evolutionary Algorithms, Artificial Neural Networks, Artificial Life and Artificial Immune Systems) is the emphasis on the program writer not solving the problem but merely

creating conditions whereby a program, if exposed to sufficient training examples, will learn to extract sufficient information from a problem in order to solve it itself. The artificial immune system does just that.“

Ihre Arbeit wird in Abschnitt 6.3 angesprochen. Zunächst möchte ich jedoch als Grundlage für alle vorgestellten Ideen beschreiben, wie man aus den Spieldaten sinnvolle Informationen herausfiltern kann. In Abschnitt 3 wird dann beschrieben, wie in heutigen, kommerziellen Echtzeitstrategiespielen der Computergegner realisiert wird. Abschnitt 4 beschäftigt sich mit Lernverfahren in Echtzeitstrategiespielen. Danach folgt der Hauptteil dieser Ausarbeitung mit Beispielen zu KIs und schließlich eine Zusammenfassung.

2 Abstraktion und Hierarchiebildung

Livingstone [4] postuliert eine an realen militärischen Hierarchien angelehnte Steuerung der Einheiten des Spielers. Dabei muss man sich bei der KI für das Spiel entscheiden, ob eine übergeordnete Hierarchieebene gegenüber der nächstniedrigeren Ebene nur beratende Funktion hat, oder ob die niedrigere Ebene weisungsgebunden ist. Hat man sich für eine mehrschichtige KI entschieden, stellt sich das Problem, den Lernvorgang zwischen den Ebenen zu koordinieren. Eine Möglichkeit, dieses Problem zu umgehen, besteht darin, nur eine Ebene zur Zeit lernen zu lassen. Durch dieses Vorgehen verkleinert man den Suchraum an Verhaltensmustern, da eine Ebene nur die Kooperation mit den festen Ebenen lernen kann. Es ist möglich, dass dadurch intelligenteres Verhalten, entstehend durch Kombination von einzeln betrachtet suboptimalen Verhaltensweisen, unentdeckt bleibt. Aufgrund des zusätzlichen Aufwandes, der durch Hierarchien verursacht wird, sollte man die Anzahl der Ebenen gering halten. Chung et al. [5] schlagen eine Einteilung in drei Ebenen vor:

1. Strategische Ebene: beinhaltet unter anderem Entscheidungen darüber, was erobert, verteidigt, ausgekundschaftet, welche Einheiten produziert und welche Ressourcen eingesetzt werden sollen.
2. Taktische Ebene: Auf dieser Ebene muss festgelegt werden, wie die Ziele der strategischen Ebene erreicht werden sollen. Zum Beispiel könnte entschieden werden, wie die eigenen Streitkräfte bei einem Feldzug aufgeteilt und eingesetzt werden.
3. Einheitenebene: Hier führt die individuelle Einheit ihren Befehl aus.

Unabhängig vom Aufbau der KI steht die Frage, wie das Spiel der KI als Wahrnehmung präsentiert werden soll. Miles et al. [6] schlagen vor, das Echtzeitstrategiepiel als ein Ressourcenallokationsproblem aufzufassen. Um so viele Ressourcen wie möglich zu erhalten, muss die KI zuerst ihre bereits verfügbaren Ressourcen identifizieren. Dann müssen räumliche oder allgemeine Ziele definiert werden, zu denen die verfügbaren Ressourcen alloziert werden. Der Begriff Allokation bedeutet, dass eine Ressource zu einem Zeitpunkt höchstens einem Ziel zugeordnet werden kann. Die Ressourcenallokation muss anschließend wieder auf Aktionen und Befehle im Spiel abgebildet werden. Räumliche Ziele, wie

zum Beispiel „Greife den Gegner an Position (x, y) an“ sind komplexer als nicht-räumliche, wie zum Beispiel „Produziere Einheiten vom Typ z “. Daher werden hierzu *influence maps* als weitere Abstraktion des Spiels eingesetzt.

2.1 Influence Maps

Offensichtlich ist es für die KI nicht sinnvoll, jede Einheit einzeln wahrzunehmen, da Einheiten üblicherweise nicht einzeln, sondern in Gruppen agieren. Auch das Spielfeld kann zu groß sein, um als Ganzes betrachtet zu werden. Influence Maps sind eine Möglichkeit, um das Geschehen auf dem Spielfeld auf numerische Werte abzubilden. Dazu wird das Spielfeld in ein größeres, rechteckiges Raster aufgeteilt, siehe Abb. 2. Beispielsweise kann eine Influence Map für gegnerische Einheiten erstellt werden, indem die Kampfstärke aller Einheiten, die sich in einem Quadrat aufhalten, aufsummiert und als Wert diesem Quadrat zugewiesen wird. Nun könnte diese mit einer Influence Map für den eigenen militärischen Einfluss verrechnet werden, und man erhielt eine Influence Map für die militärischen Machtverhältnisse auf der Karte.

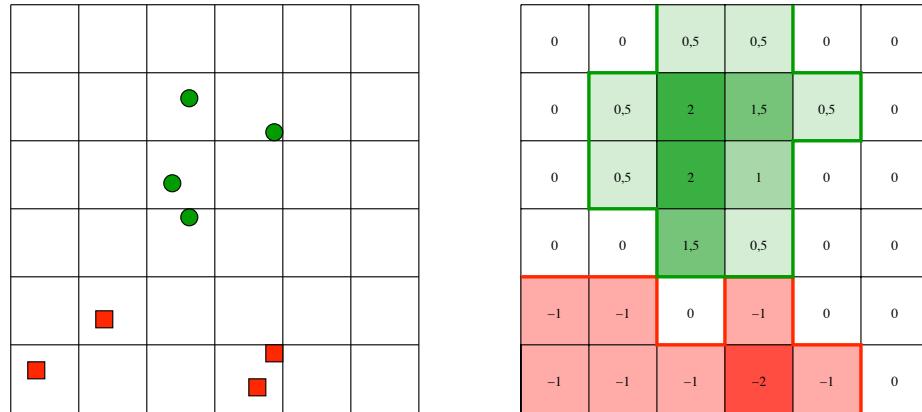


Abbildung 2. Links die befreundeten (rund) und gegnerischen (quadratisch) Einheiten auf dem Spielfeld. Rechts die daraus berechnete Influence Map. Einer Einheit wurde in ihrem Quadrat ein Einfluss von 1 und in den horizontal und vertikal benachbarten Quadranten ein Einfluss von 0,5 zugerechnet. Die gegnerischen Einflüsse wurden von den eigenen subtrahiert.

Miles et al. [6] haben diesen Ansatz auf Bäume von Influence Maps verallgemeinert. Blätter stellen dabei normale Influence Maps dar. Innere Knoten erhalten ihre Map durch beliebige arithmetische Verknüpfungsoperationen der Maps in den Kindknoten. Das Verändern der Gewichte und Operationen in den Bäumen durch einen evolutionären Algorithmus stellt in der KI der Autoren den Lernvorgang dar. Die Influence Maps werden zum Finden und Bewerten

von räumlichen Zielen benötigt. Für jede mögliche räumliche Zielkategorie, wie zum Beispiel Angriff, Verteidigung usw., existiert ein eigener Baum. Wenn der Baum ausgewertet wird, werden für alle lokalen Optima der Map an der Wurzel Ziele definiert. Das Ziel wird dann mit einem Koeffizienten aus Nutzen und Kosten bewertet, in den der Wert der Influence Map an der entsprechenden Stelle eingeht. In einer Endlosschleife wird jeweils das Ziel mit dem höchsten Koeffizienten zur Realisation ausgewählt.

2.2 SORTS

Wintermute et al. [7] haben die Schnittstelle SORTS für das Echtzeitstrategiespiel ORTS [8] geschrieben. Eine regelbasierte KI, die diese Schnittstelle benutzte, gewann zwei von drei Disziplinen des AIIDE 2006 ORTS Wettbewerbes. In diesem Wettbewerb treten ausschließlich computergesteuerte Spieler gegeneinander an. Im Wesentlichen setzt die Schnittstelle die in Abschnitt 2 genannten Ideen um. Die Autoren haben allerdings Wert darauf gelegt, psychologische und biologische Erkenntnisse über die menschliche Wahrnehmung in ihre Arbeit einzubeziehen, um eine möglichst „menschliche“ Abstraktion zu erhalten. SORTS führt zwei Filterungen durch: Gruppieren von Objekten und Fokussierung der Aufmerksamkeit.

Objekte lassen sich in SORTS nach beliebigen Kombinationen der drei Kriterien Objekttyp, Partei und Nachbarschaft gruppieren. Der Objekttyp bezeichnet dabei die Art der Einheit oder des Gebäudes. Die Partei gibt an, für wen das Objekt arbeitet, beziehungsweise kämpft. Für die Nachbarschaft kann man den Radius der Entfernung, die Objekte in einer Gruppe maximal zueinander haben dürfen, angeben. Stellt man den Entfernungsradius auf null, nimmt man einzelne Einheiten wahr. Für eine Gruppe werden die Informationen der Objekte, wie zum Beispiel Angriffskraft, Verteidigungsvermögen usw. zusammengefasst. Sobald die Gruppe einen Auftrag erhält, wird die Gruppierung nicht mehr geändert. Das heißt, weitere Einheiten können der Gruppe nicht mehr beitreten und die Nachbarschaftsbedingung darf verletzt werden, ohne dass die Gruppenzugehörigkeit aufgehoben wird.

Da auf großen Spielfeldern auch nach der Gruppierung von Objekten noch zu viele Informationen anfallen können, um sie zu verarbeiten, beschränkt SORTS die Wahrnehmung auf einen Teilausschnitt des Spielfeldes, siehe Abb. 3. Dieses rechteckige Sichtfeld ist fest in drei Spalten und drei Zeilen aufgeteilt, kann aber eine variable Größe haben. Ein beliebiger Punkt in diesem Feld kann fokussiert werden. Eine feste Anzahl von Gruppen, die diesem Punkt am nächsten sind, werden der KI detailliert dargestellt. Die Eigenschaften der restlichen Gruppen werden ähnlich wie bei den Influence Maps für ein ganzes Rechteck zusammengefasst. Der Unterschied zu ihnen ist, dass hier ein Rechteck mehrere Eigenschaften und somit mehrere numerische Werte enthält. Die Autoren nennen dieses Konzept *feature map*.

Die KI, für die diese Schnittstelle konzipiert wurde, war in zwei Schichten aufgeteilt. Die strategische Ebene operierte auf der beschriebenen Abstraktion des Spiel, während die einzelnen Einheiten als endliche Automaten implementiert

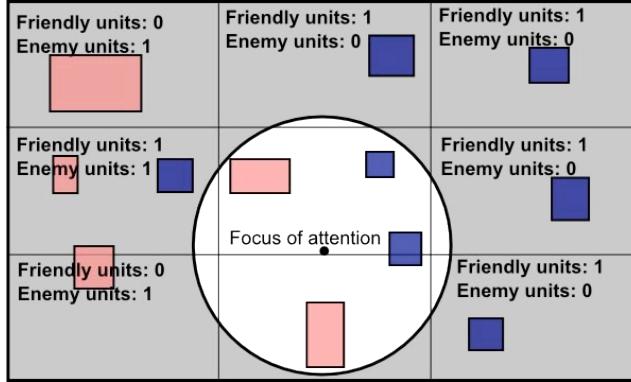


Abbildung 3. Ein Teilausschnitt des Spielfeldes, wie die KI ihn durch die SORTS-Schnittstelle wahrnehmen würde. Er wird eingeteilt in neun Quadranten, in denen nur die Anzahlen der Einheiten wahrgenommen werden. Im engeren Fokus werden auch die konkreten Gruppen (hier dargestellt durch Rechtecke) mit ihren Positionen betrachtet. Entnommen aus [7].

waren und ungefilterten Zugriff auf das Spiel hatten. Die Einheiten hatten ein Standardverhalten, das angewandt wurde, wenn keine anderslautenden Befehle von der strategischen Ebene vorlagen. Dies entspricht auch dem, was Chung et al. [5] vorgeschlagen haben.

3 Herkömmliche KI in Echtzeitstrategiespielen

Die klassische Technik, um ein Echtzeitstrategiespiel mit künstlicher Intelligenz auszustatten, ist Scripting. Bei diesem Verfahren wird einfach ein Drehbuch geschrieben, nach dem der Computerspieler handelt. Sehr einfach und erfolgreich ist zum Beispiel die *rush*-Taktik. Dabei baut man bei Spielbeginn möglichst schnell viele billige Einheiten, um den Gegner anzugreifen bevor er seine Verteidigung aufgebaut hat. Da es so natürlich zu einem immer gleichen Ablauf kommt, wird es für den menschlichen Spieler langweilig, sobald er den Computer das erste mal besiegt hat. Andererseits ist es für die Spieleproduzenten aufwendiger, mehr als eine Strategie zu entwickeln und zu testen. Weitere Argumente der Hersteller für Scripting sind die Vorhersagbarkeit des KI-Verhaltens, sowie die Möglichkeit, den gesamten KI-Programmteil aus der Programmlogik auszulagern. Statt im Programm wird die KI separat in einer Scriptsprache formuliert und dann während des Programmablaufes geparsert. Ein Grund für dieses Vorgehen ist, dass die Arbeit dadurch nicht unbedingt von Programmierern übernommen werden muss. Ein großer Nachteil aller Script-KIs ist, dass sie in eine Endlosschleife von einfallslosen Angriffen übergehen, sobald das Script abgearbeitet ist.

Von Ponsen und Spronck [9] wurde dynamisches Scripting für Echtzeitstrategiespiele entwickelt. Um auch Adaptivität während des laufenden Spiels zu

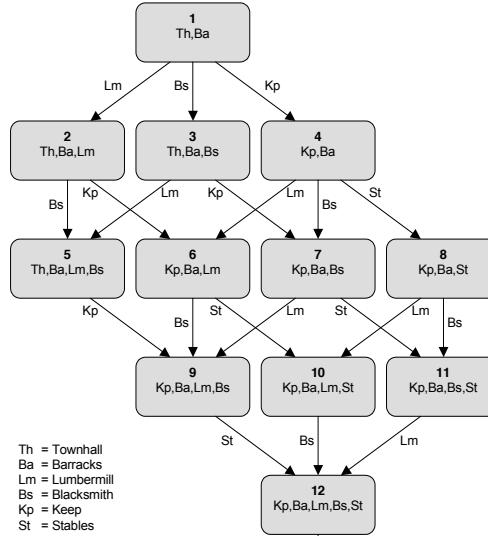


Abbildung 4. Ein Teil des Zustandsgraphen im Echtzeitstrategiespiel Wargus. Die Zustände basieren auf den Gebäuden, die ein Spieler bisher gebaut hat. Somit enthält der aktuelle Spielzustand ungefähre Information, wie weit das Spiel schon fortgeschritten ist. Entnommen aus [9].

erhalten, wurde das Spiel in Zustände eingeteilt, die auf den bisher gebauten Gebäuden, also der Erforschung des Technologiebaums, basieren, siehe Abbildung 4. Jedem dieser Zustände sind dabei Regeln zugeordnet, die in zufälliger Reihenfolge abgearbeitet werden. Regeln sind dabei als taktische Züge anzusehen, die in die vier Kategorien Gebäudebau, Forschung, Ressourcengewinnung und Gefecht eingeteilt sind. Normalerweise werden die Regeln für das Script manuell entwickelt, wofür viel Expertenwissen benötigt wird. Da solches Wissen fehlerhaft und unvollständig sein kann und zudem auch schwer erhältlich ist, ist es wünschenswert, auch ohne auszukommen. Daher suchten die Autoren zusätzlich mit einem evolutionären Algorithmus offline nach besseren Taktiken, die sie dann manuell in das Expertenwissen einbanden. Dies führte zu besseren Leistungen des dynamischen Scripts, allerdings blieben rush-Taktiken den dynamischen immer überlegen. In einer Weiterentwicklung [10] wurden dann die durch den EA gefundenen Regeln automatisch in die Wissensbasis übernommen.

Da die Erweiterung des Scripting etwas umständlich wirkt, soll in dieser Ausarbeitung innovativeren Ideen Vorrang gegeben werden. Vom Aufgabenbereich her vergleichbar sind die Verfahren in Abschnitt 6. Ein weiterer Ansatz, die herkömmliche KI durch Elimination von falschem Expertenwissen zu verbessern, wird in Abschnitt 4.2 beschrieben.

4 Mögliche Lernverfahren in Echtzeitstrategiespielen

Häufig verwendete Lernverfahren in Echtzeitstrategiespielen sind Co-Evolution und Temporal Difference Learning (TDL). Während bei TDL nur ein Spieler – eventuell sogar gegen sich selbst – spielt, existiert bei Co-Evolution eine Population von Spielern, die gegeneinander spielen und durch die Spielergebnisse bewertet werden. Die Methoden sollen hier nicht im Detail erklärt, sondern nur der Bezug zu Echtzeitstrategiespielen exemplarisch aufgezeigt werden.

4.1 Co-Evolution

Die Besonderheit der Co-Evolution liegt darin, dass die Fitness eines Individuums durch Vergleich mit den anderen, auch der Evolution unterworfenen, Individuen in der Population gemessen wird. Daher ändert sich durch Mutationen der Individuen auch die Fitnessfunktion. Im Kontext von Spielen geschieht die Ermittlung der Fitness in der Regel durch Spiele von Individuen gegeneinander. Dieses Vorgehen hat zur Folge, dass die Fitness nicht absolut, sondern nur relativ zu den getesteten Gegnern ist.

Zusätzlich zu den schon in Abschnitt 2 erwähnten Schwierigkeiten mit Hierarchien bestehen weitere, die von Livingstone [4] beschrieben werden: Die relative Fitness wird durch den von Van Valen [11] beschriebenen *Red-Queen-Effekt* zum Problem. Seine Hypothese besagt, dass jedes Individuum in einem evolutionären System sich ständig weiterentwickeln muss, um nicht bezüglich der Fitness, relativ zu den Wettbewerbern gesehen, zurück zu fallen. Es ist also umgekehrt auch möglich, dass sich alle Individuen unter Beibehaltung der Rangfolge verschlechtern. In diesem Fall verschlechtert sich die absolute Fitness der Individuen, ohne dass dies bemerkt und bestraft würde. Um das zu verhindern, kann man weitere Gegner mit fixer Strategie zum Vergleich heranziehen, oder eine Population von früheren besten Spielern bewahren.

Ein weiteres Problem besteht in der Ausbalanciertheit von Echtzeitstrategiespielen: Damit ein Spiel als gerecht empfunden wird, darf natürlich kein Spieler einen Vorteil haben. Daher verfügt normalerweise keine der Seiten über einen überlegenen Einheitentyp. Jeder Einheitentyp hat bestimmte Vor- und Nachteile, die ihn für manche Aufgaben und gegen manche anderen Einheitentypen besonders geeignet machen. Zum Beispiel wäre eine Konstellation denkbar, in der für die Einheitentypen A , B und C die intransitive Beziehung $A < B$, $B < C$ und $C < A$ bezüglich der Überlegenheit gilt. Daher kann es keine global optimale Strategie für die Einheitenwahl geben. Die optimale Strategie ist immer abhängig von der Strategie des Gegners. Dadurch entsteht eine Zirkulation in der Evolution, die nicht zu einem Fortschritt führt, siehe Abbildung 10. Eine Möglichkeit, dieses Problem zu vermeiden, besteht in der Benutzung eines Gedächtnisses für Paare von gegnerischen und eigenen Strategien.

4.2 Temporal Difference Learning

TDL gehört in die Kategorie des verstärkenden Lernens, welches dadurch gekennzeichnet ist, dass der Lernende Informationen darüber erhält, ob sein Verhalten

insgesamt gut oder schlecht war, aber nicht *was* im speziellen zu dieser Bewertung führt. Letzteres ist in unserem Fall in der Regel nicht möglich, da nicht genug Informationen zur Verfügung stehen. TDL ist für Situationen gedacht, in denen eine Folge von Vorhersagen für einen festen Zeitpunkt benötigt wird. Das Ziel im Kontext von Spielen könnte zum Beispiel eine korrekte Vorhersage des Spieldurchgangs aufgrund einer Spielsituation sein. Auch Probleme wie Mustererkennung oder Funktionsapproximation lassen sich als Vorhersageprobleme begreifen. Das Besondere an TDL ist die Lernregel, die mit der Differenz zwischen zwei zeitlich aufeinander folgenden Vorhersagen arbeitet. Herkömmliche Verfahren des überwachten Lernens arbeiten stattdessen mit dem Fehler zwischen der Vorhersage und dem tatsächlichen Ergebnis. Dies hat den Vorteil, dass TDL auf natürliche Weise ein inkrementelles Verfahren ist. Sutton und Barto [12] bieten eine grundlegende Einführung in TDL und verstärkendes Lernen allgemein. Hier soll nun eine interessante Anwendung in Echtzeitstrategiespielen präsentiert werden.

Kerbusch [13] benutzte TDL, um Expertenwissen in der regelbasierten KI aus Abschnitt 3 zu ersetzen. Das dynamische Scripting stellte online eine Strategie aus einer Folge von Taktiken zufällig zusammen. Die Wahrscheinlichkeit einer Taktik, gewählt zu werden, ergab sich aus dem Erfolg der Taktik in früheren Spielen. Der Erfolg wiederum wurde durch eine Funktion gemessen, in die unter anderem die gewichtete Summe der ausgeschalteten gegnerischen Einheiten einging. Die Gewichte entsprachen ursprünglich der Anzahl der Lebenspunkte einer Einheit und waren damit eher schwach begründet. Wie man in Tabelle 1 sieht, unterscheiden sich die neu erlernten Gewichte erheblich. Besonders hervor sticht die Bewertung des Arbeiters. Da Arbeiter keine militärische Relevanz haben, wurden sie von den Experten, die die Gewichte festlegten nicht als vielversprechende Ziele eingestuft. Völlig entgegengesetzt dazu war der Arbeiter bei den erlernten Gewichten die wertvollste Einheit. Eine plausible Erklärung dafür ist, dass die Arbeiter das Rückgrat der Wirtschaft eines Spielers sind und sich zudem hauptsächlich in der Nähe der Basis aufhalten. Daher muss ein Spieler, der in der Lage ist, Arbeiter anzugreifen, schon einige Verteidigungsanlagen erfolgreich durchbrochen haben. In Experimenten ließ sich dementsprechend auch eine deutliche Verbesserung der KI durch die Benutzung der neuen Gewichte feststellen.

5 Intelligenz auf Einheitenlevel

Die Anforderungen an eine einzelne Einheit in einem Echtzeitstrategiespiel unterscheiden sich grundsätzlich nicht von denen an einen Agenten in anderen Spielen. Daher soll das Thema hier nur am Rande angesprochen werden. Als herkömmliches Verhaltensmodell für Einheiten in Echtzeitstrategiespielen dürfen endliche Automaten angesehen werden. Ausführlichere Erläuterungen dazu lassen sich beispielsweise in [14] nachlesen. Diese schon sehr alte Technik bedingt ein deterministisches Verhalten der Einheiten, dass sich in klar abgrenzte Zustände unterteilen lässt. Die Zustände abstrahieren vom konkreten Spiel. Das

Tabelle 1. Vergleich der alten Bewertungen der Einheiten mit den neu Erlernten. Wie man sieht unterscheiden sich die Werte deutlich. Der Arbeiter wurde von der unwichtigsten Einheit zur wichtigsten, andere wurden herabgestuft oder sogar negativ bewertet.

Einheitentyp	Alter Wert	Neuer Wert
Arbeiter	30	239
Fußsoldat	50	24
Bogenschütze	60	-2
Katapult	100	79
Ritter	100	94
Ranger	70	-13

Verhalten in einem Zustand kann durch ein Script definiert werden, in dessen Ablauf weitere Parameter aus dem Spiel eingehen. Zum Beispiel muss bei einer Zustandstransition in den Zustand „Angreifen“ in Abbildung 5 das Ziel des Angriffs angegeben werden. Falls sich das Ziel nicht in Reichweite befindet, sollte die Einheit sich dem Ziel zunächst annähern.

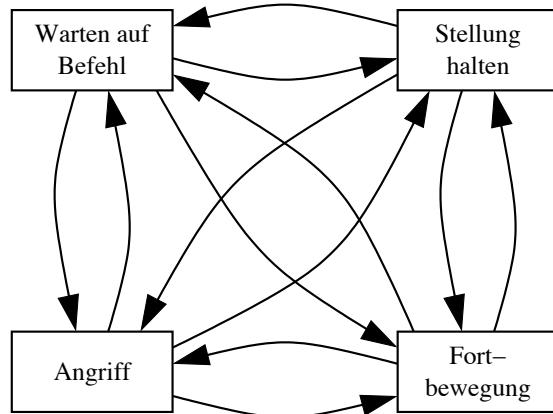


Abbildung 5. Ein endlicher Automat, wie er zum Beispiel einen Soldaten beschreiben könnte. Die Zustandsübergänge können durch Befehle der übergeordneten Hierarchieebene der KI (oder des menschlichen Spielers) oder durch Ereignisse im Spielablauf ausgelöst werden.

Wenn keine übergeordnete Kontrollinstanz in der KI existiert, kommt wegen der hohen Anzahl von Einheiten in Echtzeitstrategiespielen der Kommunikation untereinander hohe Bedeutung zu. Kooperatives Verhalten ist dann notwendig, um im Wettbewerb erfolgreich zu sein. Ein Spiel, in dem dieser Aspekt spielbestimmend ist, ist NERO (*neuroevolving robotic operatives*).



Abbildung 6. Screenshot vom Training in Nero. Im rechten Menü werden durch Schieberegler bestimmte Verhaltensaspekte belohnt oder bestraft. Auf dem Spielfeld kann der Spieler Mauern und Gegner postieren, um den Robotern Trainingsgelegenheit zu bieten.

Stanley et al. [15] haben mit NERO ein Strategiespiel mit einem innovativen Konzept entwickelt. Das eigentliche Spiel besteht daraus, eine Gruppe von Robotern durch verstärkendes Lernen zu trainieren. Gebäude und Techonologiebäume gibt es in diesem Spiel nicht. Jeder Roboter besitzt ein individuelles neuronales Netz als Gehirn, hat aber zu Beginn des Trainings überhaupt kein Wissen über seine Umwelt. Aufgabe des Spielers ist es, den Robotern in einer Art Sandkasten Schritt für Schritt immer komplexeres Verhalten anzutrainieren. Dazu muss er eine das Verhalten der Roboter bewertende Zielfunktion festlegen und eine geeignete Trainingsumgebung schaffen, siehe Abb. 6. Das trainierte Team von Robotern kann man dann in einem Kampf gegen andere computergesteuerte Teams antreten lassen. Der Spieler ist dann nur noch Zuschauer und hat keine Möglichkeit, in den Kampf einzugreifen.

Die neuronalen Netze der Roboter erhalten Eingaben über gegnerische Einheiten, Hindernisse und ob auf den Roboter gerade geschossen wird. Dazu wurde der Umkreis um den Roboter in fünf gleich große Sektoren eingeteilt. Jeder Sektor ist eine eigene Eingabeveriable, die die Entfernung des jeweiligen Objekts in dieser Richtung angibt. Als Ausgaben aus dem Netz ergeben sich Bewegung links/rechts, Bewegung vorwärts/rückwärts und Schießen. Der Lernvorgang geschieht durch Evolution der neuronalen Netze. Die Roboter haben eine feste Lebensdauer, an deren Ende die Zielfunktion ausgewertet wird und die Selektion stattfindet. Es werden bei der Mutation und Rekombination sowohl die Gewichte, als auch die Struktur der neuronalen Netze verändert. Zu Beginn sei-

nes Lebens hat der Roboter ein minimales neuronales Netz, dass mit der Zeit strukturell immer komplexer wird, falls sich das als vorteilhaft herausstellt. Da strukturelle Veränderungen meist auf Anhieb keine Verbesserung der Fitnessfunktion bringen, werden neue Strukturen für eine Weile in Subpopulationen geschützt.

Wegen seines neuartigen Spielprinzips kann man NERO nicht eindeutig den Echtzeitstrategiespielen zuordnen. Nur der Kampfmodus, in dem die fertigen Teams gegeneinander antreten, kann wirklich als Echtzeitstrategie gelten. Der andere Teil des Spiels, das Lernen in Echtzeit, war erst durch die Entwicklung eines Lernalgorithmus möglich, der so schnell ist, dass der Lernfortschritt innerhalb von wenigen Generationen erkennbar ist und dem Spieler nicht langweilig wird.

6 Planen

Die in diesem Abschnitt vorgestellten Verfahren sind vor allem gut für die in Abschnitt 2 „Strategische Ebene“ genannte Abstraktion des Spiels geeignet. Auf der strategischen Ebene muss die KI Vorhersagen über das gegnerische Verhalten und den Spielverlauf machen, um das eigene Verhalten bewerten zu können. Da das gegnerische Verhalten nicht bekannt ist, muss sich die KI ein Modell davon erstellen, mit dem sie arbeiten kann. In der experimentellen Anwendung der folgenden Verfahren haben die Autoren aber meist einfache Spiele benutzt, weswegen nur ein geringer Bedarf für Abstraktion bestand. Ohne Abstraktion muss allerdings jeder Plan exakt definiert sein in Bezug auf die teilnehmenden Einheiten, Situationen, auf die der Plan anwendbar ist, mögliche Reaktionen des Gegners und vieles mehr. In einem regelbasierten System ist dazu sehr viel Expertenwissen nötig. Die folgenden Verfahren sind darauf ausgelegt, diesen Bedarf zu minimieren.

6.1 Monte-Carlo-Simulation

Bei der Monte-Carlo-Simulation benutzt man eine große Anzahl an Zufallsexperimenten, um das unbekannte Verhalten eines Systems zu schätzen. Im konkreten Fall der von Chung et al. [5] entwickelten KI ist das unbekannte Verhalten das des Gegenspielers. Bei der Monte-Carlo-Simulation wird es durch ein rein zufälliges Verhalten simuliert.

Die Spielvariante, die in diesem Fall gespielt wurde, nennt sich *capture the flag* (CTF). Im Spiel treten zwei Gruppen von Einheiten gegeneinander an. Die zu Anfang zugeteilten Einheiten sind die einzigen Ressourcen der Spieler. Jede Gruppe besitzt eine transportable Flagge, die aber nur vom gegnerischen Team bewegt werden kann. Die beiden Gruppen starten an unterschiedlichen Positionen auf dem Spielfeld. Die Position der gegnerischen Flagge ist bekannt, die der gegnerischen Gruppe nicht. Ziel des Spiels ist es für beide Parteien, als erste die gegnerische Flagge in die eigene Basis zu entführen. Um das Spiel gerecht zu

gestalten, wurden von den Autoren nur symmetrische Spielfelder verwendet, auf denen die gegnerischen Teams an entgegengesetzten Enden starteten (Abb. 7).

Die Definition eines Plans ist anwendungsabhängig. Im Fall der Echtzeitstrategiespiele wären dies elementare, abstrakte Tätigkeiten wie Erforschen, Angreifen oder das Erreichen irgendeines Zwischenziels. Ein Plan kann dabei auch Parameter, wie zum Beispiel den Ort, an dem er ausgeführt wird, haben. Außerdem können auch Sequenzen von Plänen wiederum als Plan angesehen werden. Der von den Autoren entworfene Algorithmus names MCPlan verläuft grundsätzlich wie folgt:

1. Erzeuge zufällig einen Plan für die KI.
2. Simuliere diesen Plan ausgehend vom aktuellen Zustand gegen einen zufällig gewählten gegnerischen Plan. Bewerte den Erfolg des benutzten Plans.
3. Speichere das Ergebnis der Ausführung des KI-Plans.
4. Führe die Schritte 2 und 3 für eine feste Anzahl von Wiederholungen aus. Führe danach die Bewertungen der einzelnen Simulationsläufe zu einer Gesamtbewertung des Plans zusammen.
5. Wiederhole die Schritte 1 bis 4 so oft wie unter den gegebenen Ressourcenbeschränkungen (vor allem Zeit) möglich.
6. Wähle den Plan mit der besten statistischen Bewertung zur tatsächlichen Ausführung aus.

Das zufällige Erzeugen eines Planes geschieht, indem zufällige Abfolgen von Aktionen mit zufälligen Parametern gebildet werden. Der in Schritt 2 erwähnte „aktuelle Zustand“ steht für eine Abstraktion des Spiels im jeweiligen Moment. Wie stark die Abstraktion ist, ist dem Implementierenden überlassen. Da dieser Schritt aber möglichst oft ausgeführt werden muss, um eine aussagekräftige Bewertung des Plans zu erhalten, sollte auf eine geringe Laufzeit geachtet werden. Die Simulation eines Plans wird nur für ein festes Zeitfenster durchgeführt. Da das Spiel bei Simulationsende in der Regel noch nicht beendet ist, wird Expertenwissen benötigt, um den Erfolg des Plans bewerten zu können. Allerdings kann auch dieses Wissen zumindest teilweise automatisch erlernt werden, wie zum Beispiel Kerbusch [13] demonstriert hat.

Zur Gesamtbewertung des Plans in Schritt 4 sind verschiedene Funktionen denkbar. Am sichersten ist es, entsprechend der MiniMax-Strategie in der Spieltheorie, das Minimum der Einzelbewertungen aus Schritt 2 zu nehmen. Die Funktion, die im CTF-Spiel für diese Einzelbewertungen verwendet wurde, enthielt Komponenten bezüglich Ressourcen, Erforschung und Überwachung des Spielfeldes und Positionen der Flaggen. Die Gewichte, die innerhalb der Komponenten verwendet wurden und bei der Kombination der Komponenten eingingen, wurden von den Autoren manuell ausgewählt. Da die Gewichte fest waren, war diese KI nicht in der Lage zu lernen. Die Qualität der Bewertungsfunktion wurde durch Tests ohne Gegner und gegen einen zufällig agierenden Gegner abgesichert. Ein zufällig handelnder Gegner entspricht einem Gegner mit zufälliger Bewertungsfunktion.

Die Simulation der Pläne wurde durchgeführt, indem die Einheitenbewegungen schrittweise auf einem größeren Raster als dem Spielfeld entsprechend der

Pläne durchgeführt wurden. Nach jedem Schritt wurden, falls in der Nähe gegnerische Einheiten waren, Kämpfe simuliert, indem die Lebensenergie der Einheiten verringert wurde. Der Verlauf der Simulation musste dabei nicht exakt dem tatsächlichen Spielverlauf entsprechen. Zum Beispiel war es möglich, dass gegnerische Einheiten im Spiel nicht in Angriffsweite waren, obwohl es in der Simulation wegen des größeren Rasters so wirkte.

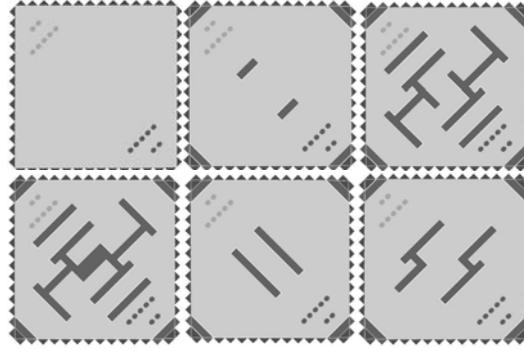


Abbildung 7. Die im Experiment benutzten Spielfelder. Die beiden gegnerischen Teams starten in den Ecken links oben und rechts unten. Die Balken in der Spielfeldmitte stellen Hindernisse dar. Entnommen aus [5].

In Experimenten haben die Autoren das Verhalten der KI in Abhängigkeit von den Parametern Spielfeldtyp, Einheitenanzahl, Weite der Vorhersagen und Anzahl an erwogenen Plänen untersucht. Je komplexer das Spielfeld war, desto erfolgreicher spielte die KI. Die Einheitenanzahl hatte keinen wesentlichen Einfluss auf die Strategie. Es muss auch nicht unbedingt besser sein, weiter in die Zukunft vorherzusagen. Dies liegt vermutlich am Fehler, der durch die Simulation gemacht wird. Mehr Pläne in Betracht zu ziehen, erwies sich aber immer als hilfreich. Bei einfachen Aufgaben stagnierte der Erfolg zwar schon ab 16 betrachteten Plänen, bei schwierigeren Aufgaben brachte dem Team aber auch der Wechsel von 64 auf 128 Pläne noch einen deutlichen Erfolgszuwachs. An dieser Stelle wäre wahrscheinlich ein guter Ansatz, mit einer lokalen Suche ausgehend von bereits gefundenen Plänen nach besseren zu suchen, anstatt immer mehr zufällige Pläne zu generieren. Im nächsten Abschnitt wird ein Verfahren vorgestellt, das dies tut.

6.2 Case-injected EA

Miles und Louis [16], [17] benutzen einen evolutionären Algorithmus (EA), um Pläne für das Verhalten der KI zu finden. Um die Suche nicht mit einer völlig zufällig initialisierten Population zu beginnen, wird die aus Plänen bestehende Population des EA um ältere Pläne ergänzt (*case-injection*). Die Technik wurde

in dem Echtzeitstrategiespiel Strike Ops angewendet. Bei diesem Spiel nimmt ein Spieler eine defensive Rolle ein, in der er seine Gebäude und Verteidigungsanlagen am Boden erhalten muss. Der andere, offensive Spieler versucht, diese durch Luftangriffe zu zerstören. Die Spannung des Spiels speist sich aus einem hohen Maß an Unsicherheit. Überraschungsmomente, die durch Fallen und Wetterumschwünge entstehen, und die unterschiedlichen Wirkungsgrade der Waffensysteme gegeneinander bedingen häufige Planänderungen.

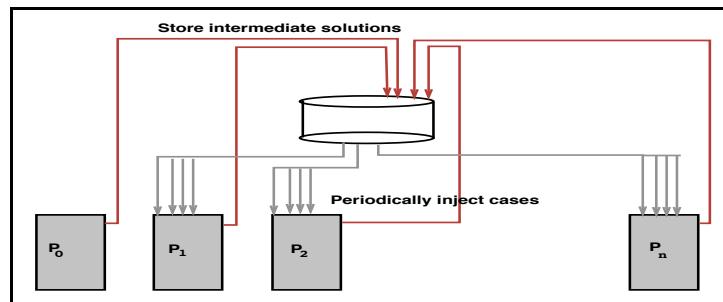


Abbildung 8. Der Langzeitspeicher für bisher gefundene Individuen (mitte, oben) und die zeitliche Folge von Problemen P_0, \dots, P_n , zu deren Bearbeitung sie wieder herangezogen werden. Entnommen aus [16].

Das Besondere an dem von den Autoren verwendeten evolutionären Algorithmus ist die Verbindung mit einem Langzeitgedächtnis für gefundene Lösungen, siehe Abb. 8. In diesem Gedächtnis werden Ergebnisse aus früheren Suchen gespeichert. Ist es leer, verläuft der Algorithmus wie ein normaler EA. Andernfalls werden von Zeit zu Zeit die schlechtesten Individuen in der Population durch Individuen aus dem Gedächtnis ersetzt, die dem besten Individuum der Population ähnlich sind, siehe Listing 1.1. Die Ähnlichkeit von Individuen wird allerdings nur über den Hammingabstand der binären Kodierung definiert, obwohl eine ähnliche binäre Kodierung keine Ähnlichkeit der Individuen garantiert. Die Autoren rechtfertigen ihre Wahl jedoch mit der Einfachheit der Berechnung des Hammingabstandes. Unpassende Lösungen würden wegen ihrer geringen Fitness und der elitistischen Selektion wieder aus der Population eliminiert. Ein weiteres Problem ist, dass die injizierten Pläne zwar dem aktuell besten Plan ähnlich sind, jedoch in völlig anderen Situationen entstanden sein können. Die Autoren nehmen an, dass ähnliche Lösungen durch ähnliche Probleme entstanden sind und genügend sinnvolle Informationen enthalten, um die Suche in die richtige Richtung zu lenken. Die Fitness der Individuen wird berechnet, indem der weitere Spielverlauf simuliert wird. Aufgrund der Simulation werden Wahrscheinlichkeiten für das Erreichen von Zielen und Überleben von Objekten angenommen, welche in die Fitnessfunktion eingehen. Wenn die Suche des EA über mehrere Generationen stagniert, wird sie abgebrochen, bevor die maximale Anzahl an

Funktionsauswertungen erreicht ist. In den Experimenten betrug diese Anzahl zwischen 400 und 1600 Funktionsauswertungen.

Listing 1.1. Pseudocode des Case-Injected-EA. Fettgedruckt sind die Bestandteile, die hinzugefügt wurden, um den Algorithmus mit dem Gedächtnis zu verbinden. $P(t)$ bezeichnet die Population zum Zeitpunkt t .

```

t = 0;
Initialize P(t);
While( not termination condition )
Begin
    if(t % injectPeriod == 0)
        InjectFromCaseBase(P(t), CaseBase);
    Select P(t+1) from P(t);
    Crossover P(t+1);
    Mutate P(t+1);
    t = t + 1;
    if(NewBest(P(t)))
        CacheNewBestIndividual(P(t), Cache);
End
SaveCacheToCaseBase(Cache, CaseBase);

```

Bevor das Spiel beginnt, haben beide Seiten die Möglichkeit, ihre zu benutzenden Ressourcen auszuwählen. Der offensive Spieler plant außerdem die Angriffsroute, der defensive Spieler die Positionen der Verteidigungsanlagen und Fallen. Die Routenplanung geschieht, indem die KI zu besuchende Wegpunkte festlegt, zwischen denen die Routen mit dem Wegfindungsalgorithmus A* berechnet werden. Die Wegkosten, die der A*-Berechnung zu Grunde liegen, können von der KI durch Parameter justiert werden, allerdings versucht die KI nicht explizit die Positionen der Verteidigungsanlagen vorherzusagen (Abb. 9).

Nur der offensive Spieler nutzte in den Experimenten die EA-basierte KI. Der defensive Spieler wurde durch ein Script gesteuert. Das Resultat der Experimente war, dass die Case-Injection gegenüber dem normalen EA tatsächlich eine kürzere Laufzeit benötigte und bessere Pläne fand. Die Verbesserungen fielen umso deutlicher aus, je komplexer die Mission war. Die Autoren stellten außerdem fest, dass das Ausmaß der Neu-Planungen mit der Tragweite des auslösenden Ereignisses korrelierte. So verursachten unbedeutende Ereignisse nur geringe Planänderungen, während starke Änderungen der Lage praktisch komplett neue Pläne nach sich zogen.

Die Autoren nennen als eines der verbleibenden Probleme mit ihrem Ansatz die zu lange Reaktionszeit, bis ein Plan gefunden ist. Sie schlagen vor, zu diesem Zweck die Zielfunktion zu vereinfachen. Auch die Ähnlichkeitsbestimmung von Individuen und die Auswahl aus dem Gedächtnis wäre noch verbesserungsfähig. Die Idee, Mutation mit einem Gedächtnis zu verbinden, ist hingegen auch an



Abbildung 9. Ein Screenshot aus dem Spiel Strike Ops. Der angreifende Spieler startet rechts oben in der Ecke. Von dort ausgehend sind die Flugrouten der Flugzeuge zu den Zielen im Vordergrund und zurück zu sehen. Die Kreise stellen die Reichweite der Verteidigungsanlagen dar. Entnommen aus [16].

anderer Stelle gekommen. In künstlichen Immunsystemen ist ein Gedächtnis für Bedrohungen auch ein Bestandteil.

6.3 Künstliches Immunsystem

Immunsysteme sind die Abwehr höherer Lebewesen gegen Bakterien und Viren. Für den Einsatz in Spielen besitzt das Immunsystem einige nützliche Aspekte, die von Fyfe [18] beschrieben werden. Es ist teilweise angeboren, kann sich aber auch durch den Kontakt mit Feinden an diese anpassen. Das Immunsystem produziert fortlaufend sogenannte Antikörper, die Eindringlinge anhand ihrer Oberflächenstruktur erkennen und unschädlich machen können. Erfolgreiche Antikörper werden mit einer hohen Mutationsrate vervielfältigt. Da die Ressourcen des Immunsystems begrenzt sind, bedeutet ein hoher Anteil von Antikörpern des einen Typs einen geringen Teil von anderen Typen.

Dieses Konzept wurde von Fyfe auf ein stark vereinfachtes Strategiespiel übertragen. In dem Spiel traten zwei Gegner mit Armeen bestehend aus Kavallerie, Infanterie und Bogenschützen gegeneinander auf einem einfachen Gitter als Spielfeld an. Die Mengenverhältnisse der Einheitentypen in der Armee waren durch die Spieler wählbar. Die Kräfteverhältnisse zwischen den Einheitentypen waren wie in Abbildung 10 gegeben. Bei Zusammenstößen gleichartiger Einheiten wurde der Gewinner zufällig gewählt. Zusätzlich schlugen zwei Einheiten vom selben Typ auf benachbarten Feldern alle anderen einzelnen Einheiten. Wie schon in Abschnitt 4.1 beschrieben, gibt es in diesem Spiel keine global optimale Strategie. Zunächst spielte die KI gegen einen Gegner mit fester Strategie. Das

Immunsystem wurde durch folgendes, sehr einfaches Regelwerk repräsentiert. Wenn eine Einheit des Gegners eine des Immunsystems schlug, erzeugte das Immunsystem eine neue mit zufällig gewähltem Typ. Im umgekehrten Fall ersetzte es eine zufällig gewählte mit einer des Einheitentyps, der gewonnen hatte. Experimente mit diesem Spiel über 1000 Runden zeigten zwar starke Schwankungen in den Anteilen der Einheitentypen des Immunsystems, jedoch entsprach das durchschnittliche Verhältnis von Einheitentypen der optimalen Strategie gegen die des Gegners.

MacDonald und Fyfe [19] ließen in diesem Spiel außerdem zwei künstliche Immunsysteme in Co-Evolution gegeneinander spielen. Zusätzlich änderten sie die Art, wie Einheiten nach Duellen ersetzt wurden. Es wurden Wahrscheinlichkeitsvektoren eingeführt, die für jede Startposition der Einheiten die Wahrscheinlichkeit für jeden Typ, gezogen zu werden, angaben. Dadurch wurde es möglich, eine Lernrate einzuführen, mit der der Vektor nach jedem Duell angepasst wurde. Die Lernrate wurde für jeden Spieler konstant gehalten. Im Experiment der Autoren gewann ein Immunsystem mit einer fünfmal so hohen Lernrate wie der des Gegners 70% der Spiele.

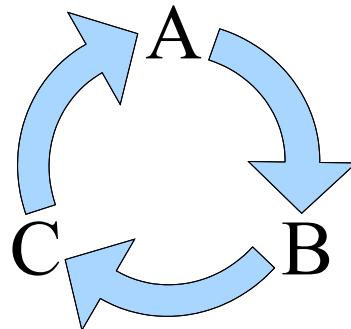


Abbildung 10. Ritter (B) schlägt Fußsoldat (A), Bogenschütze (C) schlägt Ritter (B) und Fußsoldat (A) schlägt Bogenschütze (C). Die Konstellation führt zu einem Wettrüsten der beiden Gegner, das im Kreis verläuft. Der schneller lernende Spieler gewinnt dabei langfristig die Oberhand.

Das genannte Spielprinzip ist zu einfach, um alle oben genannten Aspekte des Immunsystems beinhalten zu können. Zum Beispiel findet keine Mutation der Antikörper statt. Leen und Fyfe [3] untersuchten daher die Einsatzmöglichkeiten eines künstlichen Immunsystems in weiteren Spielen, die allerdings auch alle sehr theoretischer Natur waren. Da die Ergebnisse alle durchweg positiv waren, stellt sich allerdings die Frage, ob die Spiele überhaupt geeignet waren, die Leistungsfähigkeit von künstlichen Immunsystemen bezüglich der Intelligenz zu bewerten. Einige der in Abschnitt 1 genannten Anforderungen, wie zum Beispiel Ressourcenmanagement, stellten sich in den abstrakten Experimenten nicht. Da, wie man in Abschnitt 6.2 sieht, andere Autoren auf ganz ähnliche Ideen ge-

kommen sind, ist auch der Mehrwert des natürlichen Vorbilds nicht zwingend einleuchtend.

7 Zusammenfassung

Der Forschungszweig der künstlichen Intelligenz in Echtzeitstrategiespielen ist noch recht jung [1]. Das Gebiet der Computational Intelligence scheint bei den Forschern ein beliebtes Reservoir für neue Ideen zu sein. Da die hohe Komplexität des Genre offenbar hohe Anforderungen stellt, behandeln viele Ansätze nur Teilespekte der Problemstellung. Vor allem der Gebäudebau wurde häufig ausgeklammert. Die auf dynamischem Scripting basierende KI bewältigt von den vorgestellten Verfahren eines der kompliziertesten Spiele. Diese ist gleichzeitig den in kommerziellen Spielen vorherrschenden, regelbasierten Systemen am nächsten. Sie benutzt auch das meiste Expertenwissen, was sich in Abschnitt 4.2 als potenziell schädlich für den Erfolg und den Spielspaß herausgestellt hat.

Als Lernverfahren wurden TDL und Co-Evolution erwähnt. Während TDL nur in einem Fall angewendet wurde, wurde Co-Evolution bzw. Evolution benutzt, um Lernen in Kombination mit Influence Maps, dynamischem Scripting, neuronalen Netzen und künstlichen Immunsystems durchzuführen. Es stellt sich die Frage, ob diese Präferenz an einer geringeren Leistung von TDL liegt, oder ob Co-Evolution intuitiver zu implementieren oder schlicht bekannter ist.

Interessanterweise existiert eine Vielzahl von sehr unterschiedlichen Darstellungsformen des Spiels. Welche davon am besten den Charakter eines Echtzeitstrategiespiels abbildet und ob eine der genannten Techniken alleine für eine KI in einem komplexeren Spiel mit Gebäudebau ausreicht, lässt sich anhand der gegebenen Literatur vermutlich noch nicht sagen. Der Gebäudebau verkompliziert das Spiel vor allem durch das Hinzufügen von Abhängigkeiten, die das Planen erschweren. Dem kann man zum Beispiel begegnen, indem man die Abhängigkeiten direkt in einem Graph modelliert, oder indem man das Spiel in Zustände aufteilt, innerhalb derer keine Abhängigkeiten existieren.

Angesichts des frühen Stadiums der Forschung bleiben Echtzeitstrategiespiele voraussichtlich noch für viele Jahre eine Herausforderung für Entwickler von künstlicher Intelligenz. Ein Gradmesser des Erfolgs wird sicherlich zunächst sein, in wie weit kommerzielle Spielehersteller neue Techniken in ihre Produkte übernehmen. In ferner Zukunft werden KI-Methoden, die wir heute aus Computerspielen kennen, vermutlich auch in der realen Welt beim Militär zum Einsatz kommen, sollten sie sich als erfolgsversprechend erweisen.

Literatur

1. Buro, M.: Real-time strategy games: A new ai research challenge. In Gottlob, G., Walsh, T., eds.: IJCAI, Morgan Kaufmann (2003) 1534–1535
2. The Glest Team: Glest <http://www.glest.org>, 20.09.2007.
3. Leen, G., Fyfe, C.: Agent wars with artificial immune systems. In Rauterberg, M., ed.: Entertainment Computing - ICEC 2004, Third International Conference,

- Eindhoven, The Netherlands, September 1-3, 2004, Proceedings. Volume 3166 of Lecture Notes in Computer Science., Springer (2004) 420–428
4. Livingstone, D.: Coevolution in hierarchical ai for strategy games. In Kendall, G., Lucas, S., eds.: Proceedings of the 2005 IEEE Symposium on Computational Intelligence and Games (CIG05), Essex University, Colchester, Essex, UK, 4-6 April, 2005, IEEE (2005)
 5. Chung, M., Buro, M., Schaeffer, J.: Monte carlo planning in rts games. In Kendall, G., Lucas, S., eds.: Proceedings of the 2005 IEEE Symposium on Computational Intelligence and Games (CIG05), Essex University, Colchester, Essex, UK, 4-6 April, 2005, IEEE (2005)
 6. Miles, C., Quiroz, J., Leigh, R., Louis, S.J.: Co-evolving influence map tree based strategy game players. In: CIG, IEEE (2007)
 7. Wintermute, S., Xu, J., Laird, J.E.: Sorts: A human-level approach to real-time strategy ai. In: Proceedings of the Third Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-07). (2007)
 8. Buro, M.: Orts: A hack-free rts game environment. In Schaeffer, J., Müller, M., Björnsson, Y., eds.: Computers and Games. Volume 2883 of Lecture Notes in Computer Science., Springer (2002) 280–291
 9. Ponsen, M., Spronck, P.: Improving adaptive game ai with evolutionary learning. In: Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004), University of Wolverhampton (2004) 389–396
 10. Ponsen, M.J.V., Muñoz-Avila, H., Spronck, P., Aha, D.W.: Automatically acquiring domain knowledge for adaptive game ai using evolutionary learning. In Veloso, M.M., Kambhampati, S., eds.: AAAI, AAAI Press / The MIT Press (2005) 1535–1540
 11. Van Valen, L.: A new evolutionary law. *Evolutionary Theory* **1** (1973) 1–30
 12. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA (1998)
 13. Kerbusch, P.J.M.: Learning unit values in wargus using temporal differences. Master's thesis, Universiteit Maastricht (2005) http://www.cs.unimaas.nl/~uiterwyk/Theses/BSc/Kerbusch_BSc-paper.pdf, 29.07.2007.
 14. Bourg, D.M., Seemann, G.: AI for Game Developers. O'Reilly Media, Inc. (2004)
 15. Stanley, K.O., Bryant, B.D., Miikkulainen, R.: Real-time evolution in the nero video game (winner of cig 2005 best paper award). In Kendall, G., Lucas, S., eds.: Proceedings of the 2005 IEEE Symposium on Computational Intelligence and Games (CIG05), Essex University, Colchester, Essex, UK, 4-6 April, 2005, IEEE (2005)
 16. Louis, S.J., Miles, C.: Case-injection improves response time for a real-time strategy game. In Kendall, G., Lucas, S., eds.: Proceedings of the 2005 IEEE Symposium on Computational Intelligence and Games (CIG05), Essex University, Colchester, Essex, UK, 4-6 April, 2005, IEEE (2005)
 17. Louis, S.J., McDonnell, J.: Learning with case-injected genetic algorithms. *IEEE Trans. Evolutionary Computation* **8**(4) (2004) 316–328
 18. Fyfe, C.: Dynamic strategy creation and selection using artificial immune systems. *Int. J. Intell. Games & Simulation* **3**(1) (2004) 1–6
 19. MacDonald, D., Fyfe, C.: Strategy selection in games using co-evolution between artificial immune systems. In: Entertainment Computing - ICEC 2004, Third International Conference, Eindhoven, The Netherlands, September 1-3, 2004, Proceedings. Volume 3166 of Lecture Notes in Computer Science. (2004) 445–450

Computational Intelligence in Rollenspielen

Seminar: CI in Computerspielen

Dominik Opolony

¹ TU Dortmund

² Lehrstuhl 11

³ dominik.opolony@uni-dortmund.de

Zusammenfassung. Diese Ausarbeitung im Rahmen des Seminars "Computational Intelligence in Computerspielen" wird im Folgenden eine ausführliche Einleitung in Rollenspiele gegeben, um danach einige Einsatzgebiete anzureißen. Hierauf folgend wird ein dynamisches Rufsystem als Beispieldarstellung vorgestellt, während die nachfolgende AI-Skalierungsmethode Möglichkeiten aufzeigt, um Intelligenz handhabbar zu halten. Am Ende wird noch kurz eine neue Entwicklung vorgestellt, welche noch nicht besonders ausgereift ist, aber viel Potenzial hat.

1 Was sind Rollenspiele ?

In Rollenspielen kurz RPG (Role Playing Game) dreht sich alles darum, dass der Spieler ein oder mehrere Avatare steuert, welche sich in der Spielwelt mehr oder minder frei bewegen können. Ziel des Spiels ist es, die komplette Geschichte des Spiels, welche erzählt werden soll, nachzuspielen. Die Geschichte der Rollenspiele fängt in den 70er Jahren an den Universitäten an, welche als einzige Zugang zu Rechnern ermöglichten. Inspiriert wurde das Ganze von sogenannten Pen & Paper Rollenspielen.

Folgende Meilensteine im Bereich der Computerrollenspiele gab es:

- dnd (1974) - ältestes erhaltenes Computerrollenspiel
- Dungeon (1975) - erstes Spiel mit grafischer Darstellung
- Ultima (1981) - Vorbild vieler späterer Rollenspiele
- Wizardy (1981) - Vorbild vieler späterer Rollenspiele
- Bards Tale (1985) - grafischer Standart in der Umgebungsdarstellung
- Dungeon Master (1987) - Einführung von Echtzeitelementen und Umgebungsinteraktion
- Pools of Radiance (1988) - Tabletopartige Darstellung von Kämpfen
- Neverwinter Nights (1991) - erstes Multiplayer Online Roleplaying Game
- Ultima Underworld (1992) erstes echtes 3D RPG
- Diablo (1996) - Auslöser des Actionspielebooms
- Baldur's Gate (1998) - Vorbild für Storytelling RPGs in moderner Präsentation
- Planescape: Torment (1999) - Computerrollenspiel als Kunstform

1.0.1 Der Avatar Zu Beginn eines RPGs steht der Spieler in der Regel vor der Wahl, wie sein Avatar aussehen wird. Das heißt, dass der Spieler über Größe, Rasse, Klasse, Aussehen wie z.B. Gesichtstextur, Schmuck, Kleidung, Hautfarbe, Haarfarbe, als auch oftmals eine Spezialisierung des Charakters auf bestimmte Gebiete wie z.B. Schmied, Kämpfer, Händler etc., entscheiden muss. Dieser Avatar ist für das gesamte Spiel der vom Spieler kontrollierte Charakter, ein sogenannter Player Charakter (PC). Jeder PC hat verschiedene Fähigkeiten, welche nicht nur kämpferischer Natur sind, sondern auch zum Herstellen von Waffen, der Alchemie und anderen Berufen genutzt werden können. Dabei werden jegliche Aktionen inklusive der Bewegung dieses PC vom Spieler kontrolliert. Im Laufe der Zeit wird der Avatar stärker, in dem er eine sogenannte Stufe, eoder auch Level genannt, aufsteigt. Dies hat meistens zur Folge, dass der Spieler weitere Fähigkeitsopunkte auf Spezialisierungen verteilen kann, um so eine individuelle Entwicklung seines Avatars zu forcieren.

Eine weitere wichtige Entwicklungsart des PCs ist seine getragene Ausrüstung. Innerhalb der meisten Rollenspiele gibt es unzählige Gegenstände, die der PC tragen kann. Dabei ist auch hier mit fortschreitendem Spiel eine Entwicklung der Gegenstände, welche der Spieler bekommen kann, zu beobachten. So gibt es meist im Laufe des Spiels mächtigere Schwerter, stärkere Rüstungen, größere Taschen, bessere Heiltränke zu finden und zu kaufen.

1.0.2 NPCs Innerhalb der Spielwelt gibt es unzählige Nicht-Spieler-Charaktere, sogenannte NPCs. Das sind vom Computer gesteuerte Menschen und Monster, welche mit dem PC interagieren. Dies kann ein Handel, ein Gespräch oder ein Kampf sein. NPCs sind essentiell für das Spielgefühl und für die Lebendigkeit der Spielwelt.

1.0.3 Die Spielwelt Heutige Rollenspiele bieten dem Spieler eine 3D-Welt, in der er sich frei mit seinem PC bewegen kann. Frühere Spiele benutzten eine 2D/Pseude-3D Welt. Dabei sieht er seinen PC meist aus einer Gottesperspektive von oben bzw. von schräg oben bzw. einer Ich-Perspektive. Innerhalb der Spielwelt steht es dem Spieler weitestgehend frei, wohin er sich bewegt. Weitestgehend deshalb, weil der Spieler natürlich von verschlossenen Türen und geographischen Hindernissen blockiert wird. Innerhalb dieser Welt sind in Gebäuden, welche zu Städten werden, in der Wildnis, auf Wegen usw. NPCs plaziert, welche mit dem Spieler interagieren.

Weitere Interaktion gibt es meist nur mit sogenannten Points-Of-Interest. Dies sind Objekte, welche dem Spieler Interaktionsmöglichkeiten bieten. Oftmals kann nicht jeder Schrank, jede Truhe, jede Tür innerhalb der Spielwelt geöffnet werden, sondern nur solche, welche dafür vorgesehen sind und damit Points-Of-Interests sind. Offensichtlichste Form der Points-Of-Interest sind NPCs durch die Möglichkeit, den PC mit ihnen reden, kämpfen und handeln zu lassen.

1.0.4 Aufgaben und Auseinandersetzungen Der größte Teil eines Rollenspiels wird von sogenannten Quests eingenommen. Dies sind Aufgaben, die der

Spieler mit seinem PC erledigen kann, um Gold, Erfahrungspunkte, welche zum Aufstieg des PC in eine neue Stufe benötigt werden, und Ausrüstungsgegenstände zu erhalten. Solche Aufgaben sind meistens relativ einfacher Natur wie z.B. "Töte X Monster der Art Y", "Sammel X Gegenstände der Art Y", "Bringe diese Nachricht zu Person X", usw. Im Rahmen dieser Aufträge tritt meistens ein weiteres wichtiges Element von Rollenspielen in den Vordergrund.

Eine Auseinandersetzung, im Folgenden Kampf genannt, verlangt vom Spieler, seine verschiedenen Fähigkeiten, die entweder Schaden verursachen, den Gegner schwächen oder jemanden heilen, einzusetzen, um dem Gegner zu töten. Dabei läuft dies von Spiel zu Spiel verschieden ab. Es gibt rundenbasierte Kämpfe, aber auch Echtzeitkämpfe, in denen die Anforderungen gegenüber den rundenbasierten Kämpfen eher im Timing liegen.

1.0.5 Geschichte Das Ziel der meisten Rollenspiele ist es, dass der Spieler mit seinem PC die sogenannte Story, also die Geschichte, welche das Spiel erzählt, nachzuspielt. Dies wird durch Aufgaben, Auseinandersetzungen, durch das Aufsteigen des Spielercharakters in neue Stufen usw. realisiert. Im Laufe des Spiels dringt der Spieler immer tiefer in die Geschehnisse der Spielwelt ein, um irgendwann, wenn er mächtig genug ist, das Unheil an der Wurzel besiegen zu können und dem sogenannten Endgegner entgegenzutreten.

1.1 Einzel-Spieler Rollenspiele

Einzel-Spieler Rollenspiele zeichnen sich dadurch aus, dass der Spieler entweder einen PC oder eine Gruppe von PCs steuert. Hierbei spielt der Spieler jedoch immer alleine, so dass sämtliche Interaktion nur mit NPCs stattfindet.

In Einzel-Spieler Spielen ist die Geschichte oftmals der treibende Faktor und nimmt eine sehr hohe Priorität bei der Entwicklung ein, da hierdurch hoffentlich genug Spannung erzeugt wird, um den Spieler bis zum Ende an das Spiel zu fesseln. Oftmals basieren Rollenspiele, aber vor allem Einzel-Spieler Rollenspiele, auf schon erschienenen Büchern oder Filmen.

Bekannte Beispiele der letzten Jahre:

- Baldur's Gate von BioWare
- Neverwinter Nights von BioWare
- The Elder Scrolls: Oblivion von Bethesda Softworks

1.2 Mehr-Spieler-Rollenspiele

Massivly Multiplayer Online Role Playing Games (kurz MMORPG), also Rollenspiele, welche vor allem durch eine Vielzahl von oft mehreren tausend Spielern geprägt werden, sind die neueste und aufstrebendste Entwicklung auf dem Rollenspielmarkt. Diese Spiele zeichnen sich dadurch aus, dass ein Spieler einen PC in einer Spielwelt steuert, welche auf einem Server im Internet existiert. Somit ist auch klar, dass eine ständige Online-Verbindung existieren muss, um spielen

zu können. Dieser Aspekt hat dazu geführt, dass erst mit der Verbreitung von Breitband-Internetzugängen mit einer festen Preispauschale ein Markt für solche Spiele entstanden ist. Weiterhin ist es üblich, dass eine monatliche Pauschale für die Nutzung des Angebots, für ständige Spielwelterweiterungen, für Updates und für Serverkosten zu entrichten ist.

Innerhalb dieser Spielwelt spielen mehrere tausend Spieler gleichzeitig. D.h. sie können ihre Avatare sehen, interagieren, reisen, sich in Gruppen organisieren etc. Die Kommunikation zwischen den Spielern ist hierbei oftmals in sogenannte Chat-Kanäle gezwängt, wobei auch bei einigen Spielen eine Voice-Chat Unterstützung zu finden ist. Sowohl für Chat- als auch für Voice-Chat-Kanäle gibt es Einschränkungen und Freiheiten für die Benutzer, so dass eine private Kommunikation über private Chat-/Voice-Kanäle ermöglicht wird.

Das Spiel lässt sich hierbei grob in zwei Spielarten aufteilen, für die sich der Nutzer nicht entscheiden muss, aber für die er sich gemäß seinen Interessen entscheiden kann. Der erste Bereich ist der PVE Bereich. PVE steht hierbei für Player vs. Environment. Dabei geht es um genau das, was in einem Einzel-Spieler-Rollenspiel nur möglich ist. Der Spieler kämpft alleine oder mit mehreren Spielern zusammen gegen einen vom Computer gesteuerten Gegner, also einen NPC. Als zweiten Bereich, der exklusiv den Mehr-Spieler-Rollenspielen vorenthalten ist, wäre das PVP zu nennen. PVP steht hierbei für Player vs. Player. Es geht darum, dass zwei Gruppen von Spielern, oder auch zwei Spieler alleine, gegeneinander antreten, um den oder die Anderen zu besiegen.

Ein Beispiel für ein solches MMORPG ist World of Warcraft von Blizzard Entertainment, dass weltweit über 9 Millionen Abonnenten hat und im Jahr 2004 erschienen ist. In Abbildung 1 auf der nächsten Seite ist oben links der grüne Lebensbalken und der blaue Manabalken des PC zu sehen, während sich daneben der Lebensbalken des NPCs befindet, der im Moment bekämpft wird. In der Mitte des Bildschirms ist der Avatar zu sehen, während unterhalb und zur rechten des Avatars die verschiedensten Fähigkeiten angeordnet sind, die ihm zur Verfügung stehen. Unten links ist das Chatsystem zu sehen und oben rechts ist die sogenannte Minimap zu finden, eine kleine Karte über die nähere Umgebung des Spielers.

2 Einsatzfelder von CI/AI in Rollenspielen

Im Folgenden sollen die Einsatzfelder von CI/AI in Rollenspielen anhand von Konzepten beschrieben werden, wie sie in vielen Spielen dieser Art zu finden sind. Nachfolgend erläutert ein Beispiel, welche Methoden in World of Warcraft eingesetzt werden.

2.1 Einsatzfelder

- Wegfindung

In allen RPGs wird ein Gelände eingesetzt, über das sich PCs und NPCs



Abb. 1. Ausschnitt aus der Spielwelt von World of Warcraft

bewegen müssen. Ob dies nun ein 2D oder 3D Gelände ist, spielt dabei keine Rolle. PCs und NPCs müssen Bewegungsbefehle ausführen und somit den schnellsten und optimalsten Weg finden. Ob dies in Kämpfen durch den Computer passiert oder auf Anweisung des Spielers ist nicht wichtig. Die Wegfindungsroutine muss vorhanden sein.

– Kampf

Wie verhalten sich NPCs im Kampf ? Welche Fähigkeiten setzen sie ein ? Wann setzen sie diese ein ? Wie intelligent sollen sie kämpfen ? Welches Ziel greifen sie an, wenn es mehrere gibt ? Für alle von diesen Fragen muss es Lösungen geben.

– NPCs

Jeder NPC ist ein eigenständiger Agent, welcher reden, kämpfen, laufen usw. muss, ohne dass der Spieler ihn steuert. Da die Spielwelt oft aus mehreren tausend Agenten besteht, muss jeder von diesen diese Aktionen ausführen können. Dabei kommen natürlich auch Effizienzfragen auf.

– Ruf

Wie verhalten sich NPCs gegenüber dem Spieler ? Wie verändert sich dieses Verhalten ? Kann der Spieler das Verhalten durch bestimmte Ereignisse steuern ? Auch diese Verhaltensweisen müssen gelöst werden, um eine "lebendigere" Welt zu erschaffen.

2.2 Beispiel: World of Warcraft

In World of Warcraft ist es Spielern möglich, innerhalb einer Gruppe Aufgaben zu lösen und NPCs zu bekämpfen. Die größte Gruppe, ein sogenannter Raid, besteht aus 10-40 Spielern, welche durch das Zusammenarbeiten der verschiedenen PCs und deren verschiedene Fähigkeiten den sogenannten Boss, einen besonders starken NPC, zu besiegen versuchen. Hierbei werden erhöhte Anforderungen an Kommunikation und Koordination gestellt, ohne die es nicht möglich ist, erfolgreich zu sein.

Motiviert werden die Spieler hierbei durch besonders mächtige Belohnungen wie Rüstungen usw., welche nur in solchen Großgruppen erlangt werden können.

Das Verhalten dieser Bosse basiert hierbei auf Skripts, welche in der Sprache LUA verfasst sind. Das hat zur Folge, dass Spieler das Verhalten vorhersagen können und somit durch das Lernen des Verhaltens der Bosse geeignete Gegenmaßnahmen einleiten können. Insgesamt führt dies dazu, dass immer neue und schwierigere Kämpfe implementiert werden müssen, da sonst der Unterhaltungswert leiden würde.

3 Beispiel AI: Dynamisches Rufsystem

Eine große Aufgabe bei der Entwicklung eines Spiels mit vielen Nicht-Spieler-Charakteren (NPCs) ist es, diesen Leben einzuhauen. Ein gutes Rufsystem verwaltet den Ruf des Spielers bei NPCs dynamisch und auf den Aktionen des Spielers basierend. Das Rufsystem von Ultima Online basierte z.B. darauf, dass der Spieler beim Töten eines NPCs Ruf dazugewinnt oder abgezogen bekommen und dies den Ruf bei allen NPCs verändert hat. Das heißt, der Effekt war global und direkt aktiv. Spiele, in denen dieser globale Effekt auftritt, verhindern so, dass der Spieler seinem Ruf vorrausseilen bzw. seine Aktionen verborgen kann. Basierend auf einem Kapitel von Greg Alt und Kristin King namens "A Dynamic Reputation System Based on Event Knowledge" aus AI Game Programming Wisdom [4] wird im Folgenden ein dynamisches Rufsystem beschrieben, dass den globalen Effekt verhindert und Veränderungen der Einstellung gegenüber dem Spieler nur aufgrund von direktem oder indirektem Wissen von Aktionen des Spielers herbeiführt. Gleichzeitig soll der benötigte Speicherplatz und die CPU Zeit minimiert werden. Die Idee ist, dass NPCs ihre Meinung nur ändern, wenn sie eine Aktion gesehen haben oder von ihr erzählt bekommen. Auf diese Weise, also per Kommunikation, werden Informationen in andere Teile der Spielwelt transportiert. Allerdings hat der Spieler damit die Möglichkeit, seinem Ruf vorrauszuzeigen bzw. die Informationsverteilung zu verhindern bzw. beeinflussen.

3.1 Datenstrukturen

Die wichtigsten Datenstrukturen sind das Reputation Event, also ein Rufergebnis, ein Per-NPC-Long Time Memory, also ein Langzeitgedächtnis, und eine Per-NPC-Reputation Table, also eine Ruftabelle pro NPC.

Im Abbildung 2 ist zu sehen, dass sobald ein Reputation Event erzeugt wird, dies zur Master-Event-List hinzugefügt wird. Dies ist eine kompakte Verwaltung der Ereignisse. Weiterhin werden alle Teilnehmer benachrichtigt. In der Master-Event-List werden also alle Ereignisse gespeichert, welche von NPCs beobachtet wurden.

Weiterhin besitzt jeder NPC ein Langzeitgedächtnis, welches eine Liste mit allen Ereignissen ist, von denen der NPC durch Beobachtung oder Erzählung Kenntnis hat.

Als Drittes hat jeder NPC eine Reputation Table, in der er den aktuellen Ruf des Spielers ablesen kann bzw. dieser markiert ist.

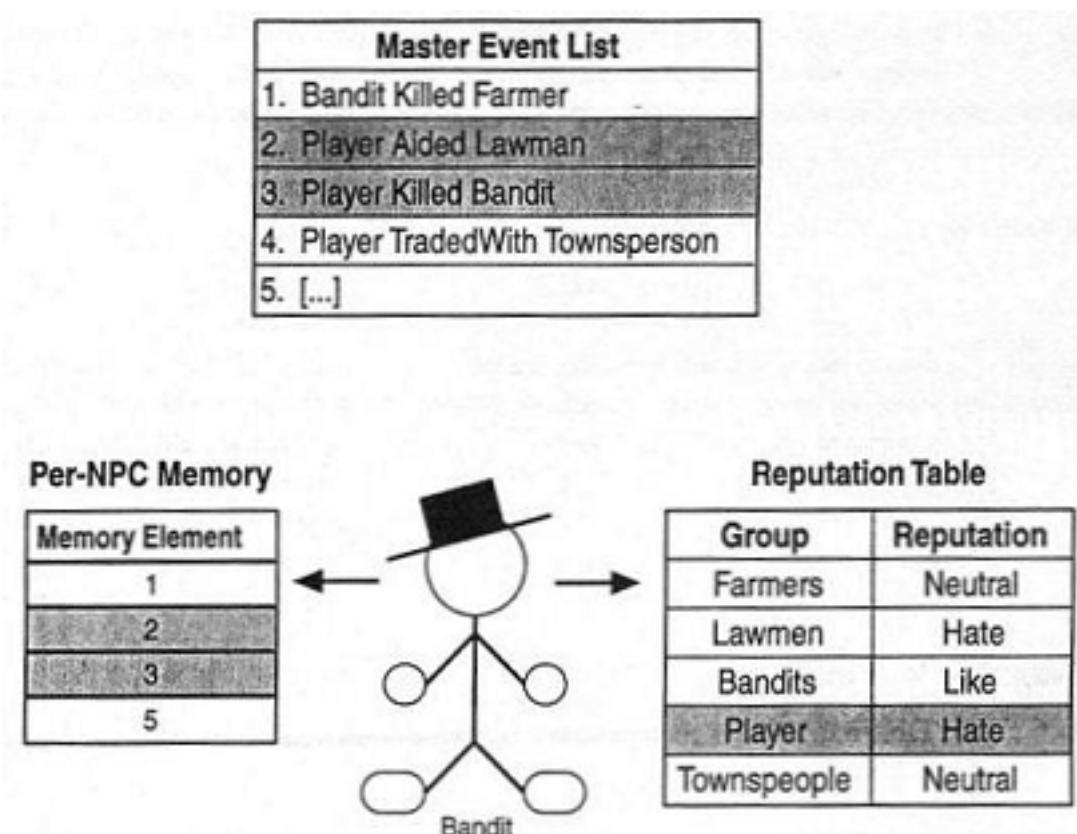


Abb. 2. Der NPC hasst den Spieler, weil er herausgefunden hat, dass der Spieler einen Banditen getötet hat und einen anderen verwundet hat

3.1.1 Reputation Event Das Reputation Event enthält folgende Informationen. Als erstes wird die Person in der Subject Group beschrieben, die das Event ausgelöst hat. Danach geht es mit der Aktion weiter, die das Event ausgelöst hat, dem Verb. Object Group und Object Individual beschreiben die Gruppe und das Individuum innerhalb der Gruppe, welche das Ereignis betreffen. Magnitude enthält den Zahlenwert der Rufänderung. Weiter geht es mit dem "Wo?" und "Wann?", der benutzten Vorlage sowie dem Reference-Count, welcher angibt, wieviele NPCs über dieses Ereignis Kenntnis haben. Am Ende stehen dort die Reputation Effects, eine Zusammenfassung, was passiert, wenn dieses Ereignis auftritt.

Subject Group	Player
Verb	DidViolenceTo
Object Group	Bandit
Object Individual	Joe
Magnitude	75 (Killed)
Where:	50, 20, 138 (In front of saloon)
When:	High noon
Template	KilledBanditTemplate
Reference Count	Known by 11 NPCs
Reputation Effects	Bandits hate player more Lawmen like player more Farmers like player more

Abb. 3. Beispiel für ein Reputation Event

3.1.2 Master Event List Ein Weg, um Ereignisse zu speichern wäre es, wenn jeder NPC jedes Ereignis speichern würde, über das er Kenntnis hätte. Dies wäre aber eine Verschwendug von Speicherplatz und würde schnell unübersichtlich werden. Um dies zu verhindern, speichert das dynamische Rufsystem alle Ereignisse an einem Ort, der so genannten Master-Event-List, welche alle Reputation Events beinhaltet, wobei diese nach ihrem Auftreten, der Event ID, sortiert sind. Die Liste wird durch einige Maßnahmen klein gehalten. Jedes Ereignis, dass in die Liste aufgenommen wird, hat einen Reference Count, der angibt, wieviele NPCs über dieses Ereignis Bescheid wissen. Sobald dieser Zähler auf 0 steht, wird das entsprechende Ereignis aus der Liste entfernt. Weiter werden nur Ereignisse gespeichert, bei denen der Spieler beteiligt ist oder bei denen ein NPC jemanden getötet hat. Dies reicht aus, da Reputation Events keinerlei Einfluss auf das Verhalten von NPCs untereinander haben.

Letzte Maßnahme, um die Liste möglichst klein zu halten ist, dass redundante Informationen nicht gespeichert werden. Dies bedeutet, dass wenn ein "Schießen auf"-Ereignis und ein "Töten"-Ereignis auftreten und diese den gleichen NPC betreffen, wird nur das "Töten"-Ereignis gespeichert.

3.1.3 Per-NPC-Long Time Memory Im NPC-Long-Term-Memory speichert jeder NPC eine komprimierte Version der Reputation Events, über die er Kenntnis hat. Dabei wird nur die Ereignis ID, der Wert des Ereignisses und der Zeitpunkt gespeichert, an dem er über dieses Ereignis Kenntnis erlangt hat. Dabei wird vom Langzeitgedächtnis nicht zwangsläufig der höchste Wert eines Ereignisses gespeichert. Beispielsweise ist eine Situation vorstellbar, in der zwei NPCs einen Kampf zwischen einem Spieler und einem dritten NPC beobachten. Ein NPC verlässt nun aber den Ort des Geschehens, bevor der Spieler den dritten NPC getötet hat. Nun passiert folgendes: Beide NPCs, welche dieses Ereignis beobachtet haben verweisen auf das selbe Ereignis in der Master Event List. Allerdings speichert der NPC, welcher gesehen hat, dass der dritte NPC getötet wurde dies mit dem Wert von z.B. 75 und der NPC, welcher nur den Kampf beobachtet hat aber nicht dessen Ausgang das Ereignis mit dem Wert von 25. Die obigen Zahlen sind natürlich nur zur Verdeutlichung und sollen eine unterschiedliche Gewichtung von "Töten" und "Auf jemanden Schießen" ausdrücken.

Um Speicher zu sparen ist es natürlich ratsam, die Tabellen, welche das Langzeitgedächtnis darstellen, klein zu halten. Ein Weg, Reputation Events nicht selber zu speichern, sondern ihre ID anstatt, wurde vorher erklärt. Eine weitere Methode ist sicherzustellen, dass NPCs niemals Kenntnis über ein Ereignis haben, dass keinerlei Auswirkungen auf ihr Verhalten hat. Sie sollten z.B. niemals über ein Ereignis Bescheid wissen, bei dem der Spieler einen Banditen getötet hat, dass aber keiner gesehen hat. Die nächste Methode ist, NPCs Ereignisse vergessen zu lassen. Nach einer bestimmten, vom Designer voreingestellten Zeit vergessen NPCs Ereignisse, was zur Folge hat, dass irgendwann der Reference Count im Ereignis auf 0 gesetzt wird und somit eine Entfernung dieses Ereignisses vorgenommen wird.

Letztendlich gibt es noch die Möglichkeit, Punkte in der Spielerentwicklung einzuführen, von denen es kein zurück mehr gibt. Das bedeutet, dass an Punkten wie z.B. einem bestimmten Level des Spielercharakters, einige Ereignisse überflüssig werden, da es ab diesem Punkt vorgesehen ist, dass der Spieler z.B. einen schlechten Ruf bei den Banditen hat. Somit können alle Ereignisse entfernt werden, die in diese Richtung gehen würden.

3.1.4 Informationsgewinnung von NPCs NPCs haben drei Möglichkeiten, Kenntnis von Ereignissen zu erhalten. Entweder sehen sie Ereignisse, hören von Ereignissen durch andere NPCs oder folgern Ereignisse aus bestehendem Wissen. Jedesmal, wenn ein NPC ein Ereignisse kennenlernt, muss entschieden werden, was mit diesem Ereignis passiert.

3.1.4.1 Event-Announcer Ereignis-Ansager oder Event-Announcer entstehen, sobald ein Ereignis in der Spielwelt geschieht, zu dem es einen Ereignistyp gibt. Der Event-Announcer zu diesem Ereignis entsteht an genau der Stelle, an der das Ereignis stattgefunden hat bzw. immer noch stattfindet. Dabei gibt es zwei Arten von Event-Announcern:

Als Erstes gibt es die einmaligen Event-Announcer, welche genau einmal ihre Nachricht verkünden. Dies kann z.B. ein Handel sein, ein Diebstahl usw.

Als Zweites gibt es die sogenannten dauerhaften Event-Announcer. Diese verkünden ihre Nachricht dauerhaft. Die kann z.B. ein Mord sein, bei dem ein dauerhafter Event-Announcer entsteht, der einmal das komplette Ereignis verkündet und danach nur noch ein Teilereignis propagiert, in welchem der Mörder als unbekannt angegeben wird. Dies ist nötig, da nach einem Mord die Leiche des getöteten NPCs an Ort und Stelle verbleibt und somit jedem der dies beobachtet, gewisse Informationen bereitstellt.

Sobald ein Event-Announcer erscheint überprüft dieser, ob sich Personen in seiner Reichweite befinden und sendet ihnen bei Bedarf, wobei dieser für jeden NPC basierend auf seinem Langzeigedächtnis individuell ist und bestimmt werden muss, ein Reputation Event. Dies hat auch zur Folge, dass zu einem Ereignis, welches niemand sieht, kein Reputation Event erzeugt wird.

3.1.4.2 Teilen von Informationen Eine weitere Möglichkeit der Informationsgewinnung ist der Austausch von Informationen zwischen NPCs.

Wenn sich zwei Nicht-Spieler-Charaktere in der Spielwelt treffen, wird überprüft, ob sich die beiden soweit mögen, dass sie Informationen austauschen können. Beispielsweise wird überprüft, ob die Banditen den Händlern feindlich gesinnt sind. Wenn nicht, tauschen sie Informationen aus.

Falls ein Informationsaustausch stattfindet, werden alle Informationen der beiden NPCs ausgetauscht, wobei aber nur nicht redundante Ereignisse gespeichert werden.

3.1.4.3 Folgern von Informationen Eine dritte Möglichkeit das NPCs neues Wissen erlangen, ist das einfache Folgern von Informationen. Wenn ein NPC Kenntnis über ein Ereignis mit einem unbekannten Auslöser hat und später über ein passendes Ereignis erfährt, wird nur das detailliertere Ereignis gespeichert, wobei der Wert aber der Größere von beiden Ereignissen ist.

Beispielsweise sieht ein NPCs eine Leiche und bekommt so vom dauerhaften Event-Announcer das Ereignis mit einem unbekannten Täter. Später teilt dieser NPC dann Informationen mit einem anderen NPC aus, der gesehen hat, dass der Spieler auf den selben Banditen geschossen hat. Beide NPCs folgern nun, dass der Spieler den Banditen getötet hat, auch wenn dies ein unsicheres Schließen ist. Allerdings ist unsichere Schließen sehr menschlich und somit gewollt.

3.1.5 Update Prozess Sobald ein NPC die Kenntnis über ein Ereignis erlangt, wird der Update Prozess ausgeführt. Dieser soll sicherstellen, dass nur relevante Ereignisse aufgenommen werden und diese auch mit dem passenden Wert gewichtet werden. Andere Ereignisse müssen nur aktualisiert werden.

Die Suche nach einem Ereignis, welches aktualisiert werden muss, läuft folgendermaßen ab:

1. Die Subjekt Gruppe, das Verb, die Objekt Gruppe und die Objekt ID müssen übereinstimmen.

2. Die Subjekt Gruppe, das Verb und die Objekt Gruppe müssen gleich sein, aber die Objekt ID kann variieren.
3. Die Subjekt Gruppe des neuen Ereignisses, auf das hin überprüft wird, ist unbekannt, so dass diese alle Werte annehmen kann. Das Verb, die Objekt Gruppe und die Objekt ID müssen allerdings stimmig sein. Alternativ dazu kann auch die Subjekt Gruppe des neuen Ereignisses bekannt sein und die Subjekt Gruppe des existierenden Ereignisses unbekannt, wobei aber das Verb, die Objekt Gruppe und die Objekt ID wieder stimmen müssen.

Wenn eine Übereinstimmung gefunden wird, versucht der Updateprozess nach bestimmten Regeln das Ereignis zu aktualisieren. Falls dies nicht funktioniert, wird nach weiteren Übereinstimmungen mit den ersten vier Punkten gesucht. Dies geschieht so lange, bis ein erfolgreiches Update durchgeführt wird oder aber keine Übereinstimmungen mehr gefunden werden. Dann handelt es sich um ein Neues und wird in das Langzeitgedächtnis aufgenommen.

Der Updateprozess versucht nach sieben Regeln, ein Update eines Ereignisses herbeizuführen.

1. Wenn der Wert des neuen Ereignisses kleiner oder gleich ist und die Subjekt Gruppe entweder unbekannt oder gleich ist und die Objekt ID auch gleich ist, wird das neue Ereignis aufgrund von Redundanz ignoriert.
2. Falls die gleichen Annahmen wie in Punkt 1 gelten mit dem Unterschied, dass der Wert des neuen Ereignisses größer ist, wird der Wert im Langzeitgedächtnis aktualisiert.
3. Falls das neue Ereignis eine bekannte Subjekt Gruppe hat und das Existierende nicht, besitzt das neue Ereignis aktuellere bzw. präzisere Informationen. Dies gilt natürlich nur, falls die Objekt ID übereinstimmt. Der NPC entfernt somit das alte Ereignis und fügt das neue Ereignis in sein Langzeitgedächtnis ein.
4. Wenn beide oder eine Subjekt Gruppen unbekannt sind und die Objekt IDs nicht übereinstimmen, wurde keine Übereinstimmung gefunden. Das neue Ereignis wird ins Langzeitgedächtnis eingefügt.
5. Falls die Subjekt Gruppen übereinstimmen aber das alte Ereignis den höheren Wert hat, ist das neue Ereignis irrelevant.
6. Wenn die Subjekt Gruppen bekannt und gleich sind und das neue Ereignis den maximalen Wert besitzt, wird das existierende Ereignis aktualisiert und es werden alle anderen Übereinstimmungen gelöscht, da diese nun redundant sind.
7. Falls die Subjekt Gruppen übereinstimmen bei verschiedener Objekt ID und kein Ereignis einen größeren Wert hat, wird das neue Ereignis ins Langzeitgedächtnis eingefügt.

3.1.5.1 Update der Master Event List Falls ein NPC ein neues Ereignis hinzufügt oder ein bestehendes Ereignis in seinem Langzeitgedächtnis aktualisiert, wird gleichermaßen die Master Event Liste benachrichtigt. Wenn das neue Ereignis keinem Ereignis in der Master Event Liste entspricht, wird es mit einem

Reference Count von 1 hinzugefügt. Falls es mit einem existierenden Ereignis übereinstimmt, wird das existierende Ereignis aktualisiert und der Reference Count um einen erhöht.

3.1.6 Per-NPC Reputation Table Jeder NPC benutzt die in seinem Langzeitgedächtnis gespeicherten Informationen, um seine Meinung von anderen Gruppierungen und dem Spieler herzustellen. Somit speichert jeder NPC eine Per-NPC Reputation Table, welche seine aktuelle Einstellung gegenüber dem Spieler und anderen Gruppierungen im Spiel enthält.

Die Tabelle wird als dynamisches Feld gespeichert, wobei jeder Eintrag ein Zähler für jede Gruppierung im Spiel und dem Spieler ist. Jeder Zähler besteht aus zwei Fließkommazahlen, wobei eine davon die positiven Effekte zählt, während die andere die Negativen kummuliert. Dies kann dann verschiedenste Auswirkungen haben:

1. Wenn "Positiv" höher ist als "Negativ", mag der NPC den Spieler bzw. die Gruppierung. Dies kann Relevanz für den Austausch von Informationen haben.
2. Wenn "Positiv" und "Negativ" beide eher klein sind, weiß der NPC nicht, was er denken soll.
3. Falls "Positiv" und "Negativ" beide eher hoch sind, hat der NPC eine starke, aber zwiegespaltene Einstellung gegenüber dem Spieler oder der Gruppe. Der NPC wird den Spieler oder die Gruppe fürchten und ihr misstrauen.

Sobald ein NPC auf den Spieler oder einen anderen NPC trifft, schaut er in seiner Reputation Table nach, wie er sich dem Getroffenen gegenüber verhalten soll. Somit würde ein NPC niemals jemanden angreifen, den er mag, während er sich jemandem, bei dem er sich nicht sicher ist, vorsichtig nähern wird.

3.2 Zusammenfassung

Das dynamische und ereignisbasierte Rufsystem ermöglicht es, eine große Zahl von menschenähnlichen NPCs zu modellieren und dabei die CPU- und Speicher Kosten minimal zu halten. Hier wird ein relativ einfaches Rufsystem beschrieben, welches aber, sobald dieses implementiert ist, leicht durch die Einführung neuer Verben wie z.B. "lügen", "handeln", "schenken", "stehlen", "zerstören" usw. erweitert werden kann. Dabei ist es auch denkbar, Ereignisse nicht nur auf NPCs wirken zu lassen, sondern auch auf Objekte wie z.B. Gebäude. Bei alledem ist es nur wichtig, den Speicherverbrauch im Auge zu behalten.

4 Schwierigkeiten beim Einsatz von AI: Level-of-Detail

Im Folgenden soll ein Ansatz von BioWare beschrieben werden, bei dem es sich um eine AI-Skalierung für große Rollenspiele handelt. Dabei bezieht sich das folgende Kapitel auf ein Kapitel namens "Level-of-Detail AI for a Large Role-Playing Game" aus AI Game Programming Wisdom [4]

Das Rollenspiel Neverwinter Nights von BioWare erschien im Juni 2002. Wie bei vielen Rollenspielen, erschafft sich der Spieler einen Charakter, welcher im Laufe der Zeit stärker und mächtiger wird. Diesen steuert er durch eine definierte Spielwelt, entweder alleine und in einer Gruppe mit anderen Spielern zusammen. In dieser Welt gibt es auch wieder unzählige Monster zu töten, Gegenstände zu sammeln und Quests zu erledigen.

Das ursprüngliche Design zu BioWares Mehrspieler-Spiel Neverwinter Nights (NWN) sieht Abenteuer vor, welche aus verschiedenen Gebieten innerhalb dieses Abenteuers bestehen. Man könnte dies mit einem Level vergleichen. Jedes dieser Abenteuer besteht aus verschiedenen Gebieten wie z.B. dem Inneren von Häusern, einem Wald, einer Stadt oder einem Teil eines Dungeons. Ein Dungeon ist hierbei ein Gebäude, eine Höhle, eine Burg, in der sich Monster tummeln und meist besonders schwierige Gegner zu finden sind. Jedes dieser Gebiete ist über eine sogenannte Area-Transition, also eine Möglichkeit sich zwischen diesen Gebieten zu bewegen, an andere Gebiete angeschlossen.

BioWare fand heraus, dass es nötig ist, die Anzahl der Ressourcen, welche vom Spiel und von der AI benutzt werden, zu reduzieren. Um dies zu bewerkstelligen, wurden Abenteuer in sogenannte Master-Area-Groups aufgeteilt. Eine Master-Area-Group besteht hierbei aus einem Außengebiet wie z.B. einer Stadt und den in diesem Gebiet liegenden Gebieten, wie z.B. dem Gebäudeinneren der Gebäude der Stadt. Dies führte dazu, dass sobald ein Spieler eine Grenze eines Gebiets erreichte, an der normalerweise ein Transfer in ein anderes Gebiet stattfinden würde, die Meldung kam, dass erst die ganze Gruppe gesammelt werden müsse, um weiterzureisen. Und genau hier kommt das Problem auf. Dies war laut den Benutzern eines der meistgehassten Funktionen des Spiels, da die Spieler in den seltsamsten Fällen wirklich als kompakte Gruppe durch die Welt reisen, sondern sich eher verteilt bewegen. Somit kamen Wartezeiten auf, welche den Spielfluss aus technischen Gründen unterbrochen haben.

Eines der Designziele von NWN war es nun, dass solche Restriktionen im Mehrspieler-Spiel nicht mehr auftreten dürfen. Somit wurde die Entscheidung getroffen, dass alle Gebiete eines Abenteuers im Speicher gehalten werden, wobei die Größe dieser Abenteuer ein wenig verringert wurde. Dies führte zu einem großen Problem für die AI-Entwickler. In einem Abenteuer befinden sich tausende von NPCs, also autonome Agenten, welche in der Spielwelt agieren. Jeder dieser Agenten stellt Anforderungen an die CPU durch Bewegung, Kämpfe und sonstige Interaktionen untereinander. Somit besteht die Gefahr, dass das ganze Programm zum Stillstand kommen könnte, während dessen sich die AI durch tausende von Anfragen arbeitet.

4.1 Level-of-Detail: Grafik-Engine

Grafik-Engines haben mit ähnlichen Problemen zu kämpfen. Auf dem Bildschirm müssen eventuell eine Vielzahl von Objekten gerendert werden, wobei sich jedes Objekt aus vielen Dreiecken zusammensetzt. Allerdings kann jeder Grafikchip nur eine gewisse Anzahl von Dreiecken gleichzeitig rendern. Eine Lösung ist es, die Anzahl der Objekte beizubehalten, allerdings mit einer geringeren Anzahl von Dreiecken bei gewissen Objekten. Abhängig von der Komplexität der darzustellenden Szene kann jedes Objekt nun mit einem angemessenen Detaillevel gerendert werden. Dieser Level-Of-Detail Algorithmus kontrolliert und steuert die Anzahl der zu zeichnenden Dreiecke bei gleichbleibender Objektanzahl auf dem Bildschirm. Der Vorteil, 3D-Objekte in verschiedenen Auflösungen oder Detaillevels zu speichern, wurde schon vor 25 Jahren von James Clark behandelt. Ein Objekt, welches auf dem Bildschirm und in der Nähe der Kamera ist, kann hunderttausend Polygone benötigen, um in angemessener Qualität gerendert zu werden, während ein Objekt, was sich in einiger Entfernung zur Kamera befindet, vielleicht mit einer handvoll Polygonen auskommt. Im Abbildung 4 sieht man, wie die 3D-Grafik einer Lampe von einer 10.000 Poly-



Abb. 4. Lampe in verschiedenen Detaillevels aus derselben Distanz betrachtet

gon Version zu einer 48 Polygon Version vereinfacht wird. In Abbildung 5 auf der nächsten Seite sieht man ein Beispiel, wie diese weniger detaillierten Modelle eingesetzt werden. Da das Ganze für Grafiken funktioniert, sollte dies auch für die AI funktionieren. Wenn man AI als die Erschaffung der Illusion einer Intelligenz ansieht, dann sollte eine Spiel-AI dem Spieler Objekte präsentieren, welche sich außergewöhnlich verhalten. Ziel ist es also, detailliertere AI-Algorithmen auf den Objekten auszuführen, welche der Spieler sieht und weniger detaillierte Algorithmen auf Objekten, die der Spieler nicht sieht.

4.2 Level-of-Detail: AI

In Neverwinter Nights kann ein Spieler seinen Spielercharakter sehen, welcher sich in der Mitte des Bildschirms befindet. Die Kamera kann frei um den Spielercharakter herum gedreht werden. Um die Anzahl der Objekte zu kontrollieren,

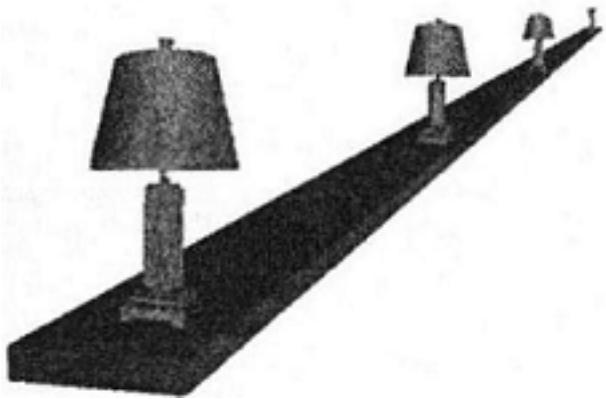


Abb. 5. Lampe in verschiedenen Detailleveln aus verschiedenen Distanzen betrachtet

hat man die Sichtweite der Kamera auf 50 Meter gesetzt, wobei nur Objekte und Kreaturen innerhalb dieser Reichweite angezeigt werden.

Wenn man Objekte klassifiziert, gibt es verschiedene Faktoren. Der beste Algorithmus sollte die Kontrolle von Spielercharakteren übernehmen, da diese immer auf dem Bildschirm zu sehen sind. Des weiteren sollten NPCs, welche mit den Spielercharakteren interagieren, auch eine vernünftige AI bekommen, während für Objekte außerhalb der Sichtweite eine Approximation des Verhaltens oder des Kampfes ausreicht bzw. möglich ist. Somit haben sich für NWN 5 verschiedene

LOD	Classification
1	Player Characters (PCs) (your Party)
2	Creatures fighting or interacting with a PC
3	Creatures within 50 meters of a PC
4	Creatures in the same large-scale area of a PC
5	Creatures in areas without a PC

Abb. 6. Detaillevel in Neverwinter Nights

Klassifikationen von Objekten ergeben, welche von höchster zu niedrigster Priorität geordnet sind und auf der Sichtweite des Spielers beruhen und in Abbildung 6 zu sehen sind.

In Abbildung 7 auf der nächsten Seite sieht man zwei verschiedene Gebiete, wobei jedes Gebiet durch ein großes Quadrat dargestellt wird. Es befinden sich drei Spieler im Spiel und jeder von ihnen kontrolliert einen Spielercharakter. Spielercharaktere sind durch kleine Kreise dargestellt, wobei die großen Kreise die Sichtweite der Charaktere beschreibt. Kleine Quadrate sind NPCs und jeder NPC hat seine LOD Klassifizierung als Zahl innerhalb des Quadrates stehen.

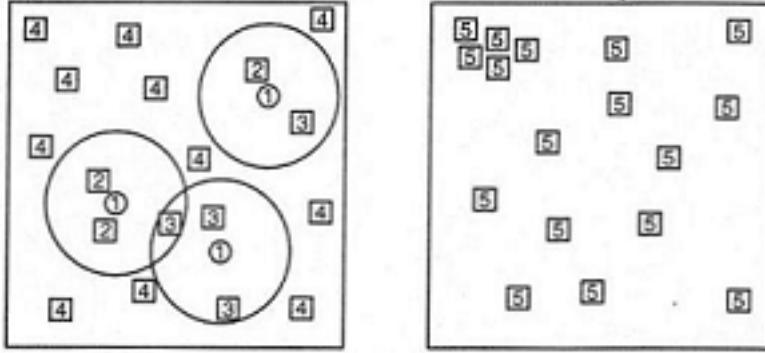


Abb. 7. Erste Klassifizierung der Detaillevel in NWN

Spielercharaktere sind immer LOD 1. NPCs, welche mit dem Spielercharakter interagieren sind LOD 2. NPCs in der näheren Umgebung, also dem Sichtfeld vom Spielercharakter, sind LOD 3, NPCs im selben Gebiet, aber außerhalb des Sichtbereichs LOD 4. NPCs im selben Abenteuer, aber nicht im selben Gebiet wie der Spielercharakter, werden in LOD 5 eingeteilt.

Jeder dieser Level-of-Detail Klassen ist sehr leicht mittels Funktionen der Game-Engine zu erkennen. Beispielsweise macht jeder NPC eine Sichtbarkeitsüberprüfung, um alle NPCs innerhalb von 50 Metern festzustellen. Hierbei kann leicht eine Überprüfung stattfinden, ob dieser NPC ein Spielercharakter ist oder nicht. Weiterhin hat jedes Gebiet ein geskriptetes Ereignis, welches bei Verlassen des Gebiets durch den Spieler ausgelöst wird. Demzufolge ist es einfach, die Gebiete zu finden, in denen Spielercharaktere sind.

Weiter werden jegliche Angriffe oder Interaktionen von NPCs durch eine einzelne Funktion aufgerufen, so dass auch hier ein eindeutiger Punkt ist, um den LOD eines NPCs zu verändern.

4.2.1 Updatezyklen Die wahrgenommene Intelligenz eines NPCs hängt sehr stark mit der Reaktionszeit dieses NPCs auf bestimmte Situationen zusammen. So wird zum Beispiel ein NPC, welcher lange braucht, um einem fliehenden Spieler zu folgen, als nicht so intelligent wahrgenommen wie ein NPC, welcher dies direkt tut. Also bestimmen die 5 LOD Klassen die benötigte Menge an Intelligenz bzw. Interaktivität.

Nachdem jeder NPC in eine der 5 Klassen eingeteilt wurde, bestimmt eine Funktion, wer den nächsten Berechnungszeitraum zugewiesen bekommt. Abbildung 8 zeigt, welche Prozentzahlen für NWN benutzt wurden, da diese gut funktionieren. Anzumerken ist hierbei noch, dass sobald einer Liste erlaubt wird, einen NPC, der in dieser "wartet", abzuarbeiten, dieser NPC immer erst komplett abgearbeitet wird. D.h. eine höher priorisierte Liste kann während eines Berechnungsvorgangs für einen NPC nicht eine neue Berechnung starten. Dies

CPU Time	Level-of-Detail
60%	LOD 1: Player Characters
24%	LOD 2: Creatures interacting with PCs
10%	LOD 3: Creatures in proximity of PCs
4%	LOD 4: Creatures in an area with a PC
2%	LOD 5: Creatures not in an area with a PC

Abb. 8. CPU Zeit pro LOD Level in Prozent

ist wichtig, damit auch die niedrig-priorisierten Listen Fortschreiten und niemals verhungern.

4.3 Einsatzgebiete von Level-of-Detail

4.3.1 Wegfindung Jede AI Aktion benötigt CPU Zeit. Da die Wegfindung über ein komplexes Gelände einen Gutteil der Rechenzeit benötigt, ist es folglich logisch, den Level-of-Detail Algorithmus hier zu implementieren.

Basierend auf der AI-Struktur von NWN wurde entschieden, dass jeder NPC eine Instanz des Wegfindungsalgorithmus speichert. Als Wegfindungsalgorithmus wurde IDA* und andere Computerschach-Techniken benutzt, um Ressourcen zu sparen [5].

Das Gelände ist in 10 mal 10 Meter Felder unterteilt, wobei jedes davon seine Nachbarn kennt. Mittels dieses grobkörnigen Systems ist es möglich, dass man eine Sequenz von Feldern erhält, die man entlanglaufen muss, um ein bestimmtes Ziel zu erreichen. Dies wird im Allgemeinen Inter-Tile-Pathfinding genannt, da hier nur die Sequenz der zu laufenden Felder interessiert, aber nicht, wie innerhalb der einzelnen Felder gelaufen wird. Der volle Pfad wird dann mittels des Intra-Tile-Pathfinding generiert. Hierbei wird ein komplexer Algorithmus zur Wegfindung über ein 3D-Gelände benutzt, der für jedes 10 mal 10 Feld den Weg bestimmt.

Dies bedeutet, dass jeder NPC, welcher auf dem Bildschirm zu sehen ist, also alle NPCs von LOD 1 bis LOD 3, den vollen Wegfindungsalgorithmus mit Inter- und Intra-Tile-Pathfinding implementieren müssen. LOD 4 NPCs implementieren nur den Inter-Tile-Algorithmus, der gerade mal 10 Prozent der Gesamtzeit des kompletten Wegfindungsalgorithmus benötigt. Dazu springen NPCs nach einer bestimmten Zeit - der Zeit die sie normalerweise benötigen würden - zwischen den grobkörnigen Feldern hin und her. Allerdings unbemerkt vom Spieler. LOD 5 NPCs werden nach der normalen Reisezeit einfach an ihr Ziel gesetzt.

4.3.2 Kampf In Neverwinter Nights werden die sehr komplexen Advanced Dungeons and Dragons Regeln benutzt. Diese füllen mehr als 60 DIN A4 Seiten. Somit ist klar, dass das Ergebnis eines Kampfes eine aufwendige Simulation erfordert. Die ist ein guter Ansatzpunkt für einen LOD Algorithmus.

LOD 1 und LOD 2 implementieren das komplette Regelsystem, da dies unmittelbar mit dem Spieler interagiert. Auf LOD 3 ist es schon nicht mehr so wichtig, dass jeder Würfelwurf ausgewertet wird, so dass eine abgespeckte Regelversion, die sehr ähnlich aussieht, zum Einsatz kommt. Bei LOD 4 NPCs, also solchen, welche sich im gleichen Gebiet, aber nicht in Sichtweite befinden, ist nur das Ergebnis des Kampfes relevant. Somit werden hier Schadenskapazitäten bestimmt und dann einfach gegeneinander aufgerechnet, um den Sieger zu bestimmen. Bei LOD 5 NPCs reicht ein Münzwurf, wobei dieser noch durch einzelne Kampfwertungen gewichtet werden kann, so dass ein Ergebnis häufiger auftritt als das andere.

4.3.3 Random Walks Random Walks sind ein Konzept, welches NPCs um ihre "Position" herum in einem bestimmten Radius zufällig herumlaufen lässt. Die einzige Intention hierbei ist es, die Spielwelt ein wenig lebendiger zu gestalten.

Auch hier gilt wieder für LOD 1 und 2 NPCs, dass sie den vollen Wegfindungsalgorithmus implementieren bzw. benutzen. LOD 3 NPCs laufen nur eine gerade Linie zu ihrer Zielposition. LOD 4 und 5 NPCs bewegen sich nicht basierend auf Random Walks, da dies keinen Nutzen hat, wenn der Spieler dies nicht sieht.

4.4 Zusammenfassung

Das Level-Of-Detail Konzept teilt Agenten in verschiedene Klassen ein. Diese Einteilung basiert auf der Sichtbarkeit von Agenten durch den Spieler. Agenten, welche durch den Spieler zu sehen sind, bekommen mehr Rechenzeit zugeteilt als solche, welche nicht sichtbar sind. Dies wird durch eine Skalierung der benutzten Algorithmen geschehen, so dass auf der höchsten Prioritätsstufe volle Algorithmen benutzt werden, während auf der geringsten Prioritätsstufe nur Approximationen benutzt werden. Somit bietet dieser Ansatz eine gute Möglichkeit, eine große Menge von Agenten innerhalb einer Spielwelt intelligent erscheinen zu lassen, gleichzeitig aber den Rechenaufwand kontrollieren zu können, in dem Intelligenz nur dort stattfindet, wo sie vom Spieler auch wahrgenommen werden kann.

5 Radiant AI

Im Titel The Elder Scrolls IV: Oblivion von Bethesda Softworks, erschienen im März 2006, wurde ein neues AI System namens Radiant AI benutzt, welches eigens von Bethesda Softworks entwickelt wurde.

5.1 Idee

Hinter dem Namen steckt folgende Idee. Jeder NPC besitzt einen Zeitplan, den er erfüllen muss. Dieser deckt dabei jeden möglichen Zeitpunkt ab und ist

somit lückenlos. Ziel des Ganzen ist es, ein KI System zu entwerfen, in dem NPCs eigenständig agieren und somit die Illusion eines eigenen Lebens erschaffen wird. Weiterhin sollen sie nicht geskriptet sein, eine eigene Stimmung haben und sowohl Mimik als auch das dementsprechende Verhalten dazu zeigen.

Somit basiert das Radiant AI System auf der folgenden Idee. Jeder NPC bekommt eine Liste von Bedürfnissen, welche er erfüllen muss. Dies können beispielsweise "kämpfen", "lernen", "essen", "bewachen", "beschützen", "schlafen" usw. sein. Aufgrund dieser Liste trifft er Entscheidungen, was wichtiger ist bzw. eine höhere Priorität hat. Weiterhin besitzt auch jeder NPC eine Liste mit Zielen, welche globaler sind als Bedürfnisse wie z.B. "Vemehre dein Gold" oder "Versorge die Stadt mit Lebensmitteln". So wird dem Spieler die Möglichkeit gegeben, am "Leben" der NPCs teilzunehmen, sie dabei zu beobachten, zu stören etc. Zu erwähnen ist hierbei noch, dass trotz der nicht geskripteten Bedürfnisse und Handlungen, die Interaktion mit dem Spieler weiterhin geskriptet ist, was weiterhin notwendig ist, um eine Geschichte erzählen zu können.

5.2 Beispiele

Im folgenden einige Beispiele zum Verhalten der AI:

- Ein Spielercharakter versucht, einen Krug aus einem Wirtshaus zu entwenden. Der Wirt erwischt ihn dabei und ruft die Stadtwachen, so dass sich diese um den Fall kümmern.
- Der Hof eines Bauern wurde von Banditen angezündet. Anstatt zu versuchen, den Hof zu retten, sucht der Bauer erstmal nach seinen Söhnen.

5.3 Probleme

Beim Einstellen des System gab es allerdings Probleme, die sich in einem nicht gewünschten Verhalten äußerten. In den Communitys [6] und Blogs zu The Elder Scrolls IV findet man zahlreiche Beispiele zu diesen Probleme. Einige davon sind:

- Ein NPC bekommt eine Harke und das Ziel "Laub harken". Ein anderer NPC bekommt das Ziel "Wege fegen" und einen Besen. Durch irgendwelche Entscheidungen der NPCs tauschen diese die beiden Gegenstände, so dass sie ihre Ziele nicht mehr direkt umsetzen können. Folge davon ist, dass ein NPC den anderen getötet hat, um seinen Gegenstand zu erlangen und sein Ziel wieder erfüllen zu können.
- Eine Stadtwache wird hungrig. Unter den vielen Möglichkeiten zur Essensbeschaffung entscheidet sie sich für die Jagd im Wald. Aufgrund dessen verlassen die anderen Wachen ihre Posten und die Stadt, um die im Wald jagende Wache einzufangen, da diese ihren Arbeitsplatz verlassen hat und somit die Arbeit verweigert. Während die Stadtwachen also damit beschäftigt sind, die hungrige Wache einzufangen und in den Kerker zu werfen, sind in der Stadt keinerlei Wachen mehr zu finden, woraufhin die anderen NPCs die Händler plündern, da keine Strafverfolgung mehr existiert

5.4 Zusammenfassung

Radiant AI versucht, Nicht-Spieler-Charakteren ein Leben einzuhauchen, indem jeder von ihnen Bedürfnisse und Ziele hat, welche er auch ohne Zutun des Spielers versucht umzusetzen. Bisher gibt es noch Probleme in der Umsetzung, da einige Verhaltensweisen nicht erwünscht sind, aber im Großen und Ganzen ist ersichtlich, dass es sich um eine mächtige Idee handelt, die im Laufe der Jahre verbessert und ausgereifter wird. Leider gibt es zu dieser interessanten Idee nur sehr wenig bzw. keine Informationen.

6 Fazit

Rollenspiele greifen auf viele Technologien zurück, welche in anderen Gebieten entwickelt wurden, ob es die Wegfindung ist oder eine Persönlichkeitssteuerung. Hierbei ist allgemein aber zu beobachten, dass das Thema AI und Rollenspiele erst in letzter Zeit an Bedeutung gewinnt, da geskriptete Verhaltensweisen von NPCs nicht mehr zufriedenstellend sind. Die Probleme, die sich hierbei insbesondere bei Rollenspielen ergeben, sind vor allem performancetechnischer Natur. Man könnte sicherlich viel intelligenteren NPCs bereitstellen, allerdings zieht das auch erhöhte Anforderungen an CPU und Speicher nach sich, was gerade bei Spielwelten mit mehreren tausend Agenten zu einem großen Problem wird.

Insgesamt wird dem Thema in den nächsten Jahren immer größere Bedeutung zukommen wie schon die Entwicklung der Radiant AI zeigt. Es deutet alles darauf hin, dass die Spielwelten immer lebendiger, immer intelligenter sein werden.

Leider ist aber auch zu sagen, dass wenig bis keine Literatur zum gesamten Thema zu finden ist, was sicherlich daran liegt, dass Rollenspiele auf dem Markt der PC Spiele erst in den letzten Jahren eine wichtigere Rollen einnehmen.

Literaturverzeichnis

1. Wikipedia - Rollenspiel (Spiel), http://de.wikipedia.org/wiki/Rollenspiel_%28Spiel%29/
2. Wikipedia - Computer-Rollenspiel, <http://de.wikipedia.org/wiki/Computer-Rollenspiel>
3. Blizzard Entertainment, World of Warcraft Europe, <http://www.wow-europe.com/>
4. Steve Rabin - AI Game Programming Wisdom, Charles River Media; 1st edition (March 12, 2002)
5. Mark Brockington - Pawn Captures Wyvern: How Computer Chess Can Improve Your Pathfinding, Game Developers Conference 2000 Proceedings
6. TTLG Forums - Radiant A.I. ?, <http://www.ttlg.com/forums/showthread.php?t=105339>

KI und CI in Pokergames

-Seminararbeit-

Computational Intelligence in Computerspielen

Holger Danielsiek

holger.danielsiek@udo.edu
TU Dortmund

1 Einleitung

Computerspiele sind in der Informatik schon seit einem langen Zeitraum ein interessantes Forschungsgebiet. Sie besitzen aufgrund ihrer festen Regeln und einem spezifizierten Ziel wünschenswerte Eigenschaften für die Forschung. Oft müssen diese Spiele aber wegen ihrer hohen Komplexität abstrahiert werden, damit die klassische KI nicht überfordert ist. Die Hauptziele der Forschung liegen in der Untersuchung, wie gut ein Computerspieler bei einem bestimmten Spiel werden kann und wie stark er im Vergleich zu menschlichen Gegenspielern ist. Meistens wird ein solcher Vergleich über eine bestimmte Anzahl von Spielen gemessen.

1.1 Definition des Begriffs Spiels

Der Kulturhistoriker Johan Huizinga [1] stellte 1939 folgende Definition auf:

„Spiel ist eine freiwillige Handlung oder Beschäftigung, die innerhalb gewisser festgesetzter Grenzen von Zeit und Raum nach freiwillig angenommenen, aber unbedingt bindenden Regeln verrichtet wird, ihr Ziel in sich selber hat und begleitet wird von einem Gefühl der Spannung und Freude und einem Bewusstsein des Andersseins als das gewöhnliche Leben.“

Somit dient ein Spiel also zur Unterhaltung der Spieler und soll vor allem Spaß machen. In der heutigen Zeit können Spiel und Beruf teilweise ineinander übergehen wie beispielsweise im Profi-Fussball oder auch beim Pokern.

1.2 Aufbau der Ausarbeitung

Zu Beginn dieser Seminarausarbeitung werden in Kapitel 2 die allgemeinen Regeln des Kartenspiels Poker eingeführt und in Kapitel 3 einige häufig gespielte Varianten vorgestellt sowie in Kapitel 4 auf Poker aus wissenschaftlicher Sicht als Spiel unvollständiger Informationen eingegangen. Anschließend wird in Kapitel 5 auf bereits veröffentlichte Arbeiten eingegangen, denn die Entwicklung von Strategien für Poker beschäftigte einige bekannte Forscher und im Hauptteil dieser

Ausarbeitung, Kapitel 6, werden exemplarisch vier Anwendungen verschiedener CI-Methoden gezeigt. Abschließend gibt es eine Zusammenfassung in Kapitel 7 und ein damit verbundener Ausblick über weitere Möglichkeiten die KI zu verbessern.

2 Pokerregeln

Wie bereits in der Einleitung erwähnt, besitzen alle Spiele fest definierte Regeln und ein spezifiziertes Ziel. Allerdings gibt es im Poker zahllose Varianten, die alle in den Casinos oder online angeboten werden. In den folgenden Abschnitten werden zuerst die Regeln erklärt und im nächsten Kapitel dann die unterschiedlichen Varianten vorgestellt. Nachzulesen sind die Regeln und Varianten in unzähligen Büchern über Poker, beispielsweise in [2].

2.1 Wertigkeit der Blätter

Beim Poker unterscheidet man zwischen zehn möglichen unterschiedlichen Kombinationen (Hand), die wertvoller werden je seltener sie vorkommen, siehe Tabelle 1. Es gewinnt immer der Spieler¹ mit der stärksten Hand. Haben zwei oder mehr Spieler eine gleichstarke Kombination, so entscheiden die höheren Karten in der Kombination und bei deren Gleichheit die Beikarten (Kicker). Hierunter fallen Karten, die die Kombination nicht stärker machen, aber helfen zwei gleichwertige Blätter zu unterscheiden. Beispielsweise ist ein Paar Buben besser als ein Paar Achte, besitzen aber beide Spieler das gleiche Paar, so gewinnt der Spieler, der ein Ass als Beikarte hat gegen den Spieler der kein Ass besitzt. Haben zwei Spieler einen Flush, so ist entscheidend, welcher Spieler den höheren Flush besitzt, wobei zuerst die beiden höchsten Karten des Flushes miteinander verglichen werden und bei Gleichheit die nächsthöheren, etc. Sind alle fünf Karten identisch, teilen sich beide Spieler den Gewinn (Split Pot). Haben zwei Spieler eine Straße, so ist ebenfalls entscheidend, welcher Spieler die höhere Straße besitzt, sind beide Straßen gleich, kommt es zum Split Pot. Grundsätzlich werden beim Poker immer die fünf besten Karten betrachtet, so dass ein Kicker als sechste Karte nicht in Frage kommt, womit möglicherweise sonst ein Split Pot noch zu verhindern wäre.

2.2 Setzrunden

Bei den meisten Pokervarianten müssen ein oder zwei Spieler vor der Kartenausgabe ein sogenanntes Bring-In (Blinds) bezahlen. Es gibt jedoch auch Spiele, in denen alle Spieler vor Erhalt ihrer Karten schon eine Pauschalabgabe (Ante) machen müssen. Mit Erhalt der ersten Karten startet nun die erste Setzrunde. Wurden Blinds bezahlt, müssen auf jeden Fall alle Spieler, die im Spiel (Pot)

¹ Um unnötige Formulierungen abzufangen, wird im weiteren Verlauf immer die männliche Form für Spieler und Spielerinnen gewählt.

Tabelle 1. Wertigkeit der verschiedenen Hände im gewöhnlichen Poker, sortiert von schwach nach stark und die Wahrscheinlichkeit ihres Auftretens bei der Variante fünf aus sieben Karten.

Name	Erklärung	Wahrscheinlichkeit
Höchste Karte	Keine der unteren Kombinationen	17,41 %
Ein Paar	Zwei Karten gleichen Wertes	43,83 %
Zwei Paare	Zwei Paare	23,50 %
Drilling	Drei Karten gleichen Wertes	4,83 %
Straße	Fünf Karten in einer Reihe	4,62 %
Flush	Fünf Karten in einer Farbe	3,03 %
Full House	Ein Drilling und ein Paar	2,60 %
Vierling	Vier Karten gleichen Wertes	0,17 %
Straight Flush	Straße in einer Farbe	0,028 %
Royal Flush	Straße in einer Farbe mit Ass als höchste Karte	0,0032 %

bleiben wollen, diese Blinds auch zahlen. Zusätzlich gibt es noch die Möglichkeit selber etwas zu setzen (Bet), weiterzuerhöhen (Raise) oder eine Erhöhung mitzugehen (Call). Jeder Spieler hat immer auch die Möglichkeit auszusteigen (Fold). In diesem Fall legt er seine Karten hin und scheidet aus dem Spiel aus. Den Pot kann er nun nicht mehr gewinnen und seine bis hierher getätigten Einsätze sind verloren. Haben alle noch im Spiel befindlichen Spieler den gleichen Einsatz gebracht, werden je nach Spielvariante neue Karten ausgeteilt, bzw. Karten eingetauscht, siehe Kapitel 3 über Pokervarianten. Nach den festgelegten Runden hat ein Spieler entweder alle Mitspieler herausgeboten und so den Pot gewonnen ohne seine Karten zeigen zu müssen oder es kommt zum Aufdecken der Karten der noch involvierten Mitspieler (Show-Down). Gewinner des Pots ist in diesem Fall der Spieler mit der besten Hand, siehe 1.

Entscheidend bei Setzen ist auch die **Position**, an der sich ein Spieler befindet. Hierunter versteht man beim Poker, ob man als einer der ersten Spieler eine Entscheidung fällen muss oder ob man weit hinten sitzt und so den Vorteil hat, dass man schon Informationen über die Hände seiner Vorgänger erhält. Die beste Position hat der Kartengeber, da er als letzte Person eine Entscheidung fällen darf. Bei vielen Pokervarianten, die im folgenden Kapitel 3 erwähnt sind, kommt es dazu, dass in früher Position Karten häufiger weggeworfen werden als weiter hinten.

3 Pokervarianten

Wie schon im vorangegangenen Kapitel erwähnt, gibt es unterschiedliche Varianten. Als erstes Unterscheidungskriterium gibt es **Turnierspiele** und **Sit-and-Go-Spiele**. Bei den Turnierspielen erhalten alle Teilnehmer zu Beginn die gleiche Anzahl an Chips und ein Turnier läuft solange bis ein Spieler alle Chips besitzt. Spieler, die keine Chips mehr haben, scheiden aus. Bei wenigen Turnieren können sie sich auch wieder einkaufen (Re-Buy) und erhalten neue Chips.

Tabelle 2. Auszahlung von PokerStars an die platzierten Spieler von www.pokerstars.com

Platzierung/ Teilnehmer	-27	28-45	46-100	101-200	201-300	301-400	401-500	501-600	601-800	801-1000	1001-1200	1201-1500	1501-2000	2001-2500	2501-3000	3001-3500	3501-4000	4001+	
1.	50.00%	40.00%	30.00%	27.50%	25.00%	25.00%	25.00%	25.00%	25.00%	25.00%	25.00%	25.00%	25.00%	25.00%	25.00%	25.00%	25.00%		
2.	30.00%	24.00%	20.00%	17.50%	16.00%	15.40%	15.00%	14.30%	14.00%	14.00%	14.00%	13.50%	13.00%	12.40%	12.00%	12.00%	12.00%	11.95%	
3.	20.00%	16.00%	12.00%	11.90%	11.60%	11.10%	10.00%	9.55%	9.20%	9.00%	8.80%	8.50%	8.20%	7.90%	7.60%	7.30%	7.00%	7.45%	
4.																			
5.																			
6.																			
7.																			
8.																			
9.																			
10. bis 18.																			
19. bis 27.																			
28. bis 36.																			
37. bis 45.																			
46. bis 54.																			
55. bis 63.																			
64. bis 81.																			
82. bis 99.																			
100. bis 135.																			
136. bis 180.																			
181. bis 225.																			
226. bis 270.																			
271. bis 324.																			
325. bis 378.																			
379. bis 423.																			

Zum Ende des Turniers wird das Startgeld mittels eines vorher vereinbarten Schlüssel auf die platzierten Spieler aufgeteilt. Als Beispiel ist die Verteilung von PokerStars² aufgeführt, siehe Tabelle 2. Im Gegensatz hierzu stehen die Sit-and-Go-Spiele, die auch unter dem Namen **Cashgames** bekannt sind. Bei dieser Spielform kann sich der Spieler jederzeit mit seinem Geld an den Tisch setzen und an der Runde teilnehmen, sofern er einen Mindestbetrag einsetzt und ein Platz am Tisch frei ist. Auch ein Ausstieg ist zu jeder Zeit möglich.

In den folgenden Abschnitten werden die beliebtesten Pokerarten vorgestellt. Hier gibt es drei verschiedene Kategorien, in die man alle Spiele einordnen kann.

3.1 Draw

Draw-Spiele sind die eigentlichen Urformen des Pokers. Hierbei erhält jeder Spieler seine eigenen Karten, die nur er sehen kann. Die Form **Five Card Draw** kennt man aus vielen Western und wurde vor zehn bis zwanzig Jahren eigentlich ausschließlich gespielt. Hierbei erhalten alle Spieler fünf Startkarten (Hole Cards). Nach einer Setzrunde können die Spieler einmal beliebig viele ihrer Karten gegen neue eintauschen. Nach einer zweiten Setzrunde müssen alle noch im Spiel befindlichen Spieler ihre Karten zeigen und der Spieler mit der höchsten Hand gewinnt den Pot. Die zweite häufig gespielte Draw-Variante ist **Triple Draw**, das sich von dem erwähnten Five Card Draw nur darin unterscheidet, dass den Spielern nun drei Tauschrunden zur Verfügung stehen, zwischen denen jeweils eine neue Setzrunde durchgeführt wird.

3.2 Stud

Bei Stud-Spielen bekommen die Spieler wie bei den Draw-Varianten ihre eigenen Karten, allerdings besteht der Unterschied zu den oben aufgeführten Varianten darin, dass nun einige Karten offen auf den Tisch gelegt werden, so dass die Mitspieler sie sehen können. Die am häufigsten gespielte Stud-Variante ist **Seven Card Stud**. Zu Beginn der Runde erhalten hierbei alle Spieler zwei verdeckte und eine offene Karte. Nach der ersten Setzrunde erhalten alle noch im Pot befindlichen Spieler eine weitere offene Karte (Fourth Street). Auf die Setzrunde folgt wiederum eine offene Karte (Fifth Street), eine Setzrunde und eine weitere offene Karte (Sixth Street). Die siebte Karte gibt es nach der Setzrunde wieder für alle noch verbliebenen Spieler verdeckt. Nach der abschließenden Setzrunde gibt es den Show-Down, sofern noch mehrere Spieler im Pot verblieben sind.

Eine andere Stud-Variante stellt **Razz** dar. Hierbei wird aber die im vorhergehenden Kapitel 1 beschriebene Wertigkeit auf den Kopf gestellt. Gewinner des Pots ist der Spieler mit der niedrigsten Hand, wobei Straßen und Flushes nicht berücksichtigt werden. Das heißt, die beste Hand ist Ass (niedrigste Karte im Spiel), 2, 3, 4, 5. Als Vergleichskriterium zählt hierbei zuerst die fünfniedrigste Karte. Ist diese bei zwei Spielern identisch, zählt die viertniedrigste, etc. Beispielsweise gewinnt der Spieler mit der Hand 7, 5, 4, 3, 2 gegen 7, 6, 3, 2, Ass. Der Ablauf mit Setzen und Kartenverteilten ist identisch dem Seven Card Stud.

² www.pokerstars.com

Bei allen Stud-Spiele beginnt immer der Spieler eine Setzrunde, der aktuell die beste Kombination aller offenen Karten hat, wobei hier bei Gleichheit die Reihenfolge Pik ist besser als Herz, Karo und Kreuz gilt. Das führt dazu, dass es während eines Spiels zu mehreren Positionswechseln³ kommen kann.

3.3 Hold'em

Hold'em-Spiele zeichnen sich dadurch aus, dass die Spieler sowohl eigene verdeckte Karten (Hole Cards) erhalten als auch Karten (Community Cards) offen für alle Spieler in die Mitte des Tisches gelegt werden. Die beliebteste und am meisten gespielte Variante ist **Texas Hold'em**. Hierbei erhalten alle Spieler zwei verdeckte Karten. Nach der ersten Setzrunde kommen drei offene Karten für alle Spieler in die Tischmitte (Flop). Nach einer weiteren Setzrunde kommt die vierte offene Karte auf den Tisch (Turn). Es folgt eine erneute Setzrunde, ehe die fünfte und letzte Karte (Turn) offen auf den Tisch gelegt wird. Nun kommt es zur abschließenden Setzrunde. Eine andere noch gespielte Hold'em-Variante ist **Omaha**. Der Unterschied zu Texas Hold'em liegt hierbei nur bei der höheren Anzahl von Hole Cards und ihrer Verwendung. Zu Beginn erhalten hierbei alle Spieler vier verdeckte Karten, von denen sie zum Schluss genau zwei Verwenden müssen. Das heißt, eine Omaha-Kombination setzt sich immer aus zwei Hole und drei Community Cards zusammen.

3.4 Begriffe, die man gehört haben sollte

Neben den drei beschriebenen Kategorien kann man die Spiele noch an Hand von anderen Attributen unterscheiden: So gibt es Varianten mit **High-Low**, bei denen der Gewinn geteilt wird, zwischen dem Spieler mit der besten Kombination (High) und dem Spieler mit den niedrigsten fünf Karten (Low). Diese Low-Hand ähnelt stark der Variante Razz. Sie wird allerdings nur ausgezahlt, wenn die fünfniedrigste Karte eine Acht oder noch geringer ist.

Außerdem unterscheidet man zwischen verschiedenen Limits. Es gibt sogenannte **Limit-Spiele**, bei denen immer um eine bestimmte Summe erhöht wird. Zusätzlich werden auch **no-Limit-Spiele** gemacht, bei denen jeder Spieler jederzeit seine gesamten Chips einsetzen kann. Hat ein Spieler alles gesetzt, so bezeichnet man ihn als **all-in**. Erhöhen nach dem all-in eines Spielers weitere Spieler, so spielen diese um einen sogenannten **Side Pot**. Die Ursache hierfür liegt darin, dass ein Spieler nur ein x-faches davon gewinnen kann, wie er selber einsetzt, wenn noch x Spieler mitgehen (callen). Machen andere Spieler höhere Einsätze, so spielen auch nur sie um den Betrag. In seltenen Fällen spielt man auch noch **Pot-Limit-Spiele**. Hierbei darf ein Spieler jeden Betrag setzen, höchstens jedoch soviel, wie sich im Pot befindet.

Es gibt jetzt auch noch Turniere, die verschiedene Pokervarianten in sich vereinigen. Als bekannteste Variation gilt **HORSE**. Wobei die Abkürzung für **Hold'em**, **Omaha**, **Razz**, **Seven Card Stud** und **Eight or better** (anderer Name

³ Auf die Wichtigkeit der Position wird später noch eingegangen.

für Seven Card Stud High/Low) steht. Bei diesen Spielen werden reihum in einem bestimmten Zeitintervall die unterschiedlichen Varianten gespielt.

4 Spiele mit unvollständiger Information

Poker ist in der Wissenschaft so interessant, da es sich um ein Spiel mit ***unvollständiger Information*** handelt. Hierbei weiß man als Spieler im Gegensatz zu beispielsweise Schach oder Dame nicht alles über die Karten des Gegners, sondern muss versuchen an Hand von Einsätzen oder anderen ***Tells***, beispielsweise Mimik, Gestik oder Zeit des Überlegens, herausfinden, was der Gegner für ein Blatt haben könnte. Ein Nachteil beim Online-Poker ist, dass Mimik und Gestik der Kontrahenten nicht gesehen werden und somit nicht in die Berücksichtigung einfließen können.

4.1 Outs

Bei der Überlegung, welche Entscheidung ein Spieler treffen soll, spielt die Anzahl der ***Outs*** eine nicht unwesentliche Rolle. Hierunter versteht man die Anzahl der Karten, die einem Spieler noch helfen, um eine stärkere Hand zu bekommen. Hat man beispielsweise einen ***Draw***⁴, so zählen als Outs die Karten, die die Sequenz vervollständigen. Man unterscheidet zwischen Open Ended- und Gutshot-Straight- sowie einem Flush-Draw. Bei einem ***Open Ended-Straight-Draw*** besitzt man vier aufeinanderfolgende Karten, beispielsweise 4,5,6,7. Als Outs gelten in diesem Fall die vorhergehenden oder folgenden Karten. Im Beispiel also alle Dreien oder Achten, somit hat ein Spieler mit einem Open Ended-Straight Draw acht Outs. Bei einem ***Gutshot-Straight-Draw*** besitzt man vier Karten für eine Straße bei denen mittig eine fehlt, beispielsweise 3,4,6,7. Als Outs gelten in diesem Fall die fehlenden Karten. Im Beispiel also alle Fünfen, somit hat ein Spieler mit einem Gutshot-Straight Draw vier Outs. Bei einem ***Flush-Draw*** besitzt man vier Karten in einer Farbe. Als Outs gelten in diesem Fall die weiteren Karten dieser Farbe. Da er bereits vier von den 13 Karten besitzt, die es von einer Farbe gibt, hat ein Spieler mit einem Flush Draw hat neun Outs.

4.2 Pot Odds

Als ***Odds*** bezeichnet man Wahrscheinlichkeit, die vorhandenen Outs aus den restlichen Karten zu bekommen. Wichtiger für einen Pokerspieler sind laut vielen Büchern von Profi-Spielern, beispielsweise [2, 3, 4] die ***Pot Odds***. Hierbei wird zwischen zubringendem Einsatz, möglichen Gewinn (Potgröße) und der Wahrscheinlichkeit den Pot zu gewinnen (Odds) abgewiegt. Gut für den Spieler sind große Pots, in die er selber nur noch wenig einzahlen muss, um viel

⁴ Unter dem Begriff Draw versteht man alternativ auch noch die Situation, in der ein Spieler bereits vier von fünf Karten besitzt, die er für eine Sequenz (Straße oder Flush) benötigt.

zu gewinnen. Überlegungen, ob sich ein Einsatz noch lohnt, müssen angestellt werden, wenn die Wahrscheinlichkeiten den Pot zu gewinnen ungefähr in gleichem Verhältnis zueinanderstehen wie der zubringende Einsatz zur bestehenden Potgröße. Ist das Verhältnis Einsatz/Potgröße viel größer als die Gewinnwahrscheinlichkeit, so sollte jeder vernünftige Spieler die Karten hinlegen (fold) und auf den Pot verzichten. Es sei denn, er ist sich sicher sein Gegner blufft.

4.3 Bluffen

Beim Bluffen versucht ein Spieler, seinen Gegenspielern eine bessere Hand darzustellen als er selber wirklich besitzt und sie dadurch verleiten ihre Karten zu schmeißen und ihm so den Pot zu überlassen. In seinem Buch erwähnt der Autor David Sklansky [4], dass gerade Anfänger im Poker meinen, es würde viel geblufft, doch in Wirklichkeit käme es viel seltener als erwartet zu einem Bluff. Er legt einem Anfänger sogar nah, zuerst völlig auf das blaffen zu verzichten. Allerdings wird man als Spieler, der nur ganz wenig Hände spielt (tight player), keine großen Pots gewinnen können. Jeder Gegenspieler schöpft Verdacht, wenn man dieser Spieler erhöht und wird größtenteils seine Karten wegwerfen, sofern diese nicht überragend sind. Somit bietet sich dem Spieler also hier auch mal die Gelegenheit zu bluffen und sein Image auszuspielen. Andererseits glaubt man einem „Vielspieler“ (loose player) auch manchmal seine gute Hand nicht. Erfolgreiches Bluffen ist also immer auch davon abhängig von dem Image, dass man sich während des laufenden Spiels aufgebaut hat. Weitere Entscheidungskriterien damit ein Bluff erfolgreich ist, sind die Position, siehe Abschnitt 2.2 und die genaue Beobachtung der Gegner, denn wenn diese mitgehen, ist der Bluff hinfällig und das eingesetzte Geld verloren.

5 Einsatz von KI

Die Künstliche Intelligenz (KI) bei Pokerspielen hat eine lange Geschichte, die bereits 1944 mit einer Veröffentlichung von von Neumann und Morgenstern [5] beginnt. Auch wenn es sich hierbei um ein stark vereinfachtes Modell handelt, in dem nur zwei Spieler mit jeweils zwei Karten spielen und keine weiteren Setzrunden stattfinden. Die Autoren erkannten, dass Bluffen ein wichtiger Bestandteil des Pokerns ist, welches bei einer Automatisierung des Spiels berücksichtigt werden sollte. Als weiterer Pionier der frühen KI-Entwicklung im Poker gilt Nicolas Findler, der in seinem 1977 veröffentlichten Paper [6], erstmals über ein gewöhnliches Pokerspiel mit fünf zu verwendeten Karten schreibt, in dem er Monte Carlo Simulation als Methode einsetzt. Als Variante verwendete er das Five Card Draw Poker, für das er folgende Vereinfachungen vornimmt:

- Es gibt kein Ante.
- Die Position beim Setzen wird nicht berücksichtigt.
- Der Computerspieler spielt immer als letzter⁵.

⁵ beste Position, da er auf die vorhergehenden Aktionen eingehen kann

Als Ergebnis des Papers kommt Findler zu der Feststellung, dass man dynamische und lernfähige Algorithmen beim Pokern für ein erfolgreiches Spiel benötigt, da die bisher verwendeten statischen mathematischen Modelle leicht besiegt werden konnten, weil man ihre Strategie nach relativ kurzer Beobachtungszeit durchschauen konnte.

6 Einsatz von CI-Methoden

In diesem Kapitel werden Arbeiten aufgegriffen, die Methoden der Computational Intelligence benutzen, um die KI beim Pokern zu verbessern.

6.1 Einsatz von Künstlichen Neuronalen Netzen beim Poker

Künstliche Neuronale Netze (KNN) sind der Lernweise menschlichen Gehirns nachempfunden. Durch die Verbindung und die Interaktion der einzelnen Neuronen haben sie die Fähigkeit, Strategien zu entwickeln und Verhaltensmuster (patterns) in einer verrauschten Datenmenge zu finden. Genauere Erläuterungen sind in der Seminarausarbeitung über Neuronale Netze oder in unzähligen Büchern beispielsweise in [7]. Daher eignet sich ihr Einsatz auch, um beim Texas Holdem Poker (siehe Abschnitt 3.3) einem Spieler Hilfestellungen zu geben. In einem Paper aus dem Jahr 1999 [8] beschreibt Aaron Davidson ein von ihm entwickeltes Netz, mit dessen Hilfe er die nächste Aktion (fold, call, bet/raise) seiner Gegner voraussagen will.

Aufbau. Die insgesamt 19 Eingangsneuronen des entwickelten Netzes umschließen folgende sieben real-Werte aus dem Intervall $[0, 1]$:

1. Immediate POT ODDS, die sich aus $(\#outs)/(\#\text{unbekannte Karten}^6)$ zusammensetzt
2. Wettverhältnis, das sich aus $(\#bets)/(\#\text{(bets+calls)})$ zusammensetzt.⁷
3. Potverhältnis, das sich aus $(\text{eigener Einsatz})/(\text{Potgröße})$ zusammensetzt.
4. Anzahl der Spieler, die am Spiel teilnehmen, wobei hier durch 10^8 geteilt wird, um im Intervall zu sein.
5. Anzahl der aktiven Spieler, also der Spieler, die sich noch in der Hand befinden und noch nicht gepasst (fold) haben. Ebenfalls durch 10 geteilt, um im Intervall zu sein.
6. Anzahl der passiven Spieler, also der Spieler, die gepasst haben. Ebenfalls durch 10 geteilt, um im Intervall zu sein.
7. Quote an hohen Gemeinschaftskarten, die sich aus $(\#\text{Assen} + \#\text{Königen} + \#\text{Damen auf dem Tisch})/(\#\text{Karten auf dem Tisch})$ zusammensetzt.

⁶ Postflop=47, Postturn=46, (Postriver entfällt, da $\#\text{outs}=0$)

⁷ Mit bets sind sowohl bet als auch (re-)raise Operationen gemeint.

⁸ 10 ist die höchste Anzahl an Mitspielern bei einem gewöhnlichen (Limit) Texas Hold'em Spiel

Zusätzlich berücksichtigt dieser Ansatz noch folgende zwölf boolsche Eingangsneuronen:

1. Committed in den Pot, d.h. man hat entweder mindestens das Top-Pair⁹ oder einen erfolgsversprechenden Draw.
2. Keine Erhöhung zu callen.
3. Eine Erhöhung zu callen.
4. Mehr als eine Erhöhung zu callen.
5. Flop liegt auf dem Tisch.
6. Turn liegt auf dem Tisch.
7. River liegt auf dem Tisch.
8. Letzte Runde wurden Erhöhungen gemacht.
9. Letztes Mal wurde selber erhöht.
10. Ein Flush ist noch möglich.
11. Ein Ass liegt auf dem Tisch.
12. Ein König liegt auf dem Tisch.

Neben diesen 19 Eingangsneuronen besitzt das entwickelte Feed Forward Netz vier Knoten in der versteckten Schicht und drei Ausgabeneuronen, die die nächste Aktion (fold, call, bet/raise) des Gegners vorhersagen sollen. Jeder Knoten im Netz verwendete die sigmoide Aktivierungsfunktion.

Während des Trainings mit insgesamt 2529 aufgezeichneten Spielen von sieben verschiedenen menschlichen Gegen Spielern wurde Backpropagation mit Gradientenabstieg verwendet, um die verschiedenen Knotengewichte einzustellen. Diese Trainingsdaten wurden anschließend mit einem Testdatensatz bestehend aus 1878 zusätzlich aufgezeichneten Spielen validiert. Neben einigen kleineren Problemen beim Training, beispielsweise lokale Optima überwinden oder zu starke Oszillation, verhinderte vor allem die Prä-Flop-Setzrunde gute Lernergebnisse. Nach Ausschluss dieser Runde konnte das Netz ansprechendes Spielerverhalten lernen. Ausschlaggebend hierfür ist nach Ansicht von Davidson die viel weniger vorhandene Information, da Spieler nur ihre zwei Hole Cards besitzen und noch keine Gemeinschaftskarten ausgelegt wurden.

Ergebnisse. Zuerst wurden die trainierten Netze auf weitere Testdaten des selben Spielers angesetzt, um zu überprüfen, ob das Netz wirklich die Trainingsdaten der Spieler generalisiert hat oder ob es einfach eine Look-up-Tabelle aufgebaut hat, mit deren Hilfe es nur die schon bekannten Daten abliest. Hierbei lassen sich beachtliche Erfolge erkennen, denn das Netz ist in der Lage, nach dem Training auf einen der sieben verschiedenen Eingabespieldaten dessen Aktionen aus dem Testdatensatz, der weitere aufgezeichnete Spiele dieses Spielers beinhaltet, zu durchschnittlich 81 % vorauszusagen, wobei durchschnittliche deterministische Vorhersagen nur zu 57 % und aufwändiger deterministische Vorhersagen, die von Davidson für diesen Versuch entwickelt wurden, zu 71 % richtig lagen.

⁹ Unter Top-Pair versteht man beim Poker, das mögliche höchste Paar mittels einer eigenen und einer Gemeinschaftskarte. Also den Besitz einer Karte gleichen Wertes zu der aktuell höchsten Karte auf dem Tisch

Anschließend wurden die Ergebnisse verallgemeinert. Als Versuchsaufbau wurde eine Trainingsmenge mit 1397 Daten von sechs verschiedenen Spielern erstellt. Nach 25 Trainingswiederholungen war das Netz in der Lage 85 % der Testmenge richtig vorauszusagen, wobei diese Testmenge Daten von sieben unterschiedlich starke Spielertypen umfasst. Nachdem alle Netze bis hierher offline trainiert wurden, gibt es abschließend noch ein online-Trainingsversuchsaufbau, bei denen die gemachten Beobachtungen langsam¹⁰ mit in die Trainingsmenge einfließen. Nach 400 Zyklen erreicht der Algorithmus eine Vorhersagegenauigkeit von über 90 %, wobei immer daran erinnert werden sollte, dass die Preflop-Setzrunden nicht berücksichtigt werden.

Insgesamt ist der Ansatz mit neuronalen Netzen noch ausbaubar, beispielsweise hinsichtlich der Echtzeitfähigkeit oder der zu bewertenden Situationen. So lässt sich beispielsweise ein Pair On Board-Eingangsneuron noch gut einbauen. Im Vergleich zu Action Frequencies Look-up-Tables, in denen aufgeführt ist, wann ein bestimmter Gegner eine bestimmte Aktion in einem bestimmten Zusammenhang ausführt, siehe Kapitel 7 in [9], lassen sich Erfolge sehen.

6.2 Einsatz von evolutionären Algorithmen beim Poker

Auch evolutionäre Algorithmen (EAs) sind randomisierte Suchheuristiken. Sie sind dem biologischen Vorbild der darwinschen Evolutionstheorie nachempfunden, bei denen die besten Individuen überleben und Nachkommen produzieren. Beim Einsatz dieser CI-Methode werden Lösungen für ein gegebenes Problem mittels einer Zielfunktion bewertet und die besten Lösungen werden weiter verwendet und leicht modifiziert, in der Hoffnung noch bessere Lösungen zu generieren. Detailliertere Beschreibungen sind in der Seminararbeit über evolutionäre Algorithmen oder in [10] zu finden. Richard G. Carter und John Levine zeigen in Paper [11] die Möglichkeit des Einsatz eines EAs zur Aktionsauswahl (fold, call oder bet/raise) beim Pokern.

Aufbau. Die beiden Autoren verwenden in ihrem Paper die Pokervariante Texas Hold'em No Limit. Allerdings vereinfachen sie das Spiel durch die Einschränkung auf ein „Zehn-Spieler, Gewinner-bekommt-alles, All-In oder Pre-Flop fold“ Texas Hold'em. Das bedeutet, dass bei dieser speziellen Variante nur eine Entscheidung pro Hand getroffen werden muss, nämlich entweder direkt All-In zu gehen oder die Karten sofort zu passen. Dieser Zustand schränkt die Komplexität eines normalen Pokerspiels und besonders eines No Limit-Spiels stark ein. Ziel des Experiments ist es aber, eine Abhängigkeit der verschiedenen Gesichtspunkte erkennen zu können, die zur Entscheidungsfindung benötigt werden. Der Suchraum, aus dem der EA seine Strategien auswählt, besteht aus einem 16-teiligen Chromosom, wobei für jede Eingabe reelle Zahlen aus dem Intervall [0, 14) zulässig sind. Im vorliegenden Fall werden folgende Faktoren berücksichtigt:

¹⁰ nicht in Echtzeit

– **Die eigene Handstärke:**

Die 169 verschiedenen Kombinationen der Starthände, „suited“¹¹ und „off-suit“ unberücksichtigt, werden in 13 verschiedene Kategorien mit jeweils 13 Karten, hinsichtlich ihre voraussichtlichen Gewinnwahrscheinlichkeit unterteilt. Die Gruppe mit den besten Händen erhält den Wert 1. Die schlechtesten Hände befinden sich in Gruppe 13.

– **Die Aktion der voraussitzenden Mitspieler:**

Hier wird einfach die Anzahl der Mitspieler gezählt, die bereits in dieser Hand All-In gegangen sind.

– **Die Höhe der Blinds im Turnier:**

Das Turnier besitzt insgesamt 11 Level, in denen die Blinds etappenweise angehoben werden.

– **Die Anzahl der eigenen Chips:**

Hier wird das Verhältnis der eigenen Chips zu den aktuellen Blinds festgelegt.

– **Die eigene Position:**

Hier stehen zwei verschiedene Optionen (früh oder spät) zur Auswahl. Sitzt ein Spieler auf den ersten fünf Plätzen links vom Kartengeber, so sitzt er in früher Position, auf den anderen fünf Plätzen, einschließlich der Kartengeber selbst, sitzt er in später Position.

Für die Simulation der neun am Tisch befindlichen Gegner wurden drei unterschiedliche Strategien verwendet:

1. **Sklansky Basic (SB)** aus [12]:

Bei dieser Strategie werden nur die stärksten Hände gespielt, sobald ein Gegenspieler erhöht hat. Ist noch kein Spieler im Pot, wird eine größere Menge an Händen gespielt.

2. **Sklansky Improved (SI)** aus [12]:

Diese Strategie verhält sich ähnlich der Basic Strategie, sobald ein Gegenspieler erhöht hat. Ist noch kein Spieler im Pot, entscheidet diese Strategie sich allerdings aufgrund des Verhältnisses zwischen Blindhöhe und eigener Chipzahl, ob es sinnvoll ist zu spielen oder zu passen.

3. **Kill Phil Rookie (KPR)** aus [13]:

Auch diese Strategie berücksichtigt zuerst, ob bereits Spieler im Pot sind oder nicht. Allerdings berücksichtigt sie auch, wieviele Spieler noch hinter Einem sitzen und wie weit ein Turnier bereits fortgeschritten ist.

Alle Experimente wurden dann mit neun Gegner durchgeführt, die alle die gleiche Strategie verwenden.

Ergebnisse. Das erste Experiment wurde so durchgeführt, dass der Spieler selber nur seine eigene Handstärke betrachtet, um sich zu entscheiden. Am erfolgreichsten war der Spieler, wenn er die vier stärksten Starthand-Kategorien

¹¹ Eine Starthand heißt suited, wenn ihre beide Karten von der gleichen Farbe sind. Suited Hands haben höhere Gewinnwahrscheinlichkeiten als unsuited Hände, da sie nur drei weitere Gemeinschaftskarten ihrer Farbe benötigen, um einen Flush zu bilden.

spielte, also die etwa 50 statistisch besten Starthände spielte. Wenn man nun mitberücksichtigte, ob bereits ein anderer voransitzender Spieler, im Pot ist. So ergaben sich folgende Kategorien. Gegen die SB-Strategie erwies es sich als erfolgversprechend, wenn man mit den 9 stärksten Kategorien callt, falls noch kein Spieler im Pot ist. Ist aber schon ein Spieler mitgegangen, so sollte man auf jeden Fall folden. Gegen die anderen beiden Strategien war es am erfolgreichsten mit ziemlich allen Starthänden mitzugehen, wenn keiner vor dem Spieler gecallt hat und ein Mitspielen mit anderen sofern man Starthände aus den besten drei Kategorien besitzt. Betrachtungen der Starthände und des Blindlevels, in dem man sich aktuell befindet, so empfiehlt es sich gegen alle Gegner wenig Hände zu spielen, sofern die Blinds noch gering (nur erste Kategorie gegen alle drei Strategien) sind und viele Hände zu spielen, wenn die Blinds hoch sind. Wenn man seine eigene Chipanzahl in Bezug zu der aktuellen Blindgröße setzt, ergaben sich Unterschiede für die drei Strategien. Gegen SB sollte man die fünf besten Kategorien spielen, sofern man nicht mehr allzu viele Chips besitzt und keine Hand, wenn noch genügen Chips vorhanden sind. Gegen SI ist man am erfolgreichsten, wenn man fast alle Hände spielt (12 von 13 Kategorien), sofern man wenig Chips hat und andernfalls nur die zwei besten Kategorien. Gegen KPR sollte man mit vielen Chips ebenfalls nur die zwei besten Kategorien spielen und bei wenig Chips die acht besten Kategorien. Abschließend wurde noch die Sitzposition mitberücksichtigt. Hierbei empfiehlt es sich gegen SB in früher Position mit acht Kategorien zu spielen und weiter hinten sitzend mit den beiden besten Händen. Gegen SI soll man in früher Position jede Hand spielen und später nur die beiden besten Kategorien. Gegen KPR empfiehlt es sich vorne sitzend ebenfalls fast alle Hände (11 von 13) zu spielen und hinten sogar nur die beste Kategorie.

Aus diesen Ergebnissen kann man schon die Bedeutung der unterschiedlichen Strategieparametern sehen. In einem abschließenden Test wurden jeweils 5000 Turniere gespielt und es ergaben sich die Ergebnisse, wie sie in der Tabelle 3 zu sehen sind.

Tabelle 3. Anzahl der Turniersiege, die der beschriebene EA-Ansatz gegen fixe Strategien erzielte. Es wurden 5000 Turniere mit jeweils 9 Gegnern simuliert.

Game Knowledge	SB	SI	KPR
nur Handstärke	1.004	565	319
Hand & vorhergesetzt	3.562	1.095	595
Hand & Turnierlevel	1.997	746	574
Hand & Chipanzahl	1.637	819	534
Hand & Sitzposition	1.417	715	460
Hand & alle Faktoren	4.340	1.165	786

Man sieht ganz klar, dass es am erfolgreichsten ist, wenn man alle Faktoren berücksichtigt. Auch die Strategien unterscheiden sich ziemlich stark voneinander. So gewinnt man gegen SB schon doppelt so oft als durchschnittlich (500)

mit purer Betrachtung seiner eigenen Stärke. Gegen KPR verliert man mit der gleichen Strategie überdurchschnittlich.

Als erstaunliches Ergebnis dieser Untersuchung, das allen bekannten Büchern über Pokern widerspricht, ist das vermehrte Setzen in früher Position. Denn jeder Experte begründet mit dem fehlenden Wissen über starke Hände hinter dem Spieler die Entscheidung, Hände in früher Position vermehrt zu passen. Eine Ursache hierfür kann in dem Design des Testspiels liegen, denn ein All-In ist gewichtiger als ein Call oder leichter Raise. Viele Spieler mit guten Starthänden würden hier noch mitgehen, um mit den Gemeinschaftskarten eine neue Bewertung der Hand durchzuführen. Somit spiegelt dieser Versuchsaufbau noch nicht die volle Komplexität des Hold'em Poker wider, gibt aber gute strategische Anhaltspunkte. Ein letzter Kritikpunkt liegt im Spiel gegen statische Gegner, die ihre Strategie niemals verändern, so dass sie im echten Spielablauf leicht durchschaubar sein werden.

6.3 Einsatz von Reinforcement Learning beim Poker

Die Verwendung eines Reinforcement Learning Algorithmus beschreibt Fredrik A. Dahl in seinem Paper [14]. Reinforcement Learning beschäftigt sich mit der Aufgabe, Strategien zu entwickeln, die es einem System, das mit seiner Umwelt agiert, ermöglichen eine optimale Auswahl von Aktionen zu lernen, um sein Ziel zu erreichen. Das System versucht mittels Bestrafung und Belohnung zu lernen, welche Aktionen von großem Nutzen waren und welche Aktionen sich als weniger sinnvoll herausgestellt haben. Detailliertere Beschreibung des Reinforcement Learning findet sich in [15]. Die Schwierigkeiten, die Dahl erläutert liegen vor allem darin funktionierende Lernverfahren für Spiele mit vollständiger Information auf Poker, also einem Spiel mit unvollständiger Information umzubauen, siehe Kapitel 4.

Aufbau. Auch bei diesem Paper wird wieder eine spezielle Pokervariante betrachtet. Es handelt sich um eine eingeschränkte Limit Hold'em Variante für zwei Spieler. Die Blinds bei diesem Spiel sind immer 1 Chip und es ist Prä-Flop möglich bis zu viermal zu räisen. Sobald ein Spieler passt, gewinnt der andere Spieler den Pot. Callt ein Spieler werden direkt alle fünf Gemeinschaftskarten auf den Tisch gelegt und der Spieler mit der besseren Hand gewinnt den Pot. Es gibt also keine weitere Setzrunde. Somit ist die maximale Potgröße pro Hand 10 Chips. Ein Spieler kann deshalb maximal 5 Chips pro Hand gewinnen und verlieren.

Ein Aufbau über alle Gegnerhände ist zu groß, um alle möglichen Kombinationen abzudecken. Somit muss ein anderes Lösungsmodell entwickelt werden. Fredrik A. Dahl wählt ein neuronales Netz bestehend aus einem einfachen Multi-Layer Perceptron mit 26 Eingangsneuronen zur Kodierung der aktuellen Spielstände und Gegneraktionen, einer versteckten Schicht mit 20 Neuronen und einem Ausgangsneuron, das die aktuelle Wahrscheinlichkeitsverteilung (fold, call, bet) angibt. Im Netz gibt es eine sigmoide Aktivierungsfunktion.

Anschließend passt das Netz mittels Gradientenabstiegsverfahren (Standard-Backpropagation) die Wahrscheinlichkeitsgewichte an das letzte Spielergebnis (Sieg oder Niederlage) an. Führt die gewählte Strategie im Testszenario zu einer erfolglosen Strategie so wird dieser Ansatz durch den Lernalgorithmus bestraft.

Um die zwei lernende Computerspieler besser vergleichen zu können, besteht ein Spiel immer aus zwei Partien bei denen die Spieler in der zweiten Hand genau mit der ersten Hand des Gegners spielen. Somit sollte es sich aus Symmetriegründen eigentlich um ein Spiel ohne Gewinner und Verlierer handeln, somit kann man erfolgreichere Strategien sofort erkennen, da diese Strategien am Ende eines bestimmten Zeitraum einen Gewinn erwirtschaftet haben.

Um Richtwerte als Vergleich zu haben, wurden drei Spielertypen kreiert:

1. Ausbalancierter Spieler:

Dieser Spieler sollte am besten die Gewinne und Verluste abschätzen. Er wurde öfter nach bereits gemachten Erkenntnissen modifiziert.

2. Aggressiver Spieler:

Dieser Spieler ist eine Weiterentwicklung des ausbalancierten Spielers. Er passt nur selten und erhöht oft.

3. Zufälliger Spieler:

Dieser Spieler wählt seine kommende Aktion immer zufällig und ist eigentlich nur als Referenz vorhanden, gegen den jede Strategie gewinnen sollte.

Ergebnisse. Gegen den zufälligen Spieler ist die Gewinnverteilung zu Beginn 0, da der lernende Agent zu Beginn der Trainingsphase ebenfalls zufällig initialisiert ist. Aber der Agent übernimmt schnell die Überhand und erreicht einen kontinuierlichen Gewinn. Zu Beginn verliert der Agent am meisten gegen den aggressiven Spieler, was sich dadurch erklären lässt, dass der Agent zu Beginn noch passt, wenn viel im Pot ist, beispielsweise nach eigenem Raise. Hier lässt sich aber relativ schnell eine Anpassung des Agenten erreichen, so dass es nach ungefähr 20.000 Trainingszyklen einen kontinuierlichen Gewinn des Agenten gibt. Gegen den balancierten Spieler wurden 200.000 Trainingszyklen, ehe ein Gewinn des Agenten erreicht wurde.

Ein interessantes Ergebnis, das bei diesem Experiment herauskam, ist, dass der trainierte Agent zu Beginn des Spiels entweder passt oder erhöht und nie einfach nur mitgeht. Dieser Fakt lässt sich auch in vielen Pokerbüchern nachlesen. Die Autoren begründen es damit, dass man entweder eine gewinnfähige Hand, mit der man möglichst viel im Pot gewinnen will, oder man hat eine schlechte Hand, dann sollten überhaupt keine Chips bezahlt werden.

Abschließend lässt sich auch hierbei feststellen, dass diese Strategien irgendwann durchschaubar werden. Wobei sie für sehr gute und sehr schlechte Hände noch einfach zu handhaben sind. Problematisch ist vor allem die Abgrenzung von mittleren Händen in Erhöhen oder Passen.

6.4 Einsatz von Fuzzy beim Poker

Die Verwendung eines Fuzzy-Systems beschreiben Takehisa Onisawa und Tomoaki Yano in ihrem Paper [16]. Fuzzy-Systeme verwenden unscharfe Logik-

Operatoren der sog. Fuzzy-Logik, mit denen sie versuchen, nicht zu präzisierende umgangssprachliche Sachverhalte in mathematisch-logische Zusammenhänge zu modellieren. Einen guten Überblick über Fuzzy-Systeme und die zugrundeliegende Logik lässt sich in [17] finden. Hierbei berücksichtigen sie sehr viele Input-Faktoren, die sie dann auf immer auf sieben Fuzzy-Mengen des folgenden Typs umrechnen, wie in Abbildung 1 zu sehen ist.

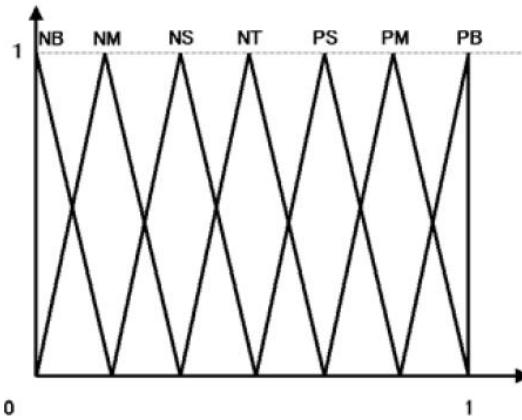


Abbildung 1. Aufbau der im Paper verwendeten Fuzzymengen mit ihren Zugehörigkeitsfunktionen.

Aufbau. Die in diesem Paper verwendete Variante ist ein Heads-up Limit Texas Hold'em mit Blinds von 10 in den ersten beiden Setzrunden und 20 in den letzten beiden Setzrunden, wobei Heads-up bedeutet, dass das Fuzzy-System gegen einen menschlichen Gegenspieler antritt. Das Fuzzy-System ist weiterhin in zwei Bereiche unterteilt, der erste versucht die Strategie des Gegners zu lesen, wohingegen der zweite die eigenen Karten beurteilt und die Odds 4.2 berechnet. Als Input dienen dem ersten Teil:

1. Der Verlauf des Spiels:

Hier wird die aktuelle Setzrunde (linguistische Variablen von *eins* bis *vier*), die Phase in der Setzrunde (linguistische Variablen von *eins* bis *sechs*¹²) und ob der Gegenspieler in der ersten Phase callt (linguistische Variable *true* oder *false*) berücksichtigt.

2. Der Status des Tisches:

Hier wird berücksichtigt, welcher Spieler am Button sitzt (linguistische Va-

¹² wegen der Möglichkeit pro Runde viermal erhöhen zu können, dazu sind zwei calls möglich

riable *menschlich* oder *System*), wie oft in der aktuellen Runde erhöht wurde (linguistische Variablen von *null* bis *vier*), wie viele Chips noch zu setzen sind (mittels der Zugehörigkeitsfunktion in Abb. 1, wobei NB=min., NM=winzig, NS=wenig, NT=mittel, PS=mehr, PM=viel und PB=max ist) und die Größe des Pots (dieselbe Zugehörigkeitsfunktion).

3. Der Spielertyp des Gegners:

In diesem Faktor wird die Aggressivität¹³ (mittels der Zugehörigkeitsfunktion in Abb. 1, wobei NB=sehr passiv, NM=passiv, NS=etwas passiv, NT=neutral, PS=etwas aggressiv, PM=aggressiv und PB=sehr aggressiv ist) und Strenge¹⁴ (mittels der Zugehörigkeitsfunktion in Abb. 1, wobei NB=sehr locker, NM=locker, NS=etwas locker, NT=neutral, PS=etwas streng, PM=streng und PB=sehr streng ist) des menschlichen Spielers bewertet.

4. Die Gemeinschaftskarten:

Hier werden Wahrscheinlichkeiten der Hände berechnet, die mit den eigenen Karten und den schon vorhandenen Gemeinschaftskarten noch möglich sind. Nach dem Flop werden 1000 Züge über die möglichen 47 Karten gemacht und die sich ergebenen Hände gezählt, nach dem Turn werden alle 46 möglichen Karten nacheinander berücksichtigt und die fertigen Hände gezählt. Nach dem River ist die Hand fest und erhält den Wert 1. (linguistische Variablen sind die unterschiedlichen Hände, also *Höchste Karte*, *Paar*, *Zwei Paare*, *Drilling*, *Straight*, *Flush*, *Full House*, *Vierling* und *Straight Flush*). Als zweiter Wert fließt die gegnerische Hand in diesen Faktor ein. Um seine Hand berechnen zu können, werden hier auch seine möglichen verdeckten Karten simuliert.

Für den zweiten Teil kommen neben den o.g. Faktoren noch folgende hinzu:

5. Die Odds:

Hierbei werden die Odds berechnet und ebenfalls mittels der Zugehörigkeitsfunktion in Abb. 1, wobei NB=darf nicht, NM=sollte nicht, NS=muss nicht, NT=könnte, PS=darf, PM=sollte und PB=muss ist, dargestellt. Zusätzlich fließt noch die Gewinnwahrscheinlichkeit in diesen Faktor ein, die die errechneten Werte des vorhergehenden Faktor *Gemeinschaftskarten* mittels Formel mitverwendet.

6. Die verdeckten Karten:

In diesem Faktor wird auf die erste Setzrunde eingegangen, bei dem die verdeckten Karten des Gegners völlig unbekannt sind und somit nur an Hand der Anzahl an Erhöhungen und der eigenen Handstärke über die zumachende Aktion entschieden werden kann. Die eigenen Starthände werden hierbei in neun verschiedene Gruppen unterteilt und mittels Formel wird entschieden, ob gecallt wird oder nicht.

Mit diesen verschiedenen Eingaben wird nun eine erste Fuzzy-Regelmenge aufgebaut, aus der dann die voraussichtliche Strategie des Gegenspielers erkannt

¹³ das Verhältnis der Erhöhungen zu calls

¹⁴ beim Pokern auch *Tightness* genannt, das Verhältnis der gespielten Hände zu folds, siehe auch Abschnitt 4.3

werden soll. Anschließend fließt diese Strategie zusätzlich mit in die zweite Regelmenge ein, mit der die eigene Strategie festgelegt wird. Als unterschiedliche Strategien stehen die folgenden zur Auswahl¹⁵:

1. Normaler Fold, da schlechte Odds
2. Schwacher Fold, da Gegner als stärker erwartet
3. Normaler Call
4. Weitere Karte, um gute Hand noch zu verbessern
5. Fangen, um den Draw zur guten Hand zu machen
6. Slowplay, um Gegner zu verwirren
7. Normaler Bet
8. Schützender Bet, damit Gegner sich nicht verbessert
9. Echter Bluff
10. Semibluff, mit der Chance eine gute Hand zu bekommen

Abschließend werden nun die Strategien noch mit Expertenwissen verifiziert, so macht Slowplay in der letzten Setzrunde keinen Sinn mehr und die Wahrscheinlichkeit jetzt die Slowplay-Strategie anzuwenden wird auf Null gesetzt. Die ausgewählte Strategie bestimmt nun die Aktion, die das System macht.

- *Fold*, bei Normaler Fold oder Schwacher Fold
- *Call*, bei Normaler Call, Weitere Karte, Fangen oder Slowplay
- *Bet*, bei Normaler Bet, Schützender Bet, Echter Bluff oder Semibluff

Ergebnisse. Um Vergleichswerte zu erhalten wurden drei verschiedene Systeme gegen 18 menschliche Gegenspieler getestet. Das erste System *divided tree*-System genannt, verwendet alle oben beschriebenen Funktionen. Als Vergleichssysteme wurden ein *odds*-System, das nur auf die eigenen Hände schaut, sowie ein *single tree*-System, das nur den Fuzzy-Baum zur Gegneranalyse benutzt aufgebaut. Gemessen wurden nun die Erfolge, die die einzelnen Systeme erzielten, wobei in jedem Duell beide Spieler zu Beginn 5000 Chips besaßen und die menschlichen Gegenspieler das Pokerspiel jederzeit beenden durften.

Tabelle 4. Gewinne, die die unterschiedlichen Systeme gegen die menschlichen Gegenspieler erzielten (negative Gewinne entsprechen Verlusten).

System	Gewinn	Spielanzahl	durchschnittlicher Gewinn
Odds	2.160	1.785	1,21
Single Tree	-2.680	794	-3,38
Divided Tree	695	700	0,99

Wie die Tabelle 4 zeigt, erzielen das *odds*-System und das *divided tree*-System Gewinne gegen die menschlichen Gegenspieler. Eine Begründung für die etwas größeren Gewinne des weniger komplexen *odds*-System liegt im Aufbau

¹⁵ Eine ausführliche Erklärung ist im Paper [16] zu finden.

des *divided tree*-System. Denn hier es ist nicht dafür konzipiert, Gewinne zu maximieren, sondern sie nur zu machen.

Als weiterer Untersuchungspunkt galt die Anwendung insbesondere der Strategien, die den menschlichen Gegenspieler bewusst täuschen sollten. Diese Täuschungsstrategien waren Slowplay, mit dem der Gegenspieler dazu gebracht werden sollte, den Pot noch zu erhöhen und die starke Hand des Systems mit einem größeren Gewinn zu belohnen, und echter Bluff, mit dem dem Gegenspieler eine sehr starke eigene Hand vorgetäuscht werden soll und dieser so zum Passen gebracht werden soll. Untersuchungen der Spiele zeigte, dass das *divided tree*-System genauso häufig und ähnlich erfolgreich wie die menschlichen Gegenspieler operierte, währenddessen die beiden anderen Systeme nur halb so häufig und auch deutlich erfolgloser versuchten den menschlichen Gegenspieler zu täuschen.

7 Zusammenfassung

Die Voraussagen mittels NN führen schon zu guten Ergebnissen. Abweichungen von der Erwartung können den Spieler vor starken Händen der Gegner warnen. Bei dem Versuch, die Gegner zu blaffen, kann es dann angewendet werden, um zu wissen, wie wahrscheinlich man gecallt wird. Der Abschnitt über EAs zeigt, dass es viele Parameter gibt, die bei der Modellierung eines guten Pokeragenten berücksichtigt werden sollten. Im Abschnitt 6.3 über die Anwendung des Reinforcement Learning sieht man, dass Anpassung auf alle Spielertypen möglich ist und als weiteres Ergebnis, dass es zu Beginn eines Spiels nicht sehr sinnvoll ist, einfach nur mitzugehen. Der Abschnitt über Fuzzy-Systeme verwendet schon sehr komplexe Eingaben und kommt zu ansprechenden Ergebnissen. Verbesserungen hierbei wären allerdings noch differenziertere Beurteilungen der Hände, denn beispielsweise ist ein Paar nicht gleich einem Paar des Gegners und eine speziellere Unterteilung der Paare könnte die Erfolge erhöhen.

Bei allen Betrachtungen sieht man allerdings die starke Komplexität und Variabilität des Spiels Poker, denn alle hier vorgestellten Paper behandeln nur vereinfachte Varianten der gewöhnlich gespielten Pokervarianten. Bis ansprechendes Pokerverhalten für die nun gebräuchlichste, beliebteste und auch bei der Weltmeisterschaft gespielte Variante Texas Hold'em No Limit mit mehr als einem Gegenspieler am Tisch, steigenden Blinds und vielen anderen Faktoren gefunden wird, die erfolgreich mitspielen kann, wird noch einige Zeit vergehen. Allerdings lassen sich die bisher gemachten Erkenntnissen und Erfahrungen sinnvoll erweitern.

Literatur

- [1] Huizinga, J.: *Homo Ludens. Vom Ursprung der Kultur im Spiel.* Amsterdam: Pantheon (1939)
- [2] Caro, M.: *Caro's Book of Poker Tells.* Cardoza (2003)
- [3] Harrington, D.: *Harrington on Hold'em.* PREMIUM POKER PUBLISHING (2007)

- [4] Sklansky, D.: No-Limit Hold'em: Theorie und Praxis. PREMIUM POKER PUBLISHING (2007)
- [5] von Neumann, J., Morgenstern., O.: Theory of Games and Economic Behavior. Princeton University Press (1944)
- [6] Findler, N.V.: Studies in Machine Cognition Using the Game of Poker. Communications of the ACM 20, no.4 (1977) Seiten 230–245
- [7] Rojas, R.: Theorie der neuronalen Netze – eine systematische Einführung. Springer Verlag, Berlin (1996)
- [8] Davidson, A.: Using Artificial Neural Networks to Model Opponents in Texas Hold'em. CMPUT 499 - Research Project (1999)
- [9] Papp, D.: Dealing with Imperfect Information in Poker (1998)
- [10] Engelbrecht, A.P.: Computational Intelligence An Introduction. Wiley-VCH Verlag, Weinheim (2002)
- [11] Carter, R.G., Levine, J.: An Investigation into Tournament Poker Strategy Using Evolutionary Algorithms. Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games (CIG 2007) (2007) Seiten 117–124
- [12] Sklansky, D.: Tournament Poker for Advanced Players. Two Plus Two Publishing, Las Vegas, NV (2004)
- [13] Rodman, B., Nelson, L.: Kill Phil - The Fast Track to Success in No-Limit Hold'em Poker Tournaments. Huntinton Press, Las Vegas, NV (2005)
- [14] Dahl, F.A.: A Reinforcement Learning Algorithm Applied to Simplified Two-Player Texas Hold'em Poker. EMCL '01: Proceedings of the 12th European Conference on Machine Learning (2001) Seiten 85–96
- [15] Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. The MIT Press (1998)
- [16] Onisawa, T., Yano, T.: Construction of Poker Playing System Considering Strategies. CyberGames '06: Proceedings of the 2006 international conference on Game research and development (2006) Seiten 121–128
- [17] Dubois, D., Prade, H.: The Place of Fuzzy Logic in the AI. Lecture Notes in Artificial Intelligence **16** (1997) Seiten 9–20
- [18] Barone, L., While, L.R.: Evolving Adaptive Play for Simplified Poker. Proceedings of IEEE International Conference on Computational Intelligence (ICEC-98) (1998) Seiten 108–113
- [19] Barone, L., While, L.R.: An Adaptive Learning Model for Simplified Poker Using Evolutionary Algorithms. Proceedings of the Congress of Evolutionary Computation (GECCO-1999) (1999) Seiten 153–160
- [20] Barone, L., While, L.R.: Adaptive Learning for Poker. Proceedings of the Genetic and Evolutionary Computation Congress (GECCO-2000) (2000) Seiten 566–573
- [21] Billings, D., Papp, D., Schaeffer, J., Szafron, D.: Poker as a Testbed for Machine Intelligence Research. Advances in Artificial Intelligence (Mercer R. and Neufeld E. eds.) (1998) Seiten 1–15
- [22] Billings, D., Papp, D., Schaeffer, J., Szafron, D.: Opponent Modeling in Poker. Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) (1998) Seiten 493–498
- [23] Billings, D., Peña, L., Schaeffer, J., Szafron, D.: Using Probabilistic Knowledge and Simulation to Play Poker. AAAI/IAAI (1999) Seiten 697–703
- [24] Billings, D., Peña, L., Schaeffer, J., Szafron, D.: Learning to Play Strong Poker. ICML-99 Workshop Machine Learning in Game Playing, J. Stefan Institute, Slovenia (1999)
- [25] Kendall, G., Wilddig, M.: An Investigation of an Adaptive Poker Player. Australian Joint Conference on Artificial Intelligence (2001) Seiten 189–200

CI & KI bei Browserspielen

Daniel Haus

Seminar Computational Intelligence bei Computerspielen, SS 2007
Lehrstuhl 11
TU Dortmund

Zusammenfassung Analog zur rasanten Entwicklung von Computerspielen und parallel der des World Wide Web – sowie der entsprechenden Server- und Webbrowser-Technologien – sind in den letzten Jahren unzählige Browser-basierte Spiele erschienen und erfreuen sich zunehmender Beliebtheit bei den Konsumenten. Im Rahmen des Seminars „Computational Intelligence bei Computerspielen“ soll in dieser Arbeit erörtert werden, was im Allgemeinen unter Browser-Spielen verstanden wird, wo CI & KI in diesem Genre bereits Anwendung findet und wie man Browser-Spiele mithilfe moderner CI/KI-Methoden interessanter gestalten und den Spielspaß fördern kann.

1 Was sind Browser-Spiele?

Spätestens seit der sechsten Generation kann man Webbrowser als multifunktionale und vielseitig erweiterbare Plattform für praktisch beliebige Software-Anwendungen verstehen und einsetzen. Dank verbreiteter Plugins wie dem Java Runtime Environment oder dem Adobe Flash Player können Webbrowser heutzutage weitaus reichere Medien darstellen als nur Hypertext. Ob 3D-Grafik, Sound oder Videos in DVD-Qualität oder besser, all solche Inhalte lassen sich in einem modernen Webbrowser wiedergeben. Auch die Spieleplattform Unity [Uni07] soll hier kurz genannt werden, die unter anderem eine vollständige, portable und top-moderne 3D-Spiele-Engine in Form eines Browser-Plugins liefert.

Mittlerweile existieren sogar Emulatoren, die auf Java- (siehe Abbildung 1 bzw. [Eri07]) oder auf Flash-Basis [SW07] im Browserfenster altbekannte 8-Bit-Rechner binärkompatibel emulieren. Somit sind alle für diese Architekturen erhältlichen Spiele in Webbrowsersn lauffähig. Die übrigen Spiele lassen sich aber theoretisch als Java-Applet oder mithilfe von sonstigen Plugins implementieren. Selbst wenn man völlig auf Plugins verzichtet, findet man flüssig spielbare Implementierungen von Jump-and-Run-Spielen (siehe Abbildung 2 und [Mak07]), Tetris, Karten- und Schachspielen sowie Grafikadventures – alle auf reiner Browser-Basis (dynamisches HTML/JavaScript).

Die Frage, worin sich nun Browser-basierte Spiele von allen anderen Sorten von Spielen – insbesondere unter dem Gesichtspunkt der künstlichen Intelligenz – unterscheiden, ist auf den ersten Blick nicht leicht zu beantworten, da sich theoretisch jede Art von Computerspiel, die auf gängigen Personal Computern

oder Spielekonsolen lauffähig ist, auch als Browser-Plugin implementieren lässt. Entsprechend findet man hier auch die für die jeweiligen Genres üblichen CI-Ansätze.

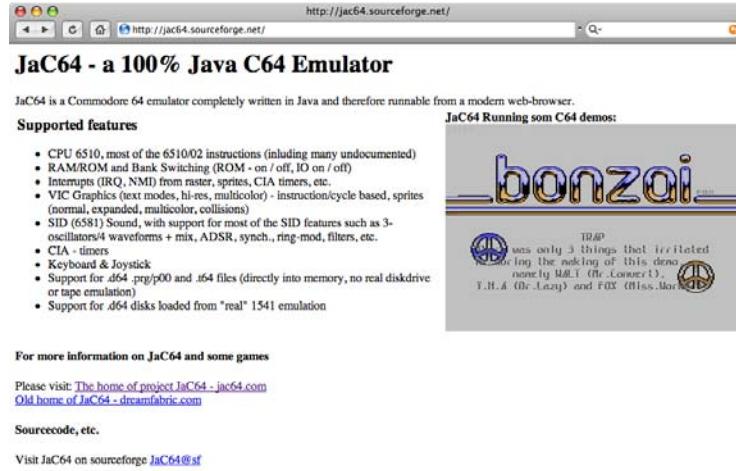


Abbildung 1. „JaC64“, ein vollwertiger, binärkompatibler Commodore 64 Emulator in Form eines Java Applets, eingebettet in eine einfache Website.

1.1 Definition „Browser-Spiel“

Die Wikipedia definiert – vermutlich als erste und zu diesem Zeitpunkt auch einzige Enzyklopädie – das Wort „Browser-Spiel“ wie folgt [Wik07a]:

Ein Browserspiel (englisch browser-based game oder browser game) ist ein Computerspiel, das einen Webbrowser als Benutzerschnittstelle benutzt. In der einfachsten Form erfolgt die Berechnung des Spielgeschehens vollständig auf den Servern des Spiele-Anbieters, wodurch keine Software-Installation auf dem Rechner des Spielers erforderlich ist.

Es existiert eine besondere Art von Computerspielen, die praktisch *ausgeschließlich* in Webbrowsern läuft – und in der Regel auch ältere Browser-Generationen unterstützt. Es handelt sich um eine spezielle Form von – meist rundenbasierten – Mehrspieler-Strategiespielen (MMOG¹) mit Rollenspiel-Elementen,

¹ MMOG: „Massen-Mehrspieler-Online-Gemeinschaftsspiel“ bzw. englisch „Massively Multiplayer Online Game“

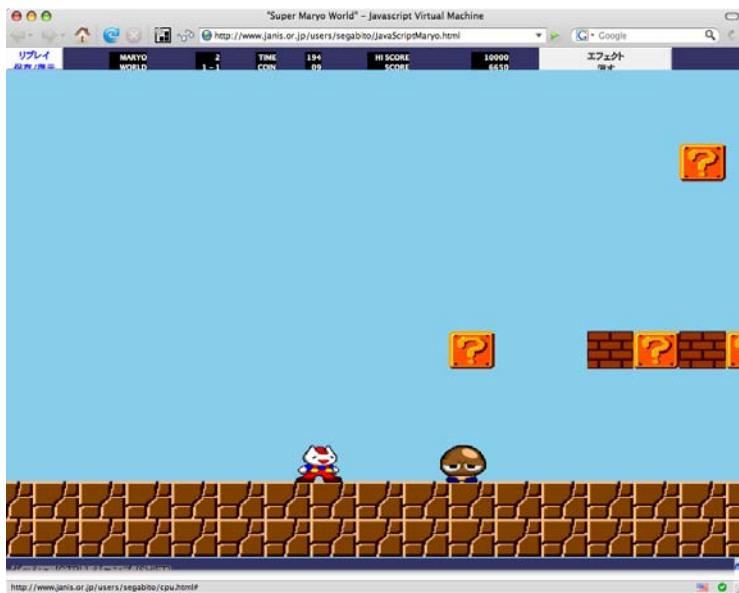


Abbildung 2. „Super Maryo World“ (sic!), eine Imitation des ähnlich benannten Nintendo-Spiels in JavaScript und HTML, ist erstaunlich flüssig spielbar und kommt völlig ohne Plugins aus.



Abbildung 3. Ein First-Person-Shooter im Unity-Webbrowser-Plugin

welche als Medium auf reinen Hypertext in seiner ursprünglichsten Erscheinung² mit höchstens ein paar kleinen Grafiken setzt. An solchen Spielen nehmen meist Hunderte, Tausende manchmal sogar Hunderttausende von Spielern gleichzeitig teil. Konkrete Zahlen sind unter [Wik07a] nachzulesen.

Aus technischer Sicht laufen sämtliche spieltechnischen Berechnungen zentral auf dem Server³. Der Browser sorgt lediglich für den Zugriff von außen, die Darstellung der Spielsituation und das Senden der Steuerbefehle zurück zum Server. Wir werden uns im Folgenden auf diese Sorte von Spielen beschränken, da sie die ursprünglichste Erscheinung von Browser-Spielen darstellt⁴, und vor allem deswegen, weil Konsumenten von Computerspielen im Allgemeinen den Begriff „Browser-Spiel“ mit dieser Art von Software assoziieren.

1.2 Szenarien

Obgleich thematisch verschiedenste Formen von Browser-Spielen existieren – das schließt Weltraumabenteuer ebenso ein wie Fantasy-Rollenspiele, Drogendealer- und Wirtschaftssimulationen – sind sich die Spiele in Bezug auf Regeln, inneren Abläufen und Steuerelementen her auffallend ähnlich.

Navigiert man nun ein Raumschiff der Galaxy-Klasse durch ein Star Trek-Universum bei „Star Trek Universe“ [Jak07], eiserne Einzelkämpfer durch eine mittelalterliche Fantasy-Welt in „Demonlords“ oder einen Straftäter durch internationale Großstädte beim kommerziell motivierten (aber deswegen nicht zwangswise professioneller umgesetzten) Spiel „Drogendealer“ – Logik und Funktionalität sind praktisch gleich, auch wenn sich Sprache, Darstellung und Zielgruppe deutlich unterscheiden können. In manchen Fällen werden sogar lediglich Texte, Farben und Grafiken eines Spiels ausgewechselt, um ein neues Spiel zu erhalten, während der Programm-Code größtenteils unverändert übernommen wird. Aus diesem Grund können wir uns im Folgenden auf die Betrachtung eines konkreten Spiels beschränken, sämtliche Ideen, Vorschläge und Methoden lassen sich entsprechend auf andere Spiele des Genres übertragen.

1.3 Beispiel: Star Trek Universe

Bei „Star Trek Universe“ handelt es sich um ein rundenbasiertes Strategiespiel im Star Trek-Universum. Ein Spieler, im Spieljargon „Siedler“ genannt, beantragt im ersten Schritt ein Kolonieschiff, sucht sich dann einen besiedelbaren Planeten, zu dem er sein neues Schiff navigiert, und kolonisiert diesen anschließend. Dann erarbeitet er sich im Laufe des Spiels verschiedene Einrichtungen, wie beispielsweise Bauernhöfe, Fabriken, Minen und Raffinerien, sowie

² HTML, eventuell gestützt von etwas JavaScript und CSS

³ In den meisten Fällen dient die verbreitete (und kostenlose) Kombination von Linux, Apache httpd, MySQL und PHP als Basis – hin und wieder, gerade bei älteren Spielen, auch Perl anstelle von PHP.

⁴ Als Spezialfälle gelten darüber hinaus die heute eher seltenen Foren-, Chat- und E-Mail-Spiele, auf die hier nicht weiter eingegangen wird.

wirtschaftliche Güter und später Raumschiffe, mit denen er dann weitere Kolonien gründen, Raumstationen errichten, Kriege führen und das Universum erobern kann. Es existieren Warenbörsen, auf denen mit anderen Siedlern gehandelt werden kann, Glücksspiel, aber auch Forschung ist möglich, die dem Spieler weitere Möglichkeiten eröffnet.

Bei jedem Rundenwechsel werden verbrauchte bzw. geförderte Rohstoffe und Energieeinheiten sowie Zu- und Abwanderungen der Bevölkerung und der Verbrauch von Nahrungsmitteln abgerechnet. Rundenwechsel finden bei Star Trek Universe fünf mal am Tag zu fest vorgegebenen Zeiten, 12:00, 15:00, 18:00, 21:00 und 24:00 Uhr, statt.

Neben den Siedlern existieren Nicht-Spieler-Charaktere (NPC, „Non-Player-Character“), die einerseits technische und inhaltliche Fragen der Spieler beantworten, andererseits aber auch beliebig ins Spielgeschehen eingreifen können, um die Interessen der Spielleiter und Administratoren umzusetzen. Entgegen dem, was der Name verspricht, handelt es sich bei den NPCs in Star Trek Universe nicht um computergesteuerte Agenten, sondern vielmehr um Einheiten, die von Spielleitung und Administratoren gleichermaßen gelenkt werden können, ohne direkt einem realen Spieler zugeordnet zu werden. Weder in der umfassenden Dokumentation, noch im Spiel selbst findet man Andeutungen oder Hinweise auf Anwendung irgendeiner Form von Computational Intelligence.

2 CI & KI: Status quo

2.1 Keine CI/KI in aktuellen Browser-Spielen

Die Suche nach dem Einsatz von künstlicher Intelligenz in Browser-Spielen allgemein ist mangels Informationsquellen eine eher mühsame Angelegenheit und führt zu keinem anderen Ergebnis, als dass in den heutigen Browser-Spielen künstliche Intelligenz, in welcher Form auch immer, nicht existiert – mit eventuell einer minimalen Ausnahme, auf die etwas später noch kurz eingegangen wird. Das hat verschiedene, aber leicht nachvollziehbare Gründe. Zunächst handelt es sich ja bei dem hier betrachteten Genre um Massen-Mehrspieler-Spiele mit mindestens Hunderten von Spielern – genügend natürliche Intelligenz, um im Rahmen eines solchen Spiels ein halbwegs lebendiges Universum zu simulieren.

Computergesteuerte Gegner, die intelligent genug, aber nicht zu intelligent sind⁵, sind in der Regel signifikant aufwändiger und komplizierter zu realisieren als die einfachen Steuerelemente, die man Spielern über das Internet zur Verfügung stellen muss, damit diese selbst gegenüber ihren Mitspielern als Gegner agieren können. Diese Tatsache – zusammen mit dem Fakt, dass es sich bei den Entwicklern von Browser-Spielen normalerweise um Jugendliche und/oder Hobbyprogrammierer handelt, die technisch wenig versiert sind, über keine oder nur minimale Budgets verfügen und in kleinen Teams mit beschränkter Zeit oder

⁵ Wie in einigen anderen Vorträgen des Seminars angesprochen wurde, senken leicht durchschaubare Gegner den Spielspaß ebenso wie unbesiegbare.

gar im Alleingang arbeiten – erklärt alleine schon plausibel, warum künstliche Intelligenz in diesem Genre schwer bis überhaupt nicht zu finden ist.

Als weiterer Grund schließt sich die zwischenmenschliche Kommunikation an, die für Spieler von Browser-Spielen einen gravierenden Faktor darstellt – werden doch in sämtlichen Spielen der Gattung umfassende Möglichkeiten zur verbalen, schriftlichen Kommunikation zwischen den Mitspielern bereitgestellt und von diesen auch intensiv genutzt. Zwar gibt es schon seit Mitte der 60er Jahren Bestrebungen Algorithmen zu entwickeln, die eine natürlichsprachige Kommunikation zwischen Mensch und Maschine auf eine Weise ermöglichen, dass der dem Ein-/Ausgabegerät gegenüberstehende Mensch den Eindruck gewinnt, er hätte es mit einem halbwegen „intelligenten“ Gesprächspartner zu tun [Wei66]. Jedoch erscheint es mehr als fragwürdig, ob aktuell verfügbare Lösungen eine solche Illusion über mehrere Stunden am Stück aufrecht erhalten und damit den Spielspaß steigern können.

Schließlich bieten insbesondere für kommerziell motivierte Anbieter von Browser-Spielen menschliche Spieler den nicht zu unterschätzenden Vorzug, dass einige von ihnen, anders als simulierte Spieler, Premium-Zugänge mieten und Werbung konsumieren.

2.2 Andromeda

Wie im letzten Abschnitt angedeutet, existiert scheinbar eine Ausnahme, also ein Browser-Spiel, welches laut seinem Erschaffer tatsächlich künstliche Intelligenz einsetzt. Das Spiel nennt sich „Andromeda“, befindet sich allerdings noch in einer Test- und Entwicklungsphase mit etwa 40 Spielern und wird voraussichtlich nicht vor dem Jahr 2008 veröffentlicht werden (dann unter [Kra07a], weitere Informationen sind bereits unter [Kra07b] zu finden). Bei Andromeda handelt es sich um eine Mischung aus Rollen- und Strategiespiel in einem Weltraum-Szenario. Es existiert, so der Autor, ein „virtueller Support-Mitarbeiter BOB“, der sowohl „Support-Anfragen“ der Spieler beantwortet, als auch diese unterhält und zur Atmosphäre beitragen soll. Neben „BOB“ existieren NPC-Gegner – in diesem Fall tatsächlich Computer-gesteuerte Agenten, welche auf „Einstellung und Verhalten des Spielers“ reagieren. Welche konkreten KI-Methoden bei Andromeda zum Tragen kommen, wird leider vom Autor nicht preisgegeben, er verweist lediglich auf „eine Mischung aus KI+ELIZA“ und auf die „Karlshorstratte“, eine Internet-basierte Variante von ELIZA, die leider nicht mehr erreichbar ist.

2.3 ELIZA

Das im Jahr 1966 vom Informatik-Pionier Joseph Weizenbaum entwickelte Programm „ELIZA“ sollte die „Möglichkeiten der Kommunikation zwischen einem Menschen und dem Computer über natürliche Sprache aufzeigen“ [Wik07c]. Weizenbaum wählte als Charakter für den virtuellen Gesprächspartner eine Psychotherapeutin mit der Begründung, dass diese dem Patienten gegenüber keinerlei Wissen über die Welt zeigen muss, ohne dabei an Glaubwürdigkeit einzubüßen.

ELIZA analysiert und transformiert eingegebene Sätze auf Basis von Dekompositions- und Transformationsregeln, welche in Abhängigkeit von in der Eingabe gefundenen Schlüsselwörtern ausgewählt werden. Die grundsätzliche Verfahrensweise von ELIZA ist sehr einfach. Zunächst wird die Eingabe an Kommata oder Punkten⁶ aufgeteilt – sofern vorhanden. Werden im ersten Teil (von links nach rechts gesehen) ein oder mehrere Schlüsselwörter gefunden, so werden die übrigen Teile verworfen und nur der ermittelte Teilsatz weiterverarbeitet. Bei mehreren gefundenen Schlüsselwörtern ist allein dasjenige von Bedeutung, welches die höchste Priorität aufweist, wobei die Prioritäten der Schlüsselwörtern nach einer vorgegebenen Rangliste ermittelt werden. Wird kein Schlüsselbegriff erkannt, so wird derjenige Teil der Eingabe behalten, der am weitesten rechts steht, und alle Zeichen links davon werden verworfen. Resultierend verarbeitet ELIZA lediglich einzelne Phrasen oder Sätze.

Regeln in ELIZA sind nicht fest mit dem Programmcode verschweißt, sondern liegen in Form von Daten vor, die nach Belieben erweitert oder ausgetauscht werden können. So existieren Regelsätze neben Englisch auch für Walisisch und die deutsche Sprache. Die Syntax der Regeln erinnert stark an LISP, was naheliegend erscheint, da ELIZA selbst in MAD-SLIP (einer LISP-artigen Programmiersprache, ebenfalls von Weizenbaum entwickelt) implementiert wurde. Dekompositionsregeln sind geordnete Tupel von Wörtern und Ganzzahlen. Beim Testen, ob eine Regel zutrifft, wird geprüft, ob die Wörter der Eingabe mit denen der Regel übereinstimmen, wobei die Position signifikant ist. Zahlen dienen als Platzhalter. Die Zahl „0“ passt auf eine beliebige Anzahl beliebiger Wörter, während jede andere positive Ganzzahl die Anzahl der beliebigen Wörter repräsentiert, auf die Sie passt.

Transformationsregeln sind ähnlich aufgebaut, aber hier referenzieren Zahlen die gefundenen Wörter der angewendeten Dekompositionsregel. Eine „2“ bezeichnet also entweder das zweite Wort der Regel, oder, falls in der Regel an zweiter Stelle eine Zahl steht, den Teil der Eingabe, der auf diese Zahl passt.

So wird beispielsweise eine Eingabe „It seems that you hate me“ durch eine passende Regel „(0 YOU 1 ME)“ in die vier Bestandteile 1: „It seems that“, 2: „you“, 3: „hate“ und 4: „me“ zerlegt und dann durch eine Transformationsregel „(WHAT MAKES YOU THINK I 3 YOU)“ zusammengesetzt, so dass die Antwort „WHAT MAKES YOU THINK I HATE YOU“ lautet.

2.4 Wie kann CI/KI Browser-Spiele grundsätzlich verbessern?

Wie wir festgestellt haben, besteht im Grunde genommen kein Bedarf an computergesteuerten Gegnern in Browser-Spielen. Folglich scheint es, dass die in anderen Typen von Computerspielen verbreiteten Einsatzmöglichkeiten für CI & KI in den hier speziell betrachteten MMOG-Browser-Spielen nicht so einfach sinnvoll anwendbar sind.

⁶ Die Verwendung des Fragezeichens als Satzzeichen war auf der damaligen Zielplattform – dem *MAC time-sharing system* am MIT (einer IBM 7094) – nicht möglich, da diesem eine Systemfunktion zugeordnet war.

Erwägt man allerdings die Frage, wie sich künstliche Intelligenz nicht *gegen* den Spieler, sondern vielmehr *den Spieler unterstützend* einsetzen lässt, und wie man damit offensichtliche Defizite, die die Spiele des Genres in der Regel gemein haben, kompensieren kann, so eröffnen sich durchaus interessante Perspektiven. Um hierauf detaillierter einzugehen, werden wir zunächst einen genaueren Blick auf die Mängel von Browser-Spielen werfen, die sich möglicherweise mithilfe von CI- & KI-Methoden ausbessern lassen können.

3 Problematiken in Browser-Spielen

3.1 Navigation der Einheiten

Spätestens nach der ersten Anmeldung trifft man in Browser-Spielen meist auf bezüglich intuitive Benutzbarkeit verbesserungswürdige Bedieneroberflächen und -Konzepte. Hierauf soll in diesem Abschnitt kurz eingegangen werden.

In dem hier exemplarisch betrachteten Spiel „Star Trek Universe“ muss der Spieler einen freien Planeten finden, sein Kolonieschiff durch das zweidimensionale, in quadratischen Kacheln organisierte Universum dorthin navigieren und schließlich den Planeten besiedeln. Das Kolonieschiff hat zu Beginn einen geringen Energievorrat, der unangenehm schnell erschöpft ist. Danach bekommt das Schiff zu jedem Rundenwechsel – also tagsüber im Drei-Stunden-Takt – jeweils acht Energieeinheiten, von denen sofort einige für den Betrieb der Schiffssensoren, Replikatoren etc. abgerechnet werden. Die restlichen Energieeinheiten kann man dann zur Fortbewegung verbauchen. Um ein Raumschiff ein Feld nach oben, unten, links oder rechts auf der Sternenkarte zu bewegen, benötigt man einen Energiepunkt, der beim Klick auf das entsprechende Steuerelement sofort abgezogen wird. Nebel, Asteroidenfelder und sonstige Hindernisse kosten entsprechend mehr Energieeinheiten und können zudem Schaden am Schiff verursachen.

Liegt der Zielplanet mehrere dutzend Felder von den Startkoordinaten des Kolonieschiffs entfernt, kann man sich leicht vorstellen, wie mühsam und demotivierend der Einstieg in das Spiel empfunden werden kann: fünf oder sechs mal klicken, dann wieder drei (bzw. einmal am Tag zwölf) Stunden auf die nächste Runde warten, um dann das Schiff weitere fünf bis sechs Felder zu bewegen und anschließend wieder zu warten. Ein Bildschirmfoto der beschriebenen Steuerelemente ist in Abbildung 4 zu sehen. Oben, unter dem Punkt „Navigation“, findet man die Koordinaten, an denen sich das Schiff momentan befindet und darüber, darunter und daneben die vier Knöpfe, um die angrenzenden Felder anzusteuern. Ganz unten in der Abbildung sieht man einen kleinen, 3×3 Felder großen Ausschnitt der Sternenkarte. Mit besser ausgestatteten Schiffen im Laufe des Spiels vergrößert sich dieser Kartenausschnitt.

Grundsätzlich wird es nicht jedem Spieler ausnahmslos möglich sein, rechtzeitig zu jedem der vorgegebenen Rundenwechsel alle seine Einheiten navigiert und seine Kolonien versorgt zu haben. Die Energie, die in Abwesenheit des Spielers nicht vom Schiff gespeichert wird, weil etwa die Batterien voll sind, verfällt, was den Spieler im Zeitplan zurückwirft. Kehrt er später zum Spiel zurück, so setzt sich dann die mühsame schrittweise Navigation wie beschrieben fort.

Es existieren Browser-Spiele, die andere Lösungen einsetzen. Bei dem Spiel „OGame“ beispielsweise trägt der Spieler die Zielkoordinaten in ein Eingabefeld ein, das Schiff wird daraufhin vorübergehend unantastbar und taucht einige Zeit später plötzlich am Zielort auf. Hindernisse, Treibgut oder Gegner, die dem Raumschiff auf dem Weg begegnen würden, werden bei diesem Ansatz einfach ignoriert, da der Weg selbst dem Spieler unterschlagen wird. Dadurch verliert das Spiel an Realismus. Es wird also offensichtlich, dass ein Spieler, allein bezogen auf die Navigation der Einheiten bereits, je nach Spiel tendenziell die Wahl zwischen weniger Realismus oder mangelndem Komfort (und somit auch eingeschränktem Spielspaß) hat.



Abbildung 4. Die Navigation der Schiffe im Browser-Spiel „Star Trek Universe“.

3.2 Hoher Zeitaufwand im fortgeschrittenen Spiel

Später, im fortgeschrittenen Spiel, kehrt sich der Effekt allerdings ins Gegenteil. Dann verkürzen sich die langen Wartezeiten zunehmend, bis schließlich die Zeit zwischen den Rundenwechseln immer knapper wird.

Besitzt der Spieler mehrere Hundert Schiffe und ein dutzend Kolonien über verschiedene Planeten und Monde verteilt, so benötigt er die Zeit zwischen den Runden, um seine Kolonien mit den nötigen Nahrungsmitteln, Behausungen, Arbeitsplätzen und Rohstoffen zu versorgen, die Umsetzung seiner Baupläne einzuleiten und die Flotten durch den Sektor zu navigieren. Auch wenn man, wie es bei Star Trek Universe der Fall ist, seine 300 Schiffe in Flotten zu je zehn Schiffen zusammenfassen kann, so hat man trotzdem noch 30 Flotten, die unabhängig voneinander mit denselben Methoden wie einzelne Schiffe mühsam

zu navigieren sind. Dabei bleiben natürlich auch alle angesprochenen Mängel unverändert erhalten. Es kommt ferner hinzu, dass die Navigation einer Flotte, wenn sie mit der eines einzelnen Schiffes gleichgesetzt wird, einerseits weniger realistisch, andererseits eher unflexibel ist. Zugegebenermaßen ist diese Lösung trotzdem besser, als wenn man alle Schiffe getrennt voneinander manuell steuern müsste.

3.3 Die Gefahr der Abwesenheit

Wie in den meisten Strategiespielen sind auch in „Star Trek Universe“ Kampfhandlungen vorgesehen. Gegenüber Verbrauch und Produktion von Ressourcen finden Schlachten jedoch in Echtzeit statt. Schiffe verfügen über eine rudimentäre, automatische Selbstverteidigung – jedenfalls wenn der Spieler rechtzeitig die entsprechende Alarmstufe aktiviert, dann feuert das Schiff bei jedem Treffer durch den Angreifer genau eine Rakete zurück.

Ein häufiges Szenario ist, dass eine Einheit eines offensichtlich abwesenden Spielers angegriffen und vernichtet wird, indem der Angreifende mit mehreren Schiffen gleichzeitig auf dasselbe Ziel feuert. Das angegriffene Objekt wird nun beispielsweise acht mal getroffen, während jede feuernde Einheit des Gegners lediglich einmal und eben auch nur vom Zielobjekt selbst zurückgetroffen wird. Es entsteht also jeweils ein schnell reparabler Schaden für den Angreifer, aber das Opfer verliert eine ganze Einheit.

Wäre der angegriffene Spieler während der Schlacht anwesend gewesen, so hätte er mit den umliegenden Einheiten und in Reichweite liegenden Schiffen die gegnerischen Schiffe angreifen, damit stärker beschädigen und teilweise auch vernichten können, um auf diese Weise eigene Verluste zu vermeiden oder zumindest einzugrenzen.

Physische Abwesenheit eines Spielteilnehmers kann in diesem Genre also den Spielerfolg gefährden oder stark einschränken, wenngleich sie hier für einige Menschen oft unvermeidbar ist. Gegnerische Mitstreiter, die sich ein solches Browser-Spiel zum Lebensinhalt machen und ihre Tage wie Nächte ohne erkennbare Unterbrechung am Computer verbringen, sind dadurch so gravierend im Vorteil, dass ihnen ein zwischenzeitlich abwesender Gelegenheitsspieler praktisch schutzlos ausgeliefert ist.

4 Spieler-programmierbare Agenten

Im folgenden Abschnitt betrachten wir das Konzept programmierbarer Agenten und die Möglichkeiten, die sich dem Spieler dadurch eröffnen, genauer. In Browser-Spielen findet es heute noch keine Anwendung, allerdings könnten sich mit diesem Konzept die angesprochenen Mängel bedeutend reduzieren lassen.

4.1 Taktiken und Aktionen

Wie bei Strategiespielen allgemein üblich, verfolgt der Spieler eine übergreifende Strategie, die sich aus verschiedenen Taktiken zusammensetzt. Diese Taktiken

wiederum bestehen aus einzelnen Schritten, den Aktionen, die der Spieler dann tatsächlich ausführt respektive den einzelnen Spielfiguren jeweils aufträgt. Siehe dazu auch [PMASA06].

Einige Taktiken führt der Spieler aktiv aus eigenem Antrieb aus, um seine Ziele zu verfolgen und umzusetzen, andere dagegen erwidert er reaktiv auf die Aktionen der Mitstreiter oder entsprechend der Spielsituation, um sich zu verteidigen, seine Einheiten zu versorgen oder eine günstige Gelegenheit, wie etwa spezielle Preislagen an den Warenbörsen, auszunutzen.

In Browser-Spielen ist es gemeinhin der Fall, dass sämtliche Aktionen vom Spielenden manuell auszuführen sind. Das kann sich zum einen als sehr mühsam herausstellen, wie oben bereits aufgezeigt, zum anderen verliert man im fortgeschrittenen Spiel aber auch schnell den Überblick. Während der Koordination und Navigation diverser Flotten können dem Spieler Taktiken, die er vorher im Zuge seiner übergeordneten Strategie geplant hatte, in Vergessenheit geraten, oder diese einfach zu kurz kommen, weil die Zeit zu knapp wird.

Programmierbare Agenten können dem Spieler hier immense Hilfestellungen leisten, indem er – sofern angebracht und erwünscht – spezielle Taktiken an diese delegiert, ohne dabei jegliche Kontrolle oder gar die Gesamtstrategie aus der Hand zu geben.

Abbildung 5 stellt schematisch dar, wie ein Spieler den computergesteuerten Agenten, die ihm vom Spiel zur Verfügung gestellt werden, Taktiken zuweisen kann. Dabei ist die Trennlinie zwischen Spieler und Computer als halbdurchlässig zu verstehen: Der Spieler kann die Ausführung der Taktiken und der einzelnen Aktionen vollständig selbst übernehmen, die Strategie – also Form und Auswahl der Taktiken – bleibt ihm in jedem Fall überlassen. An denjenigen Stellen, an denen der Spieler es wünscht, kann er den Agenten Aktionen und Taktiken auftragen, die dann von diesen automatisch abgearbeitet werden. Es soll noch einmal die Wichtigkeit betont werden, dass der Spieler selbst den Punkt bestimmt, an dem der Agent übernimmt – oder eben nicht übernimmt, und dass es sich hier in keiner Weise um eine Einschränkung, sondern um eine Erweiterung der Spieloptionen handelt.

Um eigene Taktiken zusammenzusetzen, bekommt jeder Spieler ein vorgegebenes Repertoire an Aktionen wie etwa „Navigiere das Schiff zu den Zielkoordinaten $(x | y)$ “, „Sende eine Mitteilung an den Spieler“ usw. Die daraus erzeugten Taktiken repräsentieren dann kleinere oder auch größere, modular aufgebaute Aufgaben wie beispielsweise „Finde und kolonisiere den nächstgelegenen freien Planeten der Klasse ‘M’ und benachrichtige den Spieler anschließend!“, die dann von einem Agenten abgearbeitet werden bis entweder die Aufgabe erfüllt ist, oder aber der Spieler die Ausführung unterbricht und selbst ins Geschehen eingreift oder eine andere Taktik auswählt und damit die laufende Taktik abbricht. Die Ausführung größerer Aufgaben kann auch über mehrere Rundenwechsel hinweg andauern.

Die beschriebenen programmierbaren Agenten haben unter anderem den Zweck, dem Teilnehmer das Spiel angenehmer und bequemer zu gestalten und so den Spielspaß zu erhöhen. Gerade für Anfänger stünde das Erlernen einer

entsprechenden Skriptsprache als Voraussetzung zu diesem Ziel im Gegensatz. Aus diesem Grund muss die Schnittstelle des Spiels über einen schnellen, bequemen und intuitiven Weg verfügen, Taktiken zu erstellen. Eine einfache Benutzerschnittstelle, wie sie in Abbildung 6 skizziert ist, ist daher unerlässlich.

Die auszuführenden Aktionen werden mittels einfachem Maus-Klick aus einer Liste ausgewählt und sequentiell untereinander angeordnet. Genauso mühelos kann der Spieler durch einen weiteren Klick Aktionen aus einer Taktik entfernen. Einige Aktionen können über Parameter feinjustiert werden. Als Beispiel dient in der Grafik der Planetentyp, welcher bläulich und unterstrichen hervorgehoben ist.

Die Ausgabe einer Aktion (z.B. gibt „Finde einen Planeten mit Attribut *A*“ einen gefundenen Planeten zurück) dient – falls vom passenden Typen – als Eingabe des nachfolgenden Befehls. Denkbar sind auch Verfeinerungen wie bedingte Verzweigungen, Unteraufrufe von weiteren Taktiken und Sprünge oder Schleifen, was aber im Detail den Rahmen dieser Arbeit sprengen würde ohne dabei besonders zur Veranschaulichung des Konzepts beizutragen.

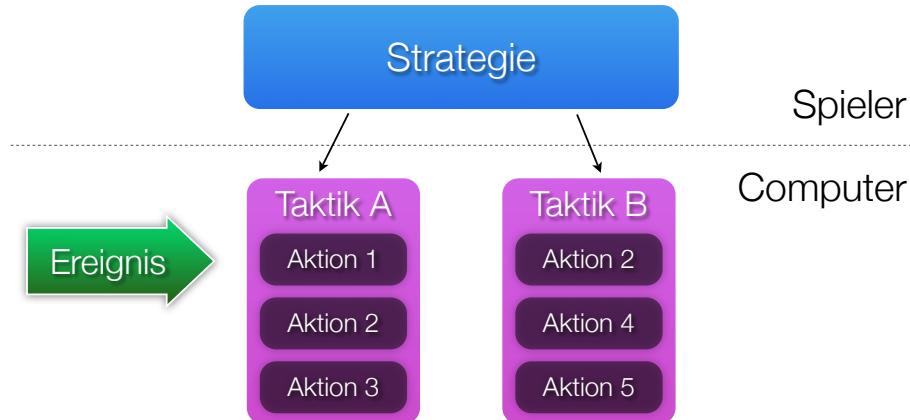


Abbildung 5. Strategien, Taktiken, Aktionen, Ereignisse und ihr Zusammenhang.

4.2 Ereignisse

Wie wir gesehen haben, kann jeder Spieler seinen Agenten Taktiken zuweisen, die von diesen dann unverzüglich umgesetzt werden. In der Spiel-Realität kommt es aber häufiger vor, dass ein Spieler die eigenen Handlungen nicht aus eigenem Antrieb anstößt, sondern mit diesen auf eine Umweltsituation reagiert.

Entsprechend sollte es daher auch möglich sein, dass Agenten nicht nur über Einheiten mit Taktiken direkt verknüpft werden können, sondern zusätzlich auch über Ereignisse, die entweder eintreten oder auch nicht eintreten. Ein Agent

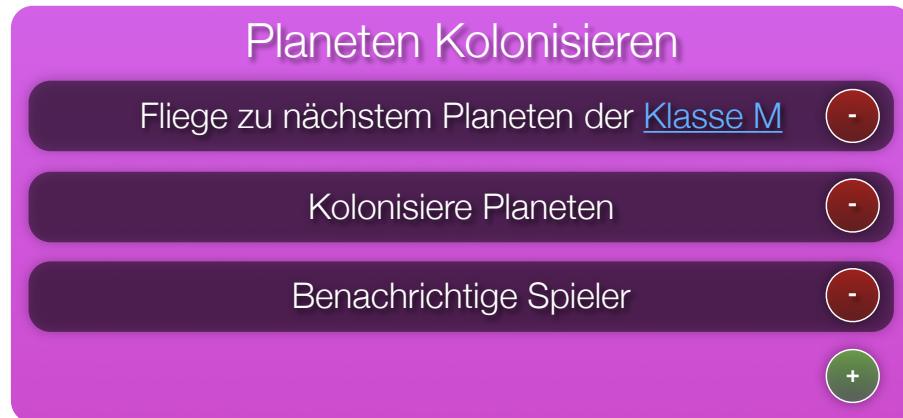


Abbildung 6. Einfacher Prototyp für ein Bedienelement, das dem Spieler auf komfortable Weise ermöglicht Taktiken aus einem Repertoire von vorgegebenen Aktionen zusammenzustellen.

würde dann beim Eintreten eines zugewiesenen Ereignisses, das die entsprechende Einheit betrifft, die Taktik, die der Spieler vorher für den speziellen Fall ausgewählt hat, initiieren.

Mögliche Beispiele für solche Ereignisse sind „Einheit wird angegriffen“ oder „Zahl der Arbeitslosen in der Kolonie steigt“. Der Spieler würde den Agenten dann automatisch passende Taktiken einleiten lassen wie „Erhöhe Schild-Energie“ oder „Baue Fabrik“.

Für differenziertere Taktiken sind auch hier einfache Bedingungen hilfreich, wie „Wenn Angreifer klingonisch, dann Flucht einleiten“ und „Wenn Nahrungs-mittelvorrat < 5, dann baue Bauernhof“.

4.3 Intelligenter navigieren

Wie zuvor beschrieben nimmt die Navigation der Schiffe in dem Spiel „Star Trek Universe“, wie auch in den meisten vergleichbaren Spielen, einen bedeutenden Teil der Spielhandlungen ein. Die Lösung, für die sich die Entwickler hier entschieden haben, ist einerseits realistischer, auf der anderen Seite aber auch mühsamer für den Spielteilnehmer zu handhaben als die Umsetzungen in einigen anderen Spielen.

Mithilfe der beschriebenen Spieler-programmierbaren Agenten und dem dazugehörigen Automationssystem lässt sich schnell und einfach eine Taktik entwickeln, die dafür sorgt, dass ein Schiff sein Ziel erreicht, ohne dass der Spieler in jeder Runde die beschriebenen repetitiven Aktionen ausführen und dann wieder auf den nächsten Rundenwechsel warten muss. Er beauftragt einfach einen Agenten, das Schiff an das gewünschte Ziel zu steuern und ihn schließlich bei Ankunft durch den spielinternen Nachrichtendienst zu benachrichtigen.

Dank dem Konzept der Ereignisse kann der Spieler ganz einfach ausgeklügelte Verfahren modellieren, die alle denkbaren Zwischenfälle berücksichtigen. Wird nun durch die Sensoren des Schiffes nach dem Einfliegen in ein neues Feld auf der Sternenkarte ein Ereignis „angrenzendes Feld ist ein Hindernis“ gefeuert, so kann der Agent das Feld automatisch umgehen, wenn diesem Ereignis vorher durch den Spieler eine entsprechende Taktik zugewiesen worden ist. Der Agent könnte bei Kollision mit Treibgut dieses beispielsweise einsammeln, falls es eine bestimmte Eigenschaft erfüllt. Auf eine ähnliche Weise könnte man den Agenten auf gegnerische Schiffe oder Angreifer reagieren lassen. Ein Ereignis „Einheit wird Angegriffen“ mit gleichzeitiger Bedingung „angreifende Flotte ist in der Übermacht“ würde eine vorgegebene Flucht-Taktik auswählen und aktivieren.

einige typischen Methoden der CI/KI einsetzen. Beispielsweise ist zum Zweck der Wegfindung durch das Universum der A^* -Algorithmus [BS04] bestens geeignet. Der Agent kann automatisch einen optimalen Weg finden (sofern ein solcher existiert), der dann in jeder Runde entsprechend den aktuellen Gegebenheiten angepasst wird. Dabei können Hindernisse wie Nebel und Asteroidenfelder, sowie Feindgebiet berücksichtigt und nach Möglichkeit gemieden werden, in dem man solchen Feldern Extrakosten zuordnet.

Gruppen von Einheiten respektive Flotten können über eine *gemeinsame* Taktik navigiert werden. Wenn man zusätzlich Schwarmverhalten simuliert, siehe dazu „Chapter 4. Flocking“ [BS04], wirkt die Bewegung der Einheiten natürlich und realistischer.

Betrachten wir das hier vorgestellte Automationssystem aus einer theoretischen Perspektive, so wird klar, dass es sich um ein einfaches, vom Spieler modifizierbares, regelbasiertes System handelt, das teilweise ereignisgesteuert funktioniert. Die Implementierung gestaltet sich entsprechend verhältnismäßig einfach.

Zur intelligenteren Einschätzung nicht ganz eindeutiger Gegebenheiten und zur besseren Entscheidungsfindung bietet es sich ferner an, den Agenten – neben den reinen Sensordaten der Einheiten – durch Fuzzy Logic mächtige Hilfsmittel an die Hand zu geben. Damit wäre die automatische Beurteilung von kritischen Situationen möglich, wie etwa „Ist das angreifende Schiff im Kampf überlegen?“, „Lohnt es sich, die geladenen Güter abzuwerfen und stattdessen das gefundene Treibgut aufzusammeln?“ oder „Erfordert die Situation in der Kolonie eher den Bau von Massenbehausungen oder die Errichtung von Bauernhöfen?“. Grundsätzliche Erläuterungen zur Fuzzy Logic und detailliertere Hinweise zur Implementierung sind ebenfalls in [BS04], Kapitel 10 nachzulesen.

4.4 Zeitkosten und Aufwand reduzieren

Anfangs wurde bemerkt, dass mit fortgeschrittenem Spiel die Zahl der zu steuernden Einheiten drastisch zunimmt und die verfügbare Zeit zwischen den Rundenwechseln immer knapper wird.

Setzt man in dieser Situation Spieler-programmierbare Agenten ein, so lassen sich statt vielen kleinen repetitiven Schritten eher übergeordnete Ziele formulie-

ren. Der Teilnehmer kann das Spiel von einer höheren Ebene aus betrachten und sich auf das wesentliche konzentrieren. Abhängig von der Qualität und Ausgefeiltheit seiner Taktiken kann sich der Spieler nach dem Prinzip „divide et impera“ um seine Strategie kümmern ohne andauernd von Kleinigkeiten abgelenkt zu werden. Er kann sich sogar in manchen Situationen, in denen er ohne seine Agenten kurz vor der Verzweiflung gestanden hätte, einfach zurücklehnen und die nächsten strategischen Schritte planen. Dank ereignisgesteuerter „getriggerter“ Taktiken muss er in gängigen, aber kritischen Situationen nicht einmal intervenieren, da er seine Agenten rechtzeitig auf Angriffe und andere unplanmäßige Ereignisse vorbereiten kann.

Aber selbst wenn der Spielteilnehmer direkt in das kurzfristige, konkrete Spielgeschehen involviert ist, weil er beispielsweise durch gefährliches, dichtbesiedeltes Feindgebiet fliegt und fest damit rechnen muss, dass es zu Kampfhandlungen kommen wird, können Agenten von bedeutendem Vorteil sein, hat er zuvor möglicherweise eine Handvoll erprobter Kampfmanöver einprogrammiert, auf die er dann spontan in der Kampfszene zurückgreifen kann.

Selbstverständlich sollte nicht nur ein einziger, besonders bevorzugter Spieler, sondern jeder der Teilnehmer über Agenten verfügen, sonst wäre ein solches Spiel ungerecht und demotivierend. Eine Möglichkeit wäre hier auch, zu Beginn des Spiels eine kleine Zahl von Agenten vorzugeben und Einrichtungen zu schaffen, dass sich Spieler im Verlauf des Spiels weitere hinzuerdienen oder bei verlorenen Schlachten Agenten verlieren können.

Vergegenwärtigt man sich nun, welche Perspektiven sich dem einzelnen Spieler eröffnen, seine eigenen Agenten gegen gegnerische (künstliche oder natürliche) Intelligenzen antreten zu lassen, blickt man in eine neuartige⁷ Welt der Möglichkeiten bei Computerspielen allgemein.

4.5 Sicherheit und Versorgung gewährleisten

Ein weiterer wichtiger Punkt ist das Wohlbefinden der Kolonisten, um das der Siedler regelmäßig bemüht sein sollte. Nur eine stets ausreichende Zahl an Wohnseinheiten, Arbeitsplätzen, Energieeinheiten und Nahrungsmitteln erlaubt wirtschaftliches Wachstum, mindestens ein Teilziel in Strategiespielen. Auch Lagerraum für Überproduktionen und Kapazitäten zur Speicherung überschüssiger Energie sind wichtig, um unnötige Verluste zu vermeiden.

Normalerweise ist der Spielteilnehmer gefordert, regelmäßig selbst zu überwachen, dass keine Notstände entstehen. Mithilfe programmierbarer Agenten könnte er aber einfache kybernetische Systeme schaffen, indem er Taktiken bedingt an Ereignisse bindet, die automatisch bei Änderungen der Lagerbestände ausgelöst werden.

Wird beispielsweise die Zahl der Nahrungsmittel allmählich knapp, so kann der Agent eventuell stillgelegte Bauernhöfe reaktivieren oder, falls das nicht

⁷ Nicht ganz neuartig, aber dennoch sehr selten: In dem Spiel „Core War“ beispielsweise, welches in den 80er Jahren kurzzeitig Berühmtheit erlangte, programmierten die Spieler Assembler-Programme, die gegeneinander antraten [Wik07b].

möglich ist, weil in der Kolonie keine stillgelegten Bauernhöfe existieren, neue Bauernhöfe oder andere Arten von Nahrungsmittelproduktionen bauen, je nach verfügbarer Energie, Baumaterial und anderen Rohstoffen.

Weiter oben wurde auf die Gefahren eingegangen, die einem drohen können, wenn man als Spieler nicht permanent anwesend und aufmerksam ist. Insbesondere gegnerische Mitstreiter können einem mit zunehmendem Einfluss gefährlich werden.

Viele Browser-Spiele, so etwa das vorgestellte „Star Trek Universe“, bieten aus diesem Grund einen Urlaubsmodus an, in dem alle Ressourcen eingefroren werden und der Spieler über einen begrenzten Zeitraum hinweg unverwundbar ist. Allerdings besteht bei aktivem Urlaubsmodus natürlich auch keine Möglichkeit, in irgendeiner Form das Spielgeschehen zu beeinflussen. Cleverer und realistischer als das Einfrieren und die Isolation wäre es hier, Agenten einzusetzen, die bei Abwesenheit des Spielers übernehmen und wie beschrieben einerseits einfach Aufträge ausführen, andererseits als Lebenserhaltungs- und Verteidigungssysteme dienen.

Gerade in Bezug auf Verteidigungsmaßnahmen ist hier die Integration von weitergehenden Methoden der CI denkbar und interessant – trainierbare oder gar selbstständig lernende Agenten seien als Beispiel genannt. Insbesondere Verfahren, die bereits eingesetzt werden, etwa in Echtzeitstrategiespielen, bieten sich hier an: „Dynamic Scripting“, evolutionäre Algorithmen, Wissensbasen, künstliche Neuronale Netze und weitere. Interessante Informationen hierzu sind unter anderem in [PMASA06] und [BS04] zu finden.

5 Zusammenfassung und Schlussbemerkung

Browser-Spiele sind trotz der vor einigen Jahren noch unvorstellbaren Möglichkeiten, die das Internet heute bietet, eine sehr statische und trockene Angelegenheit, sieht man von der spielbegleitenden Kommunikation zwischen den Spielern ab. Aufwändige Technologien, die moderne Spiele oft besonders beliebt und erfolgreich machen, sucht man hier ebenso vergeblich wie den Einsatz von künstlicher Intelligenz. Wenn sich einem auch die Notwendigkeit für die Anwendung von Methoden der Computational Intelligence nicht sofort und ohne weitere Überlegungen offenbaren mag, so könnte man mit solchen Mitteln derartige Spiele doch gravierend verbessern – nicht indem man zusätzliche, künstliche Gegner schafft, sondern vielmehr durch das Bereitstellen von Spieler-programmierbaren Agenten, die diesem unangenehme Arbeit abnehmen oder ihn bei Abwesenheit vertreten können. Schließlich ließen sich auf diesem Weg neue Perspektiven und Spielweisen schaffen und dem Spieler mehr Möglichkeiten und mehr Macht an die Hand geben, die für gewöhnlich in dieser Art den Programmierern alleine vorbehalten ist. Dies könnte vor allem den Spielspaß immens fördern.

Literatur

- [BS04] BOURG, David M. ; SEEMANN, Glenn: *AI for Game Developers*. O'Reilly Media, Inc., 2004. – ISBN 0596005555
- [Eri07] ERIKSSON, Joakim: *JaC64 - Java Commodore C64 emulation, games and demos*. <http://www.jac64.com/>, Stand: 09. Apr. 2007
- [Jak07] JAKOB, Daniel: *Star Trek Universe*. <http://www.stuniverse.de>, Stand: 27. Sep. 2007
- [Kra07a] KRAUSS, Hans J.: *Andromeda*. <http://www.andromedateam.de>, Stand: 27. Sep. 2007
- [Kra07b] KRAUSS, Hans J.: *Andromeda - Offizielles Forum*. <http://www.ogfnetwork.de/forumdisplay.php?f=1098>, Stand: 27. Sep. 2007
- [Mak07] MAKIMOTO, Kunio: „*Super Maryo World*“ - Javascript Virtual Machine. <http://www.janis.or.jp/users/segabito/JavaScriptMaryo.html>, Stand: 27. Sep. 2007
- [PMASA06] PONSEN, M.J.V. ; MUÑOZ-AVILA, H. ; SPRONCK, P. ; AHA, D.W.: Automatically Generating Game Tactics via Evolutionary Learning. In: *AI Magazine* 27 (2006), S. 75–84
- [SW07] SCHALL, Darren ; WAHLERS, Claus: *FC64 Open Source Flash*. <http://osflash.org/fc64/>, Stand: 26. Feb. 2007
- [Uni07] *Unity*. <http://unity3d.com>, Stand: 27. Sep. 2007
- [Wei66] WEIZENBAUM, J.: Eliza - a computer program for the study of natural language communication between man and machine. In: *Comm.A.C.M.* 9 (1966), Januar, Nr. 1, S. 36–45
- [Wik07a] WIKIPEDIA (Hrsg.): *Browserspiel*. <http://de.wikipedia.org/wiki/Browserspiel>, Stand: 22. Sep. 2007
- [Wik07b] WIKIPEDIA (Hrsg.): *Core War*. http://de.wikipedia.org/wiki/Core_War, Stand: 5. Sep. 2007
- [Wik07c] WIKIPEDIA (Hrsg.): *ELIZA*. <http://de.wikipedia.org/wiki/ELIZA>, Stand: 6. Sep. 2007

CI beim Roboterfußball

CI & KI bei Computerspielen

Patrick Szcypior
Matrikelnummer 96198

TU Dortmund

1 Was ist der RoboCup?

Der RoboCup ist eine internationale Initiative, gegründet im Jahr 1997, um die Forschung und Bildung der künstlichen Intelligenz und die Entwicklung von Robotern voranzutreiben. Es ist ein bestimmtes Problem zu lösen, welches ein breites Spektrum von Technologien benötigt. Um einen gewissen sportlichen Ehrgeiz bei den Forschern wie auch die Aufmerksamkeit der Öffentlichkeit zu wecken, entschied man sich für die bekannte Sportart Fußball. Das Ziel dieses Wettbewerbes ist es, im Jahr 2050 mit einer Mannschaft bestehend aus autonomen humanoiden Robotern den amtierenden Fußballweltmeister unter offiziellen FIFA Bedingungen zu besiegen. Weitere Disziplinen wie die *RescueLeague*, *RoboCup@Home* und *RoboCup Junior* vervollständigen diesen Wettbewerb.

1.1 Roboterfußball

Die Tuniere werden in verschiedenen Ligen gespielt, wobei sich jeweils die Anforderungen an die Robotern und deren Komplexität unterscheiden, siehe <http://www.robocup.org>.

– Simulation League

In der Simulationsliga spielen unabhängig voneinander handelnde Softwareagenten auf einem simulierten Fußballfeld gegeneinander. Hier sind die ausgereiftesten Strategien zu beobachten, da dies die älteste Liga ist und die Probleme einer realen Umwelt nicht auftreten. Es wird jeweils, wie beim 'echten' Fußball, mit elf Spielern gespielt.

– Small-size League

Nur 18cm im Durchmesser messen die kleinen Roboter, die auf einem Feld, groß wie eine Tischtennisplatte, mit jeweils fünf Spielern und einem Golfball gegeneinander antreten. Sie werden über einen zentralen Rechner gesteuert und die Wahrnehmung des Spiels erfolgt über mehrere Deckenkameras.

– Middle-size League

In dieser Liga sind alle Sensoren und Recheneinheiten im Roboter untergebracht, der im Durchmesser höchstens 50cm messen darf. Gespielt wird auf



Abbildung 1. Aldebaran Robotics - Nao, siehe <http://www.aldebaran-robotics.com>

einem maximalen 16x12m großen Spielfeld mit einem orangenen Fußball und vier bis sieben Feldspielern pro Team.

– **Four-legged League**

Bei den bereits vorgestellten Ligen bewegen sich die Roboter mit einem omnidirektionalen Radantrieb sehr zielstrebig über das Spielfeld. In der *Four-legged League* kommen die Roboterhunde AIBOs von Sony zum Einsatz, bei denen die Bewegungsabläufe neue Herausforderungen an die Entwickler stellten. Die Roboter besitzen eine Kamera und einen Infrarot Distanz Sensor im Kopf mit dessen Hilfe die Hunde autonom handeln sollen. Gespielt wird hier vier gegen vier auf einem 3x5m großen Feld.

Durch die Einstellung des Vertriebes der AIBOs wird diese Liga ab 2008 durch die *Standard Platform League* ersetzt. Für diese neue Liga wird ein einheitlicher humanoider Roboter der Firma Aldebaran Robotics bereitgestellt, siehe Abbildung 1.

– **Humanoid League**

Seit 2002 gibt es auch die anspruchsvolle *Humanoid League*. Die Roboter müssen menschenähnliche Proportionen aufweisen und völlig autonom handeln. Anstelle von Taktiken und ausgefeilter Strategien im Spiel steht hier das Laufen auf zwei Beinen im Vordergrund. In einem Spiel werden nur zwei Feldspieler pro Team eingesetzt, da die Komplexität und die Kosten für einen solchen Roboter überaus hoch sind.

1.2 Rescue League

Diese Liga des RoboCups dient der Erforschung und Entwicklung von mobilen Robotern, die teils selbstständig den Einsatzkräften bei einer Katastrophe helfen. Hier soll der Schwerpunkt in Einsatzgebieten liegen, die für einen Menschen unzugänglich oder zu gefährlich sind. Während eines Einsatzes sammelt der Roboter Daten und bereitet diese für den Einsatzstab auf, um basierend darauf weitere Entscheidungen fällen zu können. Der Wettbewerb findet zum einen in einer Software Simulation statt und zum anderen auch mit realen Robotern in einem simulierten Katastropheneinsatz.

2 Autonome mobile Roboter

Roboter, die sich in ihrer Umgebung selbstständig bewegen und handeln können, bezeichnet man als autonome mobile Roboter. Meist befindet sich die steuernde Elektronik, Hard- und Software, am Roboter selbst. Um in der Umwelt agieren zu können, muss er diese zuerst perzipieren. Dies geschieht über eine Reihe von Sensoren. Zum einen existieren die *exteroceptive* Sensoren, welche die Umwelt wahrnehmen und zum anderen die *proprioceptive* Sensoren, die Änderungen vom Roboter selbst bemerkten. Das Sehvermögen über Kameras ist im RoboCup die wichtigste Wahrnehmung, um ein geeignetes Modell der Umwelt zu erstellen. Es gibt bei der Nutzung von Sensoren lediglich in der *Humanoid League* eine Einschränkung, denn dort sind nur passive Sensoren erlaubt, um sich gegenüber dem Menschen keine Vorteile zu verschaffen. Im Gegensatz dazu werden in der *Rescue League* aktive Abstandssensoren benutzt, um genauere Daten der Umwelt liefern zu können. Sensoren, die Zustandsänderungen des Roboters erkennen, sind unter anderem Beschleunigungssensoren und Gyroskope [1].

Aus den gewonnenen Perzepten der Sensorik erstellt der Roboter ein Modell der Umwelt, welches als Basis für sein Verhalten dient. Die Entscheidungen, die ein Roboter selbstständig trifft, resultieren in Aktionen, welche von der Steuereinheit des Roboters ausgeführt werden. Wesentlich ist die Fortbewegung, die beim RoboCup Fußball meist über einen omnidirektionalen Radantrieb oder durch das Laufen auf zwei Beinen realisiert wird. In der *Rescue League* werden in der Regel Rad- und Kettenantriebe benutzt. Zum Schießen sind in der *Small-size*- und *Middle-size League* besondere Vorrichtungen angebracht. In den anderen Ligen wird die vorhandene Anatomie des Roboters zum Schießen bestmöglichst ausgenutzt.

Daraus ergeben sich drei Prozesse beim autonomen mobilen Roboter, welche seriell verarbeitet werden (siehe Abbildung 2). Er nimmt über Sensoren die Umwelt wahr, entscheidet daraufhin über sein Verhalten und führt schließlich Aktionen aus [3]. In dieser Arbeit werden Ansätze und Möglichkeiten erläutert, Verfahren aus der Computational Intelligence und Künstlichen Intelligenz beim RoboCup in der Entscheidungsfindung einzusetzen.

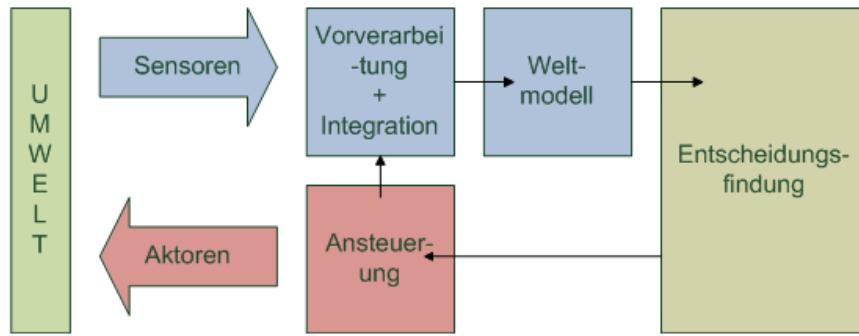


Abbildung 2. Interaktion mit der Umwelt

2.1 Simulation

Die Simulation ist eine Vorgehensweise bei der autonome mobile Roboter in einer virtuellen Umgebung analysiert werden. Dabei werden die physikalischen Eigenschaften des Roboters und der Umwelt modelliert, um so das dynamische System berechnen zu können. Dies ist für die Entwicklung von Bedeutung, da Design- und Konzeptionsfehler evaluiert werden können, bevor ein Prototyp des Roboters gebaut werden kann. Des Weiteren wird durch die Simulation die Entstehung des Roboters beschleunigt und kostengünstiger.

Die Komplexität der Simulation beschränkt sich auf das zu Grunde liegenden physikalische Modell. Zu beachten ist, dass Faktoren wie zum Beispiel das Verrauschen der Sensoren, die sich stets veränderbaren Lichtverhältnisse oder Unebenheiten in Böden, nicht realitätstreu simuliert werden und zu unterschiedlichen Ergebnissen zwischen der realen Welt und dem Simulator führen können.

3 Entscheidungsfindung

Der Roboter muss auf Grund seiner Wahrnehmung Entscheidungen treffen, die auf ein zu erfüllendes Ziel hinauslaufen. Im Roboter Fußball ist das Ziel eindeutig definiert. Das Spiel ist Gewonnen, wenn am Spielende mehr Tore erzielt wurden als sie zu zulassen. Da dies ein sehr komplexes Problem darstellt, teilt man die Aufgabe hierarchisch in kleine Teilaufgaben auf. So besitzt ein Agent eine Reihe von Verhalten, die hierarchisch miteinander verknüpft sind (siehe Abbildung 3). Die in einer Hierarchie höher liegenden Aufgaben werden Optionen oder Rollen genannt. In der untersten Schicht der Verhaltensstruktur befinden sich die Einzelfähigkeiten, welche nur aus einer Sequenz von elementaren Aktionen bestehen. Zu den elementaren Aktionen gehören zum Beispiel $Kick(x)$, $Turn(\alpha)$, Schießen mit der Stärke x oder Drehen um den Winkel α . Oft werden diese Einzelfähigkeiten ausprogrammiert, doch es gibt Ansätze diese Fähigkeiten über das Reinforcement Learning (RL) zu erlernen. Diese werden im Folgenden erläutert.

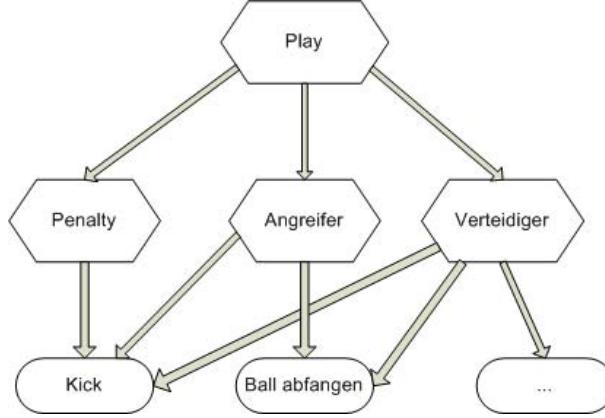


Abbildung 3. Hierarchische Darstellung der Verhalten

3.1 Reinforcement Learning

Reinforcement Learning beruht auf dem Prinzip ausschließlich durch die Interaktion mit der Umwelt zu lernen. Für jede Aktion, die der Spieler ausführt, wird er entweder belohnt, da dies zu einem gewünschten Ziel geführt hat, oder er wird bestraft. Das Ziel eines RL Algorithmus ist es, eine Strategie zu finden, die in allen Situationen stets die Aktion wählt, die zu minimalen Kosten bzw. Bestrafungen führt oder zu einer maximalen Belohnung.

Zum Modellieren dieses Problems, eignet sich der Markovsche Entscheidungsprozess (MDP). Eine Menge S beschreibt die Zustände und eine Menge A die Aktionen. Zu einem Zeitpunkt t wählt der Agent basierend auf dem aktuellen Zustand $s_t \in S$ eine Aktion $a_t \in A$ aus. Dies führt mittels der Übergangsfunktion $\delta : S \times A \rightarrow S$ zu einem Folgezustand. Die Kostenfunktion $c : S \times A \rightarrow \mathbb{R}$ liefert die Kosten, die bei einer Aktion A in einem Zustand S entstehen. Die Auswahl der nächsten Aktion trifft der Agent über die Verhaltensregel $\pi : S \rightarrow A$. Eine gute Verhaltensregel ist das Lernziel des Agenten. Das erlernte Verhalten ist optimal, wenn die erwarteten kumulierten Kosten

$$V^\pi(s) = E \sum_{t=0}^{\infty} c(s_t, \pi(s_t)), s_0 = s \quad (1)$$

für jeden Zustand minimal sind, welches durch die Wertfunktion dargestellt wird (Reinforcement Learning [2, 3]).

Bei einfachen Problemen mit nur wenigen diskreten Zuständen kann diese Funktion mittels einer Tabelle dargestellt werden. Bei realen Problemen hingegen ist der Zustandsraum kontinuierlich. Anstatt für jeden Zustand eine separate Bewertung zu erlernen und zu speichern, wird die Wertfunktion (Gleichung 1) durch ein mehrschichtiges neuronales Netz approximiert.

In den folgenden Anwendungen wird ein solches Netz verwendet, welches als Eingabeneuronen den Zustandsraum S und die Aktionsmenge A hat. Es gibt

nur ein Ausgabeneuron, das den Funktionswert der Wertfunktion (Gleichung 1) approximiert. Der Zustandsraum S wird durch das beschriebene Weltmodell repräsentiert und die Aktionen A meist durch die elementaren Aktionen eines Roboters.

3.2 Erlernen von Einzelfähigkeiten

Im RoboCup bietet die Simulationsliga gute Voraussetzungen für den Einsatz von Reinforcement Learning. Dort spielen keine realen Roboter gegeneinander, sondern ein komplettes Fußballspiel wird von einer Software, dem *SoccerServer*, simuliert. Dadurch sind die Regeln der Simulation bekannt und es können präzise Vorhersagen über das Geschehen getroffen werden.

Die Spieler werden von Softwareagenten gesteuert, die jeweils in eigenständigen Prozessen laufen und über eine Server-Client Architektur mit dem *SoccerServer* verbunden sind. Die Agenten dürfen nur über den Umweg mit dem Server untereinander kommunizieren. In diskreten 100 ms Schritten übermittelt der Server Daten an die Clienten, die die Positionen von Spielern und dem Ball im Sichtradius des Spielers beinhalten. Da diese Daten nur mit einem leichten Rauschen verfälscht sind, kann jeder Agent ein ziemlich genaues Weltmodell erstellen.

Das genaue Weltmodell und Bekanntheit der Simulationsregeln unterstützt das Reinforcement Learning. So ist die Übergangsfunktion δ stets bekannt und kann den Nachfolgezustand bei gewählter Aktion berechnen. Dies ermöglicht ein schnelles Lernen und ist nicht abhängig von einer realen Ausführungszeit. Bei realen Robotern wird diese Methode unter Realzeit-Bedingungen angewendet, da der Folgezustand nicht berechnet werden kann, sondern durch die Sensorik wahrgenommen wird.

Einzelfähigkeiten bestehen aus elementaren Aktionen. Zu den wichtigsten Aktionen, die ein Spieler in der Simulationsliga ausführen kann, gehören $dash(x)$, $turn(\alpha)$ und $kick(x)$. Ersteres ist die Beschleunigung mit einer Kraft x in Ausrichtung des Spielers. *Turn* führt eine Drehung um den Winkel α aus und letzteres führt zu einem Schuss mit der Stärke x . Da die Einzelfähigkeiten ein klar definiertes Ziel verfolgen, können sie über die Methode des RLs erlernt werden.

Um Einzelfähigkeiten zu trainieren, lässt man einen Agenten in einem zufälligen Startzustand beginnen. Der Spieler kann entweder anhand seiner aktuellen von einem neuronalen Netz dargestellten Wertfunktion entscheiden, oder er wählt zufällig eine Aktion aus, um zu explorieren. Bei der Exploration lernt der Agent durch Ausprobieren und der Reaktion der Umwelt. Erreicht der Spieler mit seiner letzten Aktion sein Ziel, so bekommt er eine Belohnung, die hier durch Nullkosten beschrieben werden. Bei Nichterreichen des Zielzustandes erhält er geringe Kosten, damit kürzere Ausführzeiten erlernt werden. Ist für eine Einzelfähigkeit ein negativer Zielzustand definiert, zum Beispiel beim Dribbeln der Verlust des Balles, und dieser wird erreicht, so wird der Agent durch hohe Kosten bestraft. Nach jedem Übergang in den Folgezustand, nimmt der Agent eine Aktualisierung der Wertfunktion vor.

Im Beispiel des Ballabfangens muss ein 6-dimensionaler Zustandsraum betrachtet werden, $S = s = (v_p x, v_p y, v_b x, v_b y, d_{bp}, \alpha_{bp})$, der die Ballgeschwindigkeit \vec{v}_b , die Geschwindigkeit des Spielers \vec{v}_p , den Abstand zwischen Ball und Spieler d_{bp} und den Winkel zwischen aktueller Ausrichtung und dem Ball α_{bp} enthält. Die elementaren Aktionen, die der Agent wählen kann, sind 76 verschiedenen parametrisierte *dash* und *turn* Befehle. Es wird nur eine begrenzte Menge an Aktionen zugelassen, da so der Zustandsraum möglichst gering gehalten werden kann.

Es ergaben sich Probleme beim Erlernen des Ballabfangens, die durch die zeitdiskrete Natur, der durch den *SoccerServer* bereitgestellten Umgebung zu erklären sind. Infolgedessen weist die Wertfunktion (1) unendlich viele Unstetigkeitsstellen auf, die dazu führen, dass beispielsweise eine kleine Änderung in einer *turn* Aktion zu einer größeren Anzahl von weiteren Schritten resultiert. Dies konnte in Situationen beobachtet werden, in denen sich der Ball mit hoher Geschwindigkeit nicht frontal auf den Agenten zu bewegte. Dennoch konnte die erlernte Einzelfähigkeit auf einer Testmenge gegenüber der konventionellen Methode, der handgeschriebenen Routine, überzeugen. Sie erreichte eine durchschnittliche Dauer zum Auffangen des Balles von 10.23 Simulationsschritten, die sich verglichen mit dem theoretischen Optimum in einer rauschfreien Umgebung nur um weniger als einen halben Simulationsschritt unterschied.

Weitere Einzelfähigkeiten wurden erfolgreich erlernt und fanden Anwendung im Team Brainstormers der Universität Osnabrück, die diesen Ansatz veröffentlicht haben [3].

Auf reale Roboter ist diese Methode nicht einfach übertragbar, da die Trainingsdurchläufe auf einem solchen Roboter zu viel Zeit in Anspruch nehmen würden. Aber es kann dort das Training im für den Roboter eigenen Simulator ausgeführt werden. Da der Simulator nie das dynamische System der realen Umwelt genau genug berechnen kann, müssen die erlernten Fähigkeiten am Roboter durch weitere Trainingssequenzen evaluiert werden, wobei die Anzahl der Durchläufe dabei deutlich verringert werden kann. Auch durch Anpassungen der Lernparameter des neuronalen Netzes kann die Anzahl der Trainingsdurchläufe verringert werden.

3.3 Erlernen von Teamverhalten

Im vorigen Abschnitt wurde das Verhalten nur von einem einzelnen Roboter betrachtet. Beim Fußballspielen ist aber auch das Zusammenspiel zwischen mehreren Agenten ein wichtiger Erfolgsfaktor. Es wird versucht das Prinzip des RL bei Einzelfähigkeiten auf das Erlernen von Teamfähigkeiten zu übertragen.

Bei der Übertragung der vorgestellten Prinzipien auf das Erlernen von Teamverhalten kommt es zu folgenden Problemen. Die Aktionsmenge der Agenten wächst exponentiell mit der Anzahl der Spieler und der wesentlichsste Nachteil ist ein einheitliches Weltmodell. Da jeder Agent nur seinen Ausschnitt der Umwelt kennt und auf diese Daten das Weltmodell generiert, kann es bei jedem Agenten zu einer unterschiedlichen Weltauffassung kommen. Die ständige Synchronisierung eines solches Weltmodells würde eine hohe Belastung des Kom-

munikationskanal zur Folge haben. Da eine feste Bandbreite nicht garantiert werden kann, wird ein einheitliches Weltmodell nicht angenommen.

Um die Aktionsmenge zu verkleinern, werden keine elementaren Aktionen zugelassen, sondern nur (erlernte) Einzelfähigkeiten: Positionierung in einer von 8 Richtungen, zum Ball gehen, Dribbeln, Torschuss, Pass zu einem Mitspieler, Ballhalten und Ballabfangen. Es werden weiterhin nicht alle Spieler und Gegenspieler in die Zustandsmenge einer Teamfähigkeit aufgenommen. Für ein Angriffsspiel werden zum Beispiel sieben eigene Agenten und acht Gegenspieler ins Weltmodell mit einbezogen. Die Kostenfunktion ist für jeden Agenten gleich:

- **Belohnung (Nullkosten)**

Alle Spieler werden bei einem erzielten Tor belohnt.

- **Bestrafung (hohe Kosten)**

Alle Spieler werden bei dem Verlust des Balles mit hohen Kosten bestraft.

- **Geringe Kosten**

Nach jedem Zustandswchsel, um den schnellen Torabschluss zu belohnen, gibt es für jeden Spieler eine geringe Bestrafung.

Durch diese Modellierung werden die Agenten zur Kooperation gezwungen, schnellstmöglich ein Tor zu erzielen, weil nur so die gemeinsame Kostenfunktion minimiert werden kann.

Das Beispiel des Angriffsspiel mit sieben gegen acht Spielern zeigt, dass der Zustandsraum aus 34 kontinuierlichen, reellwertigen Werten besteht, da eine Diskretisierung der Werte, zu Fehlentscheidungen führen kann, wenn der Erfolg der Entscheidungen von wenigen Zentimetern abhängt.

Für die Auswahl der zu spielenden Aktionen eines Agenten wird zuerst eine Vorauswahl bestimmt. Indem jede Aktion auf ihren Erfolg geprüft wird, werden widersinnige Aktionen ausgeschlossen. Die Folgezustände werden von jeder erfolgreichen Aktion über ein Modell berechnet. Diese werden über die gelernte Wertfunktion bewertet. Es wird die Aktion ausgewählt, die die geringsten Kosten vorhersagt und damit auch am schnellsten zum Torerfolg führt. Das eingesetzte Modell zur Vorhersage des Folgezustands kann nur approximativ sein, da der Agent keine Aussagen über das Verhalten der Gegen- und Mitspieler treffen kann. Es könnte hier zum Beispiel eine worst-case Analyse betrachtet werden um die Auswahl des Folgezustands genauer zu berechnen. Dies würde aber einen hohen Rechenaufwand für jeden Agenten bedeuten. Somit nimmt diese Methode an, dass sich alle auf dem Spielfeld befindenden Spieler nicht handeln und wählt die für sich bestmögliche Aktion aus.

Das Team Brainstormers hat für das Beispiel Angriffsspiel ein Neuronales Netz mit 34 Eingabeneuronen für den Zustand, 10 versteckten Neuronen und ein Ausgabeneuron benutzt. Für das Training des Netzes werden mehrere Episoden gespielt. Dabei werden die Aktionen gierig bezüglich der aktuellen Wertfunktion gewählt. Nach einer bestimmten Anzahl gespielter Episoden werden die einzelnen Situationen gemäß der Kostenfunktion bewertet. Das veränderte Netz wird an alle Agenten verteilt und mit dem Training fortgesetzt bis keine Steigerung in der Spielstärke zu beobachten ist. Die Spielstärke kann zum Beispiel die Anzahl der geschossenen Tore pro Sequenz sein.

Situationen	Erlente Str.	Handprogrammierte Str.
1	0,645	0,0
2	0,225	0,01
3	0,45	0,0
4	0,655	0,31
5	0,39	0,14
6	0,445	0,145

Tabelle 1. Durchschnittlicher Torerfolg eines Angriffs im Falle der gelernten Strategie und der handprogrammierten Agenten. Situation 1-3 traten während des Trainings auf. Situation 4-6 waren neue Situationen.[3]

Das Resultat dieses erlernten Teamspiels hat das Team Brainstormers in [3] aufgezeigt. Es wurde mit der selbstständig erlernten und der handgeschriebenen Strategie jeweils in bestimmten Situationen gespielt. Sowohl in Situationen, die erlernt wurden, als auch in neuen Spielsituationen war die erlernte Strategie der handgeschriebenen deutlich überlegen, siehe Tabelle 1. Als wesentliches Merkmal konnte beobachtet werden, dass sich Spieler frühzeitig frei ließen und anspielbar waren. Durch die korrekte Vorhersage der Wertfunktion konnten Spielsituationen, die in eine Sackgasse laufen, vermieden werden. Indem diese Situationen mit sehr hohen Kosten bewertet werden, wurde das sogenannte 'Festspielen' kaum beobachtet. Im Gegensatz dazu wurden Spielzüge wie Doppelpässe erlernt, siehe Abbildung 4.

3.4 Taktiken intelligent auswählen

Die Einzelfähigkeiten und die Kooperationen auf dem Spielfeld sind die wesentlichen Verhaltensarten in einem Fußballspiel, die die Aktionsauswahl der Agenten betreffen. Doch muss der Agent auch Entscheidungen darüber treffen, welche Rolle er in einem Fußballspiel einnimmt. In den meisten Fällen gibt es wie

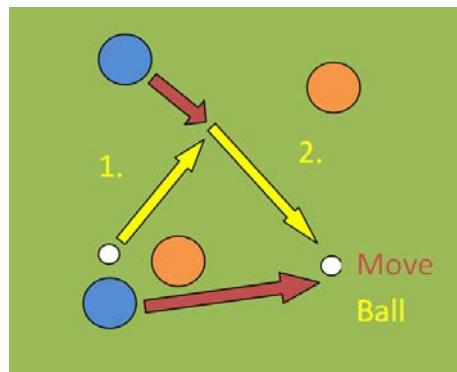


Abbildung 4. Beispiel einer gelernten Kooperation [3]

auch beim realen Fußball feste Zuteilungen in Bezug auf die Positionen (Angreifer, Mittelfeldspieler, Verteiger, Torwart). Während eines 'echten' Fußballspiels können jedoch bestimmte Ereignisse den Trainer veranlassen, Positionen/Rollen eines Spielers zu verändern. Hierzu muss ein Trainer das laufende Spiel bewerten können, um anhand dessen die Taktik bzw. die Aufstellung zu ändern. Im Folgenden wird ein Ansatz erläutert, mit dem die Taktik und Formation einer Mannschaft dynamisch verändert werden kann. Dies wird am Beispiel der *Four-legged League* gezeigt, in der üblicherweise mit vier Hunden gespielt wird, wobei ein Agent die feste Rolle des Torwarts übernimmt.

Um einem Agenten die Bewertung eines Spiels zu ermöglichen, werden statistische Daten über das laufende Spiel gesammelt. Diese sogenannten Erfahrungen werden über eine Zeit T aufgenommen und bestehen zum einen aus Aktionen die ein Spieler ausführt und zum anderen aus der Differenz des Punktestandes. Es werden folgende Erfahrungen ermittelt:

- Die Spielererfahrung stellt die Erfahrung aller Feldspieler dar und wird durch das Zählen aller Aktionen am Ball berechnet. Dazu gehören das Ball blockieren (BL) und das Schießen auf das gegnerische Tor (SH). Des Weiteren werden alle Positionsänderungen gezählt (CP) so wie der Wechsel vom Verteidiger zum Angreifer (DA) und vom Angreifer zum Verteidiger (AD). Dazu kommt die gesamte Zeit T , die angibt wie lange die Werte gezählt wurden, und die Zeiten als Angreifer (AT), Mittelfeldspieler (ST) und Verteidiger (DT) so wie die inaktive Zeit (IT).
- Die Torwarterfahrung wird an der Zeit gemessen, inder der Torwart in Bedrängnis ist (WT). In Gefahr befindet er sich, wenn er versucht den Ball zu schießen oder zu fangen.
- Der Kapitän repräsentiert die Tordifferenz (DS), die aus den erzielten Toren (SG) und den kassierten Toren (RG) berechnet wird.

Die Bewertung des Spiels wird über den Spielzustand repräsentiert, der aus allen Erfahrungen berechnet wird. Der Spielzustand (engl. game condition) GC ist ein reeller Wert aus dem Intervall $[0,1]$. Je näher dieser Wert an der Eins liegt, desto größer ist die Wahrscheinlichkeit, dass das Spiel gewonnen wird. Je näher er an der Null liegt, umso größer ist die Wahrscheinlichkeit, dass das Spiel verloren geht.

Der GC wird durch ein neuronales Netz repräsentiert. Die Erfahrungen, welche über die Zeit normiert werden, siehe Tabelle 2, sind der Eingabevektor X_{xp} für das Netz.

$$X_{xp} = \left(\sum_{i=1}^N \frac{E_{BL}^i}{N}, \sum_{i=1}^N \frac{E_{SH}^i}{N}, \sum_{i=1}^N \frac{E_{DA}^i}{N}, \sum_{i=1}^N \frac{E_{AD}^i}{N}, \sum_{i=1}^N \frac{E_{DT}^i}{N}, \sum_{i=1}^N \frac{E_{ST}^i}{N}, \sum_{i=1}^N \frac{E_{AT}^i}{N}, \sum_{i=1}^N \frac{E_{IT}^i}{N}, E_{WT}, E_{DS} \right) \quad (2)$$

Die Ausgabe ist der approxmierte GC, der durch die Erfahrungen gewonnen wird.

Erfahrung	Typ	Bedeutung
$E_{BL} = \frac{BL}{AC}$	Spielererfahrung	Ball blockieren
$E_{SH} = \frac{SH}{AC}$	Spielererfahrung	Torschüsse
$E_{DA} = \frac{DA}{CA}$	Spielererfahrung	Verteidiger zum Angreifer
$E_{AD} = \frac{AD}{CP}$	Spielererfahrung	Angreifer zum Verteidiger
$E_{DT} = \frac{DT}{TP}$	Spielererfahrung	Zeit als Verteidiger
$E_{ST} = \frac{ST}{AT}$	Spielererfahrung	Zeit als Mittelfeldspieler
$E_{AT} = \frac{AT}{IT}$	Spielererfahrung	Zeit als Angreifer
$E_{IT} = \frac{IT}{WT}$	Spielererfahrung	Inaktive Zeit
$E_{WT} = \frac{WT}{T}$	Torwarterfahrung	Zeit in Bedrängnis
$E_{DS} = \frac{1}{1+e^{SG-RG}}$	Kapitän	Tordifferenz

Tabelle 2. Normalisierte Erfahrungen der Agenten [4]

Das neuronale Netz wird mit Hilfe eines menschlichen Experten trainiert. Dafür haben Rojas und Atkinson [4], neun Spiele mit der Länge von 10 Minuten spielen lassen. Alle 30 Sekunden wurde das Spiel gestoppt, und der Experte wird zum letzten Zeitintervall nach dem Spielzustand befragt. Der menschliche Experte gab einen Wert im Intervall von $[0, 1]$ an, entsprechend der Definition des GC Wertes. Die Erfahrungen wurden aufgezeichnet und dienten im Zusammenhang mit dem vom Experten bestimmten GC als Trainingsdaten für das neuronale Netz. Als Lernalgorithmus wurde der Backpropagation verwendet. Allerdings wurden nur 40% der Trainingsdaten zum Trainieren benutzt und die anderen 60% zum Testen des Netzes.

Um eine Entscheidung über die Formation treffen zu können, muss eine Funktion für die taktische Einstellung definiert werden, die von einem bestimmten GC abhängig ist. Diese Funktion gibt die taktische Ausrichtung der Mannschaft an, die entweder defensiv oder offensiv eingestellt ist. Die Ausrichtung wird über einen Wert im Intervall $[0, 1]$ angegeben. Je näher der Wert an der Eins liegt, desto offensiver ist die Ausrichtung. Die Abbildung 5 zeigt taktische Funktionen, die eine offensive Ausrichtung repräsentieren, wenn die Mannschaft das Spiel kontrolliert. Ist die Mannschaft hingegen spielschwach, kann sie dennoch eine offensive Ausrichtung bevorzugen, siehe Abbildung 6. In Abhängigkeit zur taktischen Ausrichtung wird die Formation gewählt, die am besten zu spielen ist.

In der *Fourlegged League* spielen drei Feldspieler und ein Torwart. Dabei übernimmt meistens ein Spieler die Rolle des Verteidigers, einer die des Angreifers und der dritte Spieler, die des Supporters (Mittelfeldspieler). Eine Formation besteht aus einer Kombination, die sich aus einer möglichen Verteilung der drei Rollen auf die drei Feldspieler ergibt. Dies wird mit einem Formationsvektor F formalisiert,

$$F = (F_x, F_y, F_z) \text{ mit } F_x + F_y + F_z = N \text{ (Anzahl der Feldspieler)} \quad (3)$$

bei dem F_x, F_y, F_z für die Anzahl der Defensiv-, Mittelfeld- und Offensivspieler stehen. Die Zugehörigkeit zu einer Rolle wird anhand der Position auf dem

Spielfeld bestimmt. Dafür wird das Spielfeld in drei Zonen aufgeteilt, siehe Abbildung 7.

Um eine geeignete Formation auszuwählen, die die momentane Spielsituation und die taktische Funktion berücksichtigt, wird eine Fitnessfunktion ausgewertet. Damit die neu gewählte Formation keine großen Änderungen in der Rollenverteilung vornimmt, berücksichtigt die Fitnessfunktion auch die momentane Formation.

Der Algorithmus zur Auswahl der neuen Formation wird in den folgenden drei Schritten erläutert:

- 1) Die Fitness aller möglichen Formationen F_i wird berechnet.

$$fitness_{sim}(F_i, F') = \frac{\sqrt{(F_i - F')^2}}{\sqrt{18}} \quad (4)$$

$$fitness_{att}(F_i) = 1 - |att - (F_x * \frac{0}{N} + F_y * \frac{0.5}{N} + F_z * \frac{1}{N})| \quad (5)$$

$$fitness(F_i, F') = \alpha * fitness_{sim}(F_i, F') + (1 - \alpha) * fitness_{att}(F_i) \quad (6)$$

F' ist die aktuelle Formation und N die Anzahl der Feldspieler. Der Parameter $\alpha \in [0, 1]$ wird von einem Experten gewählt. att bezeichnet die taktische Funktion abhängig vom gegenwärtigen GC. Für das Beispiel mit drei Feldspielern werden so 10 verschiedene Fitness Werte berechnet.

- 2) Im nächsten Schritt wird für alle möglichen F_i mit $i \in [1, 10]$ eine proportionale Fitness zur gesamten Fitness berechnet.

$$fitness_{pi} = \frac{fitness_i}{\sum_{i=1}^{10} fitness_i} \quad (7)$$

Diese werden anschließend absteigend sortiert.

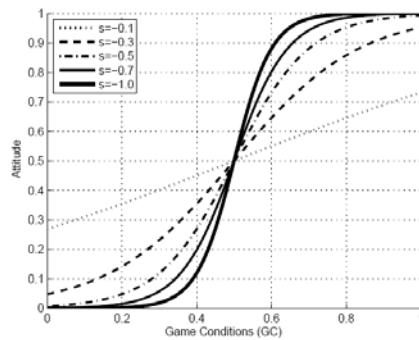


Abbildung 5. Beispiel für eine Taktikfunktion [4]

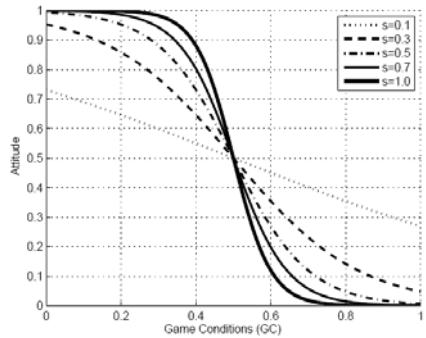


Abbildung 6. Beispiel für eine Taktikfunktion [4]

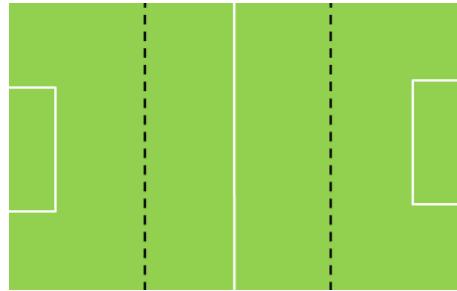


Abbildung 7. Aufteilung des Spielfeldes in drei Zonen (Verteidiger, Mittelfeld, Anfreifer)

- 3) Schließlich wird zufällig eine Variable $\beta \in [0, 1]$ bestimmt. Nun werden die proportionierten Fitness Werte der Größe nach in absteigender Reihenfolge summiert, bis der kumulierte Wert größer oder gleich β ist. Als Ausgabe wird die zugehörige Formation, der des zu letzt addiertem Fitness Wertes i , zurückgegeben.

Rojas und Atkinson [4] haben festgestellt, dass das trainierte neuronale Netz mit den Expertenaussagen aus den Testfällen übereinstimmte. Somit kann davon ausgegangen werden, dass die Agenten ihre momentane Spielsituation gut einschätzen können. Es wurden 24 Testspiele durchgeführt, wobei vier unterschiedliche Taktiken für den Gegner zur Verfügung standen. Zum einen wurde mit der dynamischen Formationsauswahl gespielt und zum anderen wurden die gleichen 24 Spiele mit einer statischen Formation bestritten. Für jedes gewonnene Spiel wurden drei Punkte vergeben, für jedes verlorene Null und für Spiele, die Unentschieden endeten, ein Punkt. Die Ergebnisse wurden in der Tabelle 3 zusammengetragen.

Team	Punkte
Dynamische Formation	38
Statische Formation	18

Tabelle 3. Ergebnisse der Testspiele [4]

4 RoboCup vs. Computerspiele

Im RoboCup treten Teams verschiedener Universitäten der Welt im Wettkampf gegeneinander an, um zu zeigen, welcher reale oder simulierter Roboter am Intelligentesten erscheint und die Gegenspieler in den verschiedenen Ligen besiegt. Die Entwickler bemühen sich, die Konzepte und Techniken so einzusetzen, um Schwachstellen der Gegner zu instrumentalisieren oder sich Vorteile zu verschaffen, z.B. durch einen schnellere und genauere Fortbewegung. Zum Teil werden auch Methoden der Computational Intelligence in den verschiedenen Bereichen eingesetzt. Ein erfolgreiches Beispiel sind die AIBO Roboterhunde des Teams Microsoft Hellhounds der Universität Dortmund. Sie konnten mit der Hilfe evolutionärer Algorithmen die Schnelligkeit im Bereich des Laufens optimieren und waren so den Gegnern überlegen. Dies führte bei den RoboCup Wettkämpfen zu zahlreichen Titeln, siehe <http://www.microsoft-hellhounds.de>.

Dennoch haben die Entwickler von Robotern mit Schwierigkeiten der realen Bedingungen (z.B. verrauschte Sensordaten) zu kämpfen. Diese erschweren außerdem den Einsatz der Techniken der Künstlichen Intelligenz.

Bei Computerspielen werden Methoden der Künstlichen Intelligenz nicht schwerpunktmäßig eingesetzt, um einen Gegenspieler zu besiegen. Vielmehr soll dem Nutzer des Spiels suggeriert werden, die computergesteuerten Spieler würden sich menschlich bzw. natürlich verhalten. Es würde den Spielspaß trüben, wenn der Spieler schon nach wenigen Minuten dem Computergegner unterlegen wäre. Somit führt der Einsatz dieser Techniken zu zwei unterschiedlichen Zielen. Beim RoboCup steht der Wettstreit im Vordergrund. Bei Computerspielen hingegen geht es darum, das Verhalten der Charaktere natürlicher und vertrauter wirken zu lassen. Beide Ziele sind im Moment noch schwer zu realisieren. Obwohl dem Computerspieler das Weltmodell in allen Einzelheiten jederzeit gegeben ist, stellt die Komplexität des Verhaltens menschlicher Wesen ein großes Problem dar. Es sind nicht nur eine hohe Rechenleistung, sondern auch eine Menge an handingegebenen Randbedingungen erforderlich, um eine Vielzahl von Charakteren zu steuern. Diese Randbedingungen sollen irrationales Verhalten der Computerfiguren vermeiden.

5 Fazit

Die Künstliche Intelligenz von autonomen mobilen Robotern wird beim RoboCup in den Simulationen erfolgreich eingesetzt. Bei realen Robotern erschweren die Perzeption der Umwelt und das Erlernen in Echtzeit den Einsatz und die

Evaluation solcher Methoden. Dennoch werden Ansätze auch bei realen Robotern verfolgt (vgl. [4]).

Die Autoren der vorgestellten Ansätze konnten positive Ergebnisse mit dem Einsatz von CI in der Verhaltenssteuerung erzielen. Die erlernten Verfahren sind gegenüber statischer handgeschriebener Verhalten überwiegend erfolgreicher. Dennoch werden beim RoboCup immer noch handgeschriebene Routinen eingesetzt, da diese schneller zu implementieren und einfacher zu testen sind.

Literatur

- [1] R. Siegwart, I.R. Nourbakhsh: Introduction to Autonomous Mobile Robots, The MIT Press, 2004.
- [2] G. Görz und C.-R. Rollinger: Handbuch der Künstlichen Intelligenz, Oldenbourg Verlag, 3. Aufl. 2000.
- [3] M. Riedmiller, T. Gabel, R. Hafner, S. Lange und M. Lauer: Die Brainstormers: Entwurfsprinzipien lernfähiger autonomer Roboter, In Informatik-Spektrum 29(3) 175–190, Springer, Juni 2006.
- [4] D. Rojas, J. Atkinson: Generating Dynamic Formation Strategies based on Human Experience and Game Conditions, Symposium RoboCup Atlanta, 2007.

Computerspiele im Fokus der Sozialpsychologie

Seminar Computational Intelligence & Games

Moritz Hofmann

Sommersemester 2007
Lehrstuhl für Algorithm Engineering
Fachbereich Informatik
TU Dortmund

1 Computerspiele in der Gesellschaft

1.1 Motivation

Machen Computerspiele aggressiv? Durch diese Frage sind Computerspiele in den letzten Jahren zu einem gesellschaftlich stark diskutierten Thema geworden. Auf der Suche nach Erklärungen für jugendliche Amokläufer und steigende Aggressivität in Schulen werden Computerspiele besonders von den Medien häufig als Auslöser herangezogen. Doch auch abseits von gewalthaltigen Spielen und aggressiven Spielern haben Computerspiele gesellschaftlich und wissenschaftlich an Bedeutung gewonnen, da sich besonders bei jüngeren Nutzern das Medienverhalten stark verändert hat. Zu betrachten sind insbesondere Phänomene wie Computerspielsucht, aber auch mögliche positive Effekte durch die Nutzung von Computerspielen. In dieser Arbeit werden die unterschiedlichen sozialpsychologischen Aspekte von Computerspielen dargestellt, wobei schwerpunktmaßig auf gewalthaltige Computerspiele und ihre Auswirkungen eingegangen wird.

1.2 Verbreitung von Computerspielen

Computerspiele stellen eine weit verbreitete Freizeitbeschäftigung dar, ein Großteil der jüngeren Bevölkerung nutzt Computerspiele. Die meisten Spieler sind männliche Jugendliche und junge Erwachsene, von denen ca. $\frac{3}{4}$ zumindest gelegentlich spielen [11]. Entgegen der allgemeinen Annahme spielt auch jede zehnte weibliche Person zwischen 12 und 25 Jahren ein oder mehrmals pro Woche am Computer. Allerdings unterscheidet sich die Häufigkeit der Nutzung deutlich zwischen den Geschlechtern, nur etwa 5% der weiblichen Spieler spielen täglich, bei den männlichen Spielern liegt dieser Wert bei 10%. [1]

Daneben steigt auch der Anteil der Computerspieler unter Kindern (unter 14 Jahre) stark an. Nach einer Studie zum Medienumgang mit Kindern[12] hatten im Jahr 2006 81% der 6- bis 13-Jährigen Erfahrungen mit Computerspielen, 3 Jahre zuvor waren es erst 70%. Auch der Anteil der spielenden Mädchen steigt kontinuierlich an. Während im Jahr 2000 58% der Mädchen angaben, mindestens einmal pro Woche Computerspiele zu nutzen, waren es im Jahr 2002 bereits 65%,

vgl. Abbildung 1. Mädchen und Jungen unterscheiden sich auch in der Computerspieldauer, Mädchen spielen in der Regel kürzer als Jungen. Fast doppelt so viele Jungen wie Mädchen spielen intensiv, also regelmäßig länger als eine Stunde am Stück, vgl. Abbildung 2. [5]

Die hohe Nutzung legt die Vermutung nahe, dass Computerspiele Auswirkungen auf soziale Kompetenzen haben und langfristig die Gesellschaft verändern könnten. Auf Grund der im Vergleich zu anderen Medien relativ kurzen Zeit, seit der Computerspiele verfügbar sind, steht die Forschung hier noch am Anfang und besonders Langzeitauswirkungen sind erst ansatzweise erforscht.

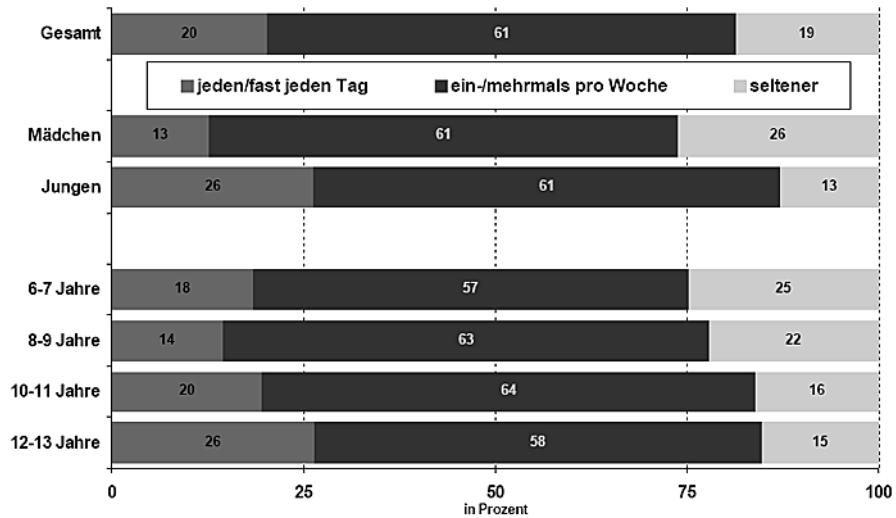


Abbildung 1. Nutzungshäufigkeit von Computerspielen bei Kindern (Kinder und Medien Studie 2006) [12]

1.3 Vergleich mit Fernsehkonsum

Schon lange ist bekannt, dass ein zu hoher Fernsehkonsum negative Auswirkungen hat. Insbesondere Kinder und Jugendliche können durch übermäßigen Konsum in ihren kognitiven Fähigkeiten, zum Beispiel Konzentration, mathematische Fähigkeiten, Lese- und Sprachverständnis, beeinträchtigt werden. Außerdem leidet die soziale Kompetenz darunter, dass die Freizeit hauptsächlich vor dem Fernseher statt mit anderen Personen verbracht wird. Die Passivität vor dem Fernseher hat weitere negative Konsequenzen wie Trägheit und Bewegungs mangel.

Im Vergleich zum Fernsehkonsum binden Computerspiele den Nutzer wesentlich stärker an das Medium und fordern dessen Aktivität, ein rein passiver Kon sum ist meist nicht möglich. Daneben ändert sich die Handlung, anders als im

Fernsehen, über die Spieldauer nicht oder nur unwesentlich, es gibt auch keine Unterbrechungen durch Werbung oder Programmwechsel. Dadurch und durch die aktive Mitbestimmung der Handlung identifiziert sich der Spieler mit der Spielfigur und verinnerlicht dessen Handlungen über einen längeren Zeitraum.

Man kann deshalb davon ausgehen, dass die Erkenntnisse über die Folgen von übermäßigem Fernsehkonsum und insbesondere die Wirkung von Gewaltszenen auf Kinder und Jugendliche auch auf die Nutzung von Computerspielen übertragbar sind und bekannte Effekte eher mit noch stärkerer Wirkung zu erwarten sind. Hingegen sind Folgen der Passivität, wie zum Beispiel die Reduzierung der kognitiven Leistung, vermutlich nicht medienübergreifend.

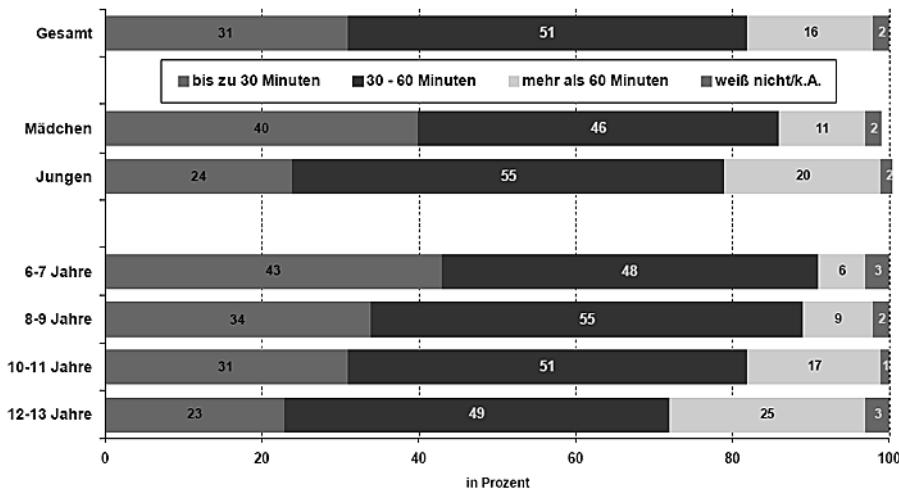


Abbildung 2. Spieldauer von Computerspielen bei Kindern (Kinder und Medien Studie 2006) [12]

1.4 Nutzungsmotive

Auf Grund der hohen Verbreitung von Computerspielen stellt sich die Frage, wieso diese überhaupt genutzt werden. Nach den Motiven zur Nutzung befragt, geben Computerspieler eine Vielzahl von Gründen an. Diese lassen sich zu den folgenden Punkten zusammenfassen, aus [11]:

Strukturelle Kopplung zur Realität Nutzungsmotive, die sich auf thematische Zusammenhänge zur Lebens- und Gedankenwelt des Spielers beziehen, bezeichnet man als strukturelle Kopplung, von der zwei Ausprägungen existieren. Eine parallele Kopplung tritt auf, wenn der Nutzer Bekanntes auch in der virtuellen Welt erleben möchte. Dies ist zum Beispiel bei Amateursportlern der

Fall, die ihren Sport auch als Computerspiel praktizieren, aber auch bei aggressiven Jugendlichen, die gewalthaltige Spiele bevorzugen.

Davon unterscheidet man die kompensatorische Kopplung, bei der der Nutzer Spielinhalte wählt, die er in der Realität gern erleben würde, dies aber nicht möglich ist. Besonders häufig ist dies bei Rennspielen und Simulationen (z.B. Flugsimulator oder Achterbahnsimulation) anzutreffen, aber auch körperlich beeinträchtigte beziehungsweise unsportliche Menschen, die Sportspiele nutzen, fallen in diese Kategorie.

Die meisten Nutzungsmotive weisen Merkmale einer strukturellen Kopplung auf, so dass diese in Kombination mit einem der folgenden Aspekte auftritt. Außerdem kann die Grenze zwischen paralleler und kompensatorischer Kopplung fließend sein und eine Spielmotivation in beide Kategorien passen, wenn in der Realität des Spielers vorkommende Elemente aufgegriffen und durch Irreales erweitert werden.

Herausforderung / Wettbewerb Besonders jugendliche Computerspieler wollen sich mit anderen messen, ähnlich wie beim Sport oder bei Gesellschaftsspielen in der realen Welt. Viele Befragte nennen auch die Herausforderung, vom Spiel gestellte Probleme zu lösen, die Spielfigur weiterzuentwickeln oder das gesamte Spiel zu Ende bringen zu wollen als Motiv.

Geselligkeit Online-Spiele und andere Genre, die für mehrere Spieler geeignet sind, fördern soziale Kontakte und werden von manchen Spielern gerade aus diesem Grund genutzt. Im Hinblick auf das Wettbewerbsmotiv suchen Computerspieler den kommunikativen Austausch mit Gleichgesinnten. Dies geschieht indirekt oft auch bei Einzelspieler-Spielen, in dem sie sich über alternative Kommunikationswege (online / offline) darüber unterhalten oder über ihren Spielfortschritt beziehungsweise ihre Erfahrungen berichten.

Flow-Erlebnis Das „Flow-Erlebnis“ ist ein psychologisches Phänomen, dass sich auch bei der Nutzung von Computerspielen einstellen kann und den Wunsch des Stressabbaus und der Entspannung erfüllt. Der Spieler widmet sich dabei vollends dem Spiel und vergisst alles für ihn Nebensächliche. Das Spiel darf den Nutzer dabei weder über- noch unterfordern, er muss die Situation unter Kontrolle haben. „Im flow-Zustand folgt Handlung auf Handlung, und zwar nach einer inneren Logik, welche kein bewusstes Eingreifen von Seiten des Handelnden zu erfordern scheint. Er erlebt den Prozess als ein einheitliches ‘Fließen’ von einem Augenblick zum nächsten, wobei er Meister seines Handelns ist und kaum eine Trennung zwischen sich und der Umwelt, zwischen Stimulus und Reaktion, oder zwischen Vergangenheit, Gegenwart und Zukunft verspürt.“, erklärt Mihaly Csikszentmihalyi [4] das Phänomen.

Körperlich entspricht dieser Zustand einer gegenseitigen Anpassung von Atmung, Herzschlag und Blutdruck. Das limbische System, das im Gehirn für Emotionen zuständig ist, ist in dieser Konstellation besonders aktiv und verhilft dem Spieler zu der oben genannten Gefühlssituation.

Die wenigsten Spieler kennen direkt das Flow-Erlebnis als Nutzungsmotiv, allerdings beschreiben viele die damit verbundene emotionale Situation. Eng verknüpft ist es auch mit dem folgenden Motiv.

Flucht aus dem Alltag Viele Nutzer geben an, mit Computerspielen ihre Langeweile vertreiben und der realen Welt entfliehen zu wollen. Insbesondere Nutzer mit sozialen oder finanziellen Problemen gehen diesen so aus dem Weg anstatt sie zu lösen. In Computerspielen können sie sich ihre Fantasiewelt aussuchen und ein virtuelles, problemloses Leben führen, dass sie so in der realen Welt nicht vorfinden. Computerspiele bieten aber auch kurzfristige Rückzugsmöglichkeiten, wenn gerade kein realer Spielpartner verfügbar ist; Computerspiele stehen hier also gleichberechtigt neben anderen Freizeitbeschäftigungen.

Ausübung von Macht Computerspiele bieten durch den hohen Grad an Interaktivität im Gegensatz zu anderen Medien die Möglichkeit, das Spielgeschehen aktiv zu beeinflussen und dadurch Macht auszuüben. Besonders Nutzer, die in der Realität wenig Gelegenheiten haben, etwas zu kontrollieren oder zu beeinflussen, nutzen Computerspiele als Ausgleich dafür. Auch Gewalt ist eine Form von Macht, bei der das gewünschte Machgefühl besonders ausgeprägt ist. Fraglich ist, ob die Spieler ihre Nutzungsmotive ehrlich angeben und sich derer auch voll bewusst sind, auch unterbewusste Motive können zum Spielen führen. Diese lassen sich jedoch kaum erfassen und bleiben meist unberücksichtigt.

2 Gewalthaltige Spiele

2.1 Definition

Virtuelle Gewalt kann in verschiedenen Ausprägungen vorkommen. In den Medien werden gewalthaltige Spiele oft mit so genannten „Ego-Shootern“ gleichgesetzt, deren Spielverlauf von Gewalt geprägt ist und die oft durch besonders realistische Gewaltszenen auffallen. Die Forschung differenziert hier weiter und sieht auch in anderen Genres gewalthaltige Elemente. Danach lassen sich gewalthaltige Computerspiele als solche definieren, die vom Nutzer Handlungen mit dem Ziel erfordern, realen oder computergesteuerten Spielcharakteren zu schaden oder zu verletzen, vgl. [11]. Recht unterschiedlich wird dabei die notwendige Realitätsnähe dieses Spielcharakters einbezogen, ob also menschenähnliches Aussehen oder menschliche Verhaltensweisen des Gewaltopfers Voraussetzungen für die Einstufung eines Spiels als gewalthaltiges Spiel sind. Einige Spiele sind demnach nicht eindeutig gewalthaltig oder nicht, sondern werden von Studie zu Studie anders beurteilt.

2.2 Formen von Gewalt

Gewalt in Computerspielen lässt sich in verschiedene Kategorien einteilen. Wie schon erwähnt spielt die Ähnlichkeit des Opfers mit einem menschlichen Wesen

eine große Rolle, aber auch die damit verbundene Realitätsnähe der Darstellung der Gewaltszene ist für die Beurteilung der Auswirkungen (und damit auch der Eignung für Kinder und Jugendliche) entscheidend. Obwohl im Fernsehen die Gewaltpflicht meist menschlich sind, dürfen viele Gewaltszenen auch von jungen Zuschauern gesehen werden - Details der Gewalthandlung werden nicht gezeigt, die Szene nicht als Selbstzweck beziehungsweise gewaltverherrlichend, sondern verharmlost dargestellt. Außerdem ist die Häufigkeit von Gewaltszenen ein Unterscheidungskriterium. Spiele mit sehr seltenen, kurzen violenten Szenen sind von Spielen, deren Haupthandlung gewaltbeherrscht ist oder deren Spielzweck in der Ausübung von Gewalt besteht, abzugrenzen. Parallel dazu differenziert man den Gewaltgehalt auch bei Computerspielen an Hand dieser Merkmale:

- Menschlichkeit des Opfer,
- Realitätsnähe der Darstellung,
- Detailgrad der Gewaltszenen,
- Häufigkeit der Gewaltszenen,
- Notwendigkeit der Gewalt für den Spielverlauf.

2.3 Verbreitung gewalthaltiger Spiele

Die Verbreitung von Gewalt in aktuellen Computerspielen ist demnach nicht eindeutig zu klären, da je nach Definition der Gewalthaltigkeit Spiele anders kategorisiert werden. Allen Ansätzen gemein ist aber, dass gewalthaltige Computerspiele keine Ausnahme darstellen. Lisbeth Schierbeck und Bo Carstens untersuchten 1998 alle 338 in Dänemark herausgebrachten Computerspiele auf deren Gewaltgehalt[14]. Einbezogen wurden sowohl PC- als auch Konsolenspielen, nur reine Online-Spiele wurden nicht betrachtet. Ihre Ergebnisse sind größtenteils auf Deutschland übertragbar, da die meisten Spiele international publiziert werden und sich die lokalisierten Versionen grafisch und inhaltlich meist nicht unterscheiden. Schierbeck und Carstens stellten fest, dass etwas mehr als die Hälfte der Spiele Gewalt enthielt, wobei je nach Genre große Unterschiede auftraten. Einen besonders hohen Anteil an violenten Spielen wiesen (Echtzeit-)Strategien (89%) Action- (86%) und Simulationsspiele (54%) auf, wohingegen Sportspiele (10% violente Titel) und virtualisierte klassische Spiele (Kartenspiele, Puzzle und dergleichen) kaum violent waren, vgl. Abbildung 3.

Die amerikanische Organisation „Children Now“ analysierte im Jahr 2001 in einer Studie[3] aktuelle Computer- und Videospiele auf deren Gewaltgehalt. Sie legte eine weit umfassende Definition von Gewalt zu Grunde, nach der von den analysierten 70 Spielen 89% Gewalt enthielten, davon etwa je zur Hälfte realistische Gewalt sowie Comic-ähnliche oder abstrakte Gewalt. Für 41% der Spiele war die Gewalt essentieller Bestandteil der Handlung und zum erfolgreichen Spielen erforderlich, in 17% der Spiele stellten Gewaltszenen den Spielzweck dar. Der Anteil an realistischer Gewalt deckt sich damit mit den Ergebnissen der dänischen Studie.

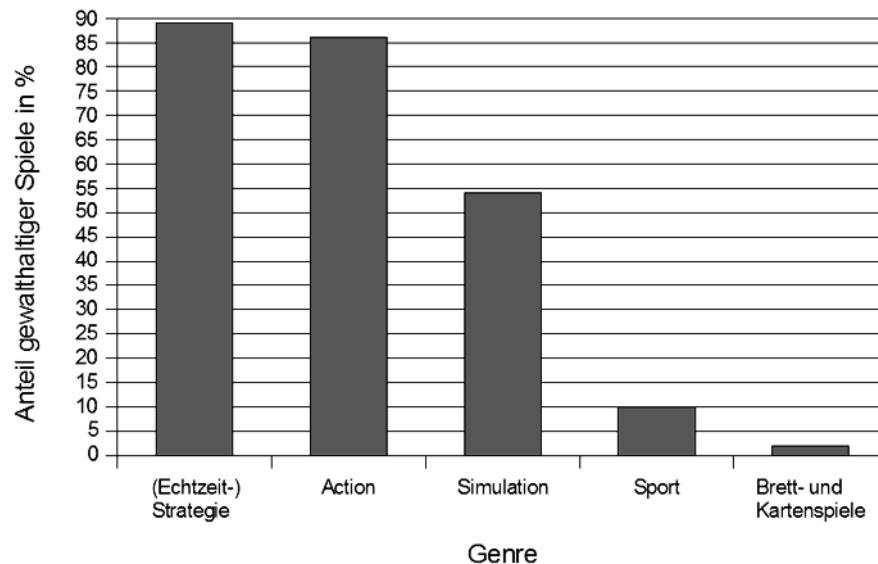


Abbildung 3. Gewalthaltige Computerspiele aufgeteilt nach Genre

2.4 Problemmerkmale

Gewalt in Computerspielen ist besonders deshalb problematisch, da sie gerechtfertigt oder sogar belohnt wird. Anders als in der realen Welt folgt im Anschluss an Gewalthandlungen keine Bestrafung, stattdessen kommt der Nutzer in der Spielhandlung weiter oder erhält positive Leistungen wie zusätzliche Waffen, Fähigkeiten oder ähnliches. Durch die steigende Realitätsnähe der Spiele und damit auch der Gewaltszenen lassen sich diese Handlungen (auch unterbewusst) leichter auf die reale Welt übertragen. Ein weiteres Problemmerkmal besteht darin, dass Computerspielcharaktere als Opfer deutlich mehr Gewalt aushalten als reale Menschen und selbst nach tödlichen Verletzungen schnell wieder „gesund“ werden oder wiederauferstehen. Negative Folgen wie Schmerzen, psychische Beeinträchtigung oder langandauernde Genesung werden selten bis nie gezeigt. Zusammenfassend lässt sich also sagen, dass die Darstellung von Gewalt in Computerspielen zunehmend realer wird, ihre Auswirkungen aber heruntergespielt werden und ihre Erfordernis übertrieben und gerechtfertigt wird.

2.5 Effekte gewalthaltiger Spiele

Es wird oft vermutet, dass violente Computerspiele auf Grund dieser Problemmerkmale Auswirkungen auf deren Nutzer haben können. In anderen Bereichen der Mediengewalt sind derartige Auswirkungen bereits intensiv erforscht, Computerspielen widmen sich ernstzunehmende Studien erst seit einigen Jahren. So ist noch nicht eindeutig geklärt, in welchem Umfang und bei welchen Nutzergruppen negative Folgen auftreten. Besonders die naheliegende Vermutung, dass

das Spielen gewalthaltiger Computerspiele die Aggressivität und reale Gewaltbereitschaft fördere und Spieler zu Gewalttaten animieren, ist stark umstritten, da diese psychischen Phänomene von einer Vielzahl von Aspekten abhängen und sich Veränderungen nur über einen langen Zeitraum bemerkbar machen und schwer messbar sind.

Kurzfristige Reaktionen Leicht feststellbar sind kurzfristige, körperliche Reaktionen. Diese können als Basis zur Erforschung und Deutung weitergehender Langzeiteffekte dienen. In mehreren Studien, vgl. z.B. [9], [2], registrierten die Forscher einen erhöhten Blutdruck und eine höhere Herzfrequenz während und kurz nach dem Spielen gewalthaltiger Spiele im Vergleich zu Nutzern gewaltloser Spiele. Auch Wolfgang Frindte und Irmgard Obwexer widmeten sich diesen Effekten in einer Studie [7]. Sie wollten damit folgende, aus anderen Befunden abgeleitete, Hypothesen belegen:

Hypothese 1: „Das kurzfristige Spielen gewalthaltiger Computerspiele führt im Vergleich zur Nutzung gewaltfreier Computerspiele zu negativer Stimmungslage, zur Erhöhung des Blutdrucks, zur Erhöhung der Pulsfrequenz und zur Erhöhung der momentanen aggressiven Neigungen.“ [7]

Hypothese 2: „Die in Hypothese 1 angenommenen Veränderungen der internalen Zustände und der momentanen aggressiven Neigungen zeigen sich vor allem bei Personen mit hoher Aggressionsbereitschaft.“ [7]

Als Stichprobe dienten 20 erwachsene, männliche Personen, die bereits langjährige Erfahrung mit Computerspielen hatten. Sie spielten jeweils 10 Minuten ein gewalthaltiges beziehungsweise ein gewaltloses Spiel. Vor, während und nach jedem Spiel wurden sowohl der Blutdruck und Puls gemessen als auch die Stimmungslage vorher und nachher subjektiv an Hand einer Skala erfasst, vgl. Abbildungen 4 und 5.

Der Blutdruck veränderte sich bei beiden Spielen kaum. Vom gewaltlosen Spiel wurden auch der Puls und die aggressiven Neigungen kaum beeinflusst. Bei den Nutzern des gewalthaltigen Spiels konnten dagegen deutliche Auswirkungen festgestellt werden. Während des Spiels stieg der Puls um fast 20% an, auch kurz nach Ende des Experiments lag dieser noch signifikant oberhalb des Ausgangswerts und des Vergleichswerts beim gewaltlosen Spiel. Außerdem verschlechterte sich die subjektive Einschätzung der Spieler in Bezug auf momentane aggressive Neigungen durch das Gewaltspiel deutlich, während bei den Vergleichspersonen nur ein minimaler Anstieg zu verzeichnen war. Damit ist die Hypothese 1 zumindest teilweise bestätigt.

Nach der zweiten Hypothese sind diese Reaktionen von der Aggressionsbereitschaft des Spielers abhängig. Auch dies belegten Frindte und Obwexer mit ihrem Experiment, hierfür erfassten sie die Aggressionsbereitschaft ihrer Probanden. Spieler mit einer friedlichen Einstellung wurden auch von gewalthaltigen Spielen in ihren Neigungen kaum beeinflusst, bei Spielern mit relativ ausgeprägter Aggressionsbereitschaft führt die Nutzung violenter Spiele dagegen zu

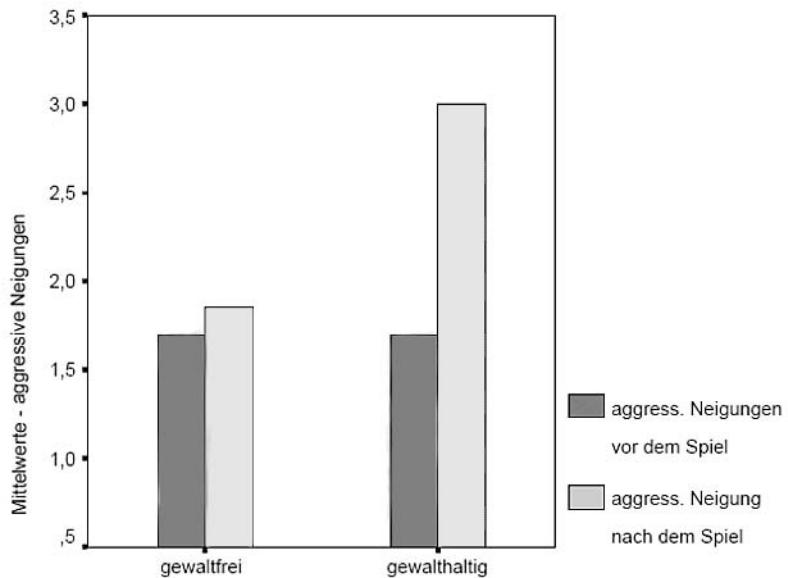


Abbildung 4. Subjektive Einschätzung der Testpersonen bzgl. ihrer aggressiven Neigungen [7]

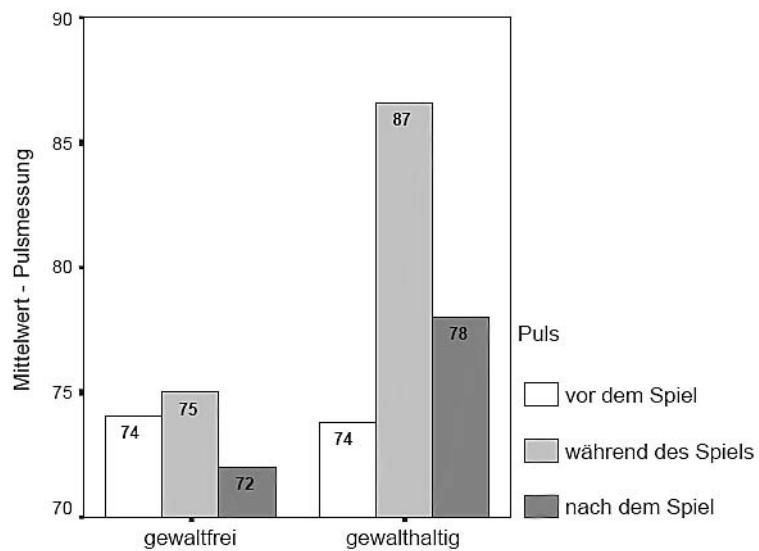


Abbildung 5. Pulswerte der Testpersonen beim Spielen [7]

einer starken Erhöhung momentaner aggressiver Neigungen. Sie kommen damit zu dem Schluss, dass kurzfristiges Spielen gewalthaltiger Computerspiele nicht zwingend Aggressivität fördert, sondern dieser Effekt vorwiegend bei aggressiven Persönlichkeiten auftritt.

Diese beiden Hypothesen können aber nicht zwangsläufig zusammenhängend betrachtet werden, höhere körperliche Erregung muss nicht zu einer höheren Aggressionsbereitschaft führen. Dies belegt eine Studie von Michele J. Fleming und Debra J. Rickwood [6]. In der Studie mit 71 Kindern zwischen 8 und 12 Jahren traten keine Zusammenhänge zwischen festgestellten höheren Erregungswerten nach violenten Computerspielen und aggressiven Neigungen auf.

Langfristige Effekte Die dauerhafte Nutzung gewalthaltiger Computerspiele kann auch längerfristige, persönlichkeitsverändernde Auswirkungen haben. Die Erforschung dieser Effekte in Langzeitstudien ist allerdings deutlich fehleranfälliger und aufwändiger als kurzfristige Studien, daher existieren in diesem Bereich auch erst wenig ernstzunehmende empirische Studien.

Michael Kunczik und Andreas Zipfel [11] fassten die Ergebnisse mehrerer solcher Untersuchungen zusammen und kamen zu dem Ergebnis, dass wiederholter Konsum violenter Computerspiele längerfristig die Tendenz erhöhen kann, die Welt feindseliger einzustufen. Außerdem wird Wissen über Abläufe von Gewalthandlungen erlernt und die Einschätzung verändert, in welchen Situationen dieses Wissen einzusetzen sind. Dies führt zu einer positiven Einstellung gegenüber Gewalthandlungen und die Betrachtung von Gewalt als universeller Problemlösungsansatz. Diese Desensibilisierung und Gewöhnung an Gewalt verringert Mitleidsgefühle und erhöht die Neigung zu härterer Bestrafung. Durch die gerechtfertigte Anwendung von Gewalt in Computerspielen werden bei den Nutzern vorhandene aggressive Emotionen und Gedanken legalisiert und gefördert. Dies zeigten Krahé und Möller in einer Untersuchung [10]. Sie ermittelten, dass sich gewalthaltige Spiele hauptsächlich bei bereits aggressiven Spielern, die im Vorfeld negativ auffällig geworden sind oder psychische Störungen aufwiesen, aggressionssteigernd auswirken.

Neben sozialer Abstumpfung gegenüber Gewalt und deren Förderung stellt sich vor allem in den Medien die Frage, ob violente Computerspiele auch Auslöser für Gewalthandlungen sein kann. Diese Nachahmung von im Computerspiel erlernter Verhaltensweisen auf die Realität wird als *Transfer* bezeichnet, welche in Bezug auf Gewalthandlungen in der Forschung stark umstritten sind. Treten sie auf, so ist dies auf Unzulänglichkeiten in der so genannten Rahmungskompetenz zurückzuführen, „das heißt [auf die] Fähigkeit, Reizeindrücke dem richtigen Zusammenhang bzw. der richtigen ‘Welt’ (reale Welt, virtuelle Welt) zuzuordnen und dementsprechend auf die richtigen Wahrnehmungs- und Verhaltensschemata zugreifen zu können.“ [11]

Manuel Ladas vertritt wie viele andere die Ansicht, dass solche Transfers nicht auftreten können: „Ein unkontrollierter Transfer von komplexeren gewaltbezogenen Handlungsschemata zwischen virtueller (Computerspiele) und ‘realer’ Welt ist aufgrund der verschiedenen Sinnzusammenhänge im Rahmen des vorgestell-

ten psychologisch-konstruktivistischen theoretischen Modells kaum vorstellbar: 'Reale' Gewalt soll verletzen, virtuelle Gewalt ist ausschließlich in die Funktionszusammenhänge des Computerspiels eingebunden und kann so prinzipbedingt keine 'realen' Schäden hervorrufen.“ [13]

Meistens hat die Nutzung gewalthaltiger Computerspiele also keine beziehungsweise nur eine kurzfristige Wirkung, allerdings kann es zu einer negativen Beeinflussung aggressiver, instabiler Persönlichkeiten kommen. Diese besonders gefährdeten Personen fasst man als „Risikogruppen“ zusammen.

2.6 Risikogruppen

Zu den Gruppen mit erhöhtem Risiko bei der Nutzung gewalthaltiger Computerspiele zählen insbesondere [11]:

- Sehr junge Spieler (unter 12 Jahren)
- Spieler aus gewalttätiger Umgebung
- Spieler mit erhöhter Reizbarkeit / Aggressivität

Diese Nutzer sind für die oben genannten Effekte besonders empfänglich. Kinder können sich von violenten Inhalten nicht genügend distanzieren und erlernen Gewalthandlungen unreflektiert, während aggressive Persönlichkeiten durch violente Spiele in ihren Neigungen bestärkt werden. Bei diesen Risikogruppen muss also ganz besonders auf einen kontrollierten Umgang geachtet werden, da sonst die Gefahr von Gewalthandlungen und dauerhafter, negativer Charakterveränderungen besteht.

3 Soziale Auswirkungen von Computerspielen

Auch wenn sich die Forschung mehrheitlich auf die Effekte gewalthaltiger Computerspiele konzentriert, sind auch allgemein bei Computerspielnutzung und insbesondere bei dem Konsum von Online-Spielen soziale Auswirkungen festzustellen. Diese treten insbesondere bei einer langfristig hohen Nutzungsintensität auf: Die Mehrheit der Spieler weist einen maßvollen Umgang mit Computerspielen auf. Es existiert allerdings ein Anteil bei dem die Nutzung sehr extensiv ist und andere Freizeitbeschäftigungen verdrängt beziehungsweise den Tagesablauf bestimmt. Hier sehen Psychologen und Sozialforscher problematische soziale Auswirkungen auf diese Nutzer und deren Umfeld, vgl. [12].

3.1 Vernachlässigung sozialer Kontakte

Übermäßiger Konsum von Computerspielen kann dazu führen, dass soziale Kontakte vernachlässigt werden. Durch die Umstellung des Tagesrhythmus und damit der Anpassung an das (zumeist Online-)Spiel kommt es zu Einschränkungen des gesellschaftlichen Lebens. Das kann bis zu schulischen oder beruflichen Veräumnissen und familiärer Abkapslung führen, da selbst gesteckte Ziele in der

Spielwelt als wichtiger als reale Verpflichtungen angesehen werden. Häufig werden damit zusammenhängend auch andere Freizeitaktivitäten und Freunde vernachlässigt. Der Kontakt wird nur noch zu „Online-Freunden“ gehalten, das Interesse an Nicht-Spielern flaut ab. Da Online-Spiele häufig so gestaltet sind, dass man rund um die Uhr Spielen kann und durch Internationalisierung auch ständig ausreichend Mitspieler vorhanden sind, sind Nutzer dieses Genres besonders betroffen.

3.2 Computerspielsucht

Die Bedeutung des Spiels für den Nutzer kann so weit gehen, dass einige Forscher von Computerspielsucht sprechen. Schon lange wird darüber diskutiert, ob man überhaupt von einer Computerspielsucht sprechen kann, und noch immer gibt es keine aussagekräftigen Zahlen. Doch ein kleiner Anteil der Computerspieler neigt zu solch suchtähnlichem Spielverhalten, je nach Studie sind das 5-10% der Spieler. Das Phänomen tritt dabei unabhängig vom Genre des Spiels auf und es sind sowohl männliche als auch weibliche Nutzer betroffen. Computerspielsucht ist mit anderen, klassischen Suchtproblemen vergleichbar, insbesondere die Auswirkungen in Form von Abhängigkeit und Vernachlässigung aller anderen Aktivitäten ist übergreifend vorzufinden. Problematisch ist, dass Computerspielsucht besonders bei Erwachsenen oft lange verborgen bleibt. Durch die reduzierten sozialen Kontakte fällt niemandem auf, welches Ausmaß der Konsum angenommen hat. Anders als zum Beispiel bei Alkohol- oder Drogenkonsum sind die Anzeichen nicht so offensichtlich und werden von Kollegen oder Mitschülern erst sehr spät wahrgenommen. Folgende Phänomene sind erste Anzeichen einer Computerspielsucht und erfordern eine genauere Untersuchung des Konsums des Spielers:

- Nutzung taglich mehrere Stunden,
- veränderte Schlaf- und Essgewohnheiten,
- Computerspiel Mittelpunkt der Interessen,
- deutliche Einschränkungen im Freizeitverhalten,
- Zeitlicher Kontrollverlust,
- Entzugserscheinungen,
- Vernachlässigung anderer Interessen.

In einer Studie der Berliner Charité-Klinik, vgl. [15], wurde anhand von hirnphysiologischen Untersuchungen bewiesen, dass Computerspielsucht auf vergleichbaren Mechanismen wie Alkohol- oder Cannabis-Abhängigkeit beruht. Dabei wurden die Reaktionen auf visuelle Reize bei exzessiven Spielern untersucht: Legten die Forscher den Spielern Bilder aus einem Computerspiel vor, fielen der Studie zufolge die Hirnreaktionen sehr viel stärker aus als beim Anblick neutraler Reize oder bei Alkoholmotiven. Exzessives Computerspielen aktiviert vermutlich gleiche Strukturen im Hirn wie weiße Drogen, außerdem begünstigt eine Computerspielsucht die Entstehung und Aufrechterhaltung späterer anderer Suchtprobleme, da sich die Verhaltensweisen im Charakter verankert haben und später wieder abgerufen werden können. [15]

Gegenmaßnahmen Lange wurde versucht, Computerspielsucht mit klassischen Suchttherapien (wie zum Beispiel bei Alkoholsucht) zu bekämpfen. Durch die Ähnlichkeit hat dies teilweise auch Erfolg, dabei findet eine Sensibilisierung für das Problemverhalten und eine Reduzierung der Nutzung statt. Damit soll ein kontrollierter Umgang erlernt werden, die Abhängigen sollen ein sinnvolles Maß für den Konsum selbst einhalten können. [15]

Es zeigen sich im Suchtverhalten und damit in der notwendigen Behandlung jedoch Unterschiede. Auf Grund dessen werden derzeit erste Computerspielsucht-Kliniken eröffnet, die sich speziell auf dieses Suchtproblem spezialisiert haben. Zunächst gilt es, den erhöhten Konsum psychotherapeutisch zu erforschen. Für viele Abhängige stellt das Spiel eine Fluchtmöglichkeit von Problemen in ihrem wirklichen Leben dar. Der erste Schritt einer Therapie ist danach daher, den Betroffenen bessere Strategien zur Stressbewältigung aufzuzeigen. [15]

4 Positive Aspekte von Computerspielen

Mögliche positive Wirkungen von Computerspielen sind bisher kaum erforscht. Dennoch gibt es erste glaubwürdige Erkenntnisse, dass der maßvolle Einsatz von Computerspielen die Entwicklung des Spielers fördern kann. Bei diesen positiven Aspekten ist zwischen Effekten bei Kindern und Effekten bei Jugendlichen beziehungsweise jungen Erwachsenen zu unterscheiden, da diese sowohl unterschiedliche Spiele bevorzugen als auch anders auf die Inhalte reagieren.

4.1 Positive Aspekte bei Kindern

Besonders speziell auf die junge Zielgruppe ausgerichtete Spiele mit pädagogischen Inhalten haben positive Effekte auf Kinder. Durch die aktive Teilnahme und den Einfluss auf das Geschehen ist die Motivation zu Lernen größer als beim Umgang mit Büchern, außerdem erzielen diese Programme zum selbstständigen Lernen. Aber auch neben diesen Lernspielen fördern kindgerechte Computerspiele die Fantasie und das Erlernen von Zusammenhängen. [5]

4.2 Positive Aspekte bei Jugendlichen und Erwachsenen

Wie oben beschrieben ist der Stressabbau ein häufiges Nutzungsmotiv für Computerspiele und dies ist auch als positiver Effekt des Spielens anzusehen. Einige Computerspiele beruhigen den Nutzer und bauen Stress und Aggressionen während des Spiels ab. Auch fördern insbesondere die so genannten „Multiplayer“-Spieler mit mehreren menschengesteuerten Charakteren reale Kontakte, da sich die Nutzer zum Spielen verabreden oder über die Spiele reden und ihre Erfahrungen austauschen. Meist sind diese Spiele so konzipiert, dass Einzelpersonen nicht weiterkommen sondern die Spieler sich gegenseitig helfen und unterstützen müssen. Strategiespiele und Spiele die den Nutzer vor Herausforderungen stellen, lassen ihn modellhafte Problemlosungsprozesse erarbeiten, die er auch auf Probleme in der realen Welt übertragen kann. Sie leiten ihn dazu an, strukturiert

vorzugehen und gewonne Erfahrungen wiederbenutzen und weiterentwickeln zu können. Viele Spiele machen es notwendig, sich mit komplexen Regelwerken zu befassen und Spielentscheidungen zu treffen, die die eigene Spielposition verbessern. Der Spieler ist in diese vielschichtige Denk- und Problemlösungsprozesse integriert, wodurch kognitive Prozesse trainiert werden können, wie zum Beispiel auch mehrere Informationsquellen gleichzeitig zu berücksichtigen und angemessen zu bewerten. Jugendliche lernen also mit Computerspielen auf spielerische Weise, sich auf andere Menschen zu verlassen, Arbeit zu teilen und strategisch zu denken. [10]

5 Fazit

Zusammenfassend kann festgestellt werden, dass Computerspiele eine große Bedeutung für die heutige Gesellschaft haben und ein fester Bestandteil im Alltag vieler Kinder, Jugendlicher und Erwachsener sind. Dazu gehören auch gewalttätige Spiele, die einen nicht zu unterschätzenden Anteil ausmachen. Entgegen der häufig geäußerten Meinung machen diese nicht grundsätzlich aggressiv, allerdings sind die genannten Risikogruppen besonders gefährdet, bleibende negative Folgen aus der Nutzung violenter Spiele zu ziehen. Hier ist die Gesellschaft gefragt, eine übermäßige Nutzung und damit auch die negativen Effekte zu verhindern. Weiterhin können Computerspiele aller Genre ähnlich wie andere Medien, insbesondere das Fernsehen, zu gesellschaftlicher Ausgrenzung und Isolation führen, was durch die besonderen Merkmale des Mediums Computerspiel noch begünstigt wird.

Bei geringer Nutzung und altersgerechter Auswahl des Spiels lassen sich auch positive, insbesondere gehirntrainierende Effekte erzielen. Insgesamt steht die Forschung im Bereich „Computerspiele und Gesellschaft“ verglichen mit anderen Medien allerdings noch am Anfang und viele Erkenntnisse müssen noch durch langfristige Studien belegt werden.

Literatur

1. Albert, Mathias; Hurrelmann, Klaus: 14. Shell Jugendstudie. Infratest Sozialforschung, Bielefeld (2002)
2. Ballard, M.E.; Wiest, J.R.: Mortal Kombat - The effects of violent videogame play on males' hostility and cardiovascular responding. Journal of Applied Social Psychology, Ausgabe 26, S. 717 ff. (1996)
3. Children Now: Fair play? Violence, gender and race in video games. Children Now, Oakland (2001)
4. Csikszentmihalyi, Mihaly: Das Flow-Erlebnis - Jenseits von Angst und Langeweile: im Tun aufgehen. Klett-Cotta, Stuttgart (1993)
5. Feierabend, Sabine; Klingler, Walter: Kinder und Medien Studie 2003. Medienpädagogischer Forschungsverbund Südwest, Stuttgart (2003)
6. Fleming, Michele J.; Rickwood, Debra J.: Effects of violent versus nonviolent video games on children's arousal, aggressive mood, and positive mood. In: Journal of Applied Social Psychology 31, S. 2047 ff. Columbia (2001)
7. Frindte, Wolfgang; Obwexer, Irmgard: Ego-Shooter - Effekte der Nutzung von gewalthaltigen Computerspielen und eine Pilotstudie. Zeitschrift für Medienpsychologie, Ausgabe 15, S. 140 ff. Bern (2003)
8. Fromme, Johannes; Meder, Norbert; Vollmer, Nikolaus: Computerspiele in der Kinderkultur. VS Verlag, Opladen (2000)
9. Griffiths, M.D.; Dancaster, I.: The effect of type a personality on physiological arousal while playing computer games. In: Addictive Behaviors, Ausgabe 20, S. 543 ff. (1995)
10. Kraam, Nadia: Problemlösungsprozesse im Computerspiel - Forschungsbericht Forschungsschwerpunkt „Wirkung virtueller Welten“. Fakultät für angewandte Sozialwissenschaften, Fachhochschule Köln (2003)
11. Kunczik, Michael; Zipfel, Astrid: Medien und Gewalt. Befunde der Forschung seit 1998, S. 183 ff. Bundesministerium für Familie, Senioren, Frauen und Jugend, Berlin (2004)
12. Kutteroff, Albrecht; Behrens, Peter: Kinder und Medien Studie 2006. Medienpädagogischer Forschungsverbund Südwest, Stuttgart (2006)
13. Ladas, Manuel: Brutale Spiele(r)? Wirkung und Nutzung von Gewalt in Computerspielen. Peter Lang-Verlag, Frankfurt (2002)
14. Schierbeck, Lisbeth; Carstens, Bo: Violent Elements in Computer Games. An Analysis of Games published in Denmark. In: Feilitzen, Cecilia von; Carlsson, Ulla: Children in the New Media Landscape: Games, Pornography, Perceptions. Children and Media Violence Yearbook 2000, S. 127 ff. UNESCO International Clearinghouse on Children and Violence on the Screen, Göteborg (2000)
15. Wölfling, K.; Grüßer-Sinopoli, S.M.: Exzessives Computerspielen als Suchtverhalten in der Adoleszenz - Ergebnisse verschiedener Studien. Interdisziplinäre Suchtforschungsgruppe, Berlin(2007)

KI in Sportspielen

Armin Büscher

Fachbereich Informatik
Technische Universität Dortmund

Seminar

Computational Intelligence und Computerspiele

1 Einleitung

1.1 Sportspiele

Sportspiele sind Simulationen einer Sportart und ermöglichen dem Spieler, reale Sportereignisse nachzuspielen und dabei die favorisierten Sportler selbst zu steuern. Gerade dieser Aspekt ist auch die größte Herausforderung bei der Entwicklung von Sportspielen, da die Computerspieler die Bewegungen, Mimiken, Gestiken und Taktiken der einzelnen Sportler bzw. der Mannschaften aus Fernsehübertragungen oder aus der Sportarena kennen. Diese Verhaltensweisen nachzubilden erweist sich dabei oftmals als schwieriger, als Fantasiewesen zu erschaffen, die mit ihrem Verhalten den Ideen der Spieldesigner entsprechen. Für diverse Sporttitel wurden bereits Aufnahmen mit der Technik des *Motion Capturing*, bei dem die Bewegung menschlicher Sportler erfasst und auf Computermodelle übertragen wird, mit internationalen Sportstars gemacht, um die dargestellten Figuren auf dem Computer so lebensecht wie möglich wirken zu lassen. Um die Namen und Fotos der Spieler, Logos und Stadien der Vereine nutzen zu dürfen, zahlen die Hersteller für einen Großteil der Sporttitel Lizenzabgaben an die Sportverbände, Vereine und einzelne Sportler. Wie lukrativ diese Verträge inzwischen sind zeigt das Beispiel des Lizenzabkommens zwischen der amerikanischen Football-Liga *NFL* und dem Unternehmen *Electronic Arts*, das bei einer Laufzeit von fünf Jahren ein geschätztes Volumen von 1 Milliarde Dollar aufweist. Im Gegensatz zu Computerspielen, bei denen die Produzenten immer mehr dazu übergehen, eine Story und Helden wie in Hollywoodfilmen zu erschaffen, schränken Sportspiele die Entwickler beim Design der Spielwelt und der Spielregeln ein. Bei jeder simulierten Sportart gibt es Regeln, die für das Spiel übernommen werden müssen, um den Anspruch einer Simulation zu erfüllen. Vereinfachungen im Design des Gameplays in Richtung von Arcade-Sportspielen, wie beispielsweise in Rennspielen, sollen die Möglichkeit geben, Sportarten nachzuspielen, die bei einer realitätsnäheren Simulation für viele Spieler zu schwer und zu zeitaufwändig wären. Vom Spieler wird bei einer Sportsimulation das Verständnis für die sportartspezifischen Handlungen und Taktiken sowie gutes Reaktionsvermögen gefordert.

1.2 KI in Sportspielen

Die künstliche Intelligenz in Sportspielen soll das Verhalten der am Spiel beteiligten Personen nachbilden. Dabei steuert der menschliche Spieler immer genau einen Sportler. Die restlichen am Spiel beteiligten Figuren, auch eventuelle Mitspieler in der Mannschaft des Spielers, müssen von der KI gesteuert werden. In vielen Sportsimulationen wird außerdem die Simulation eines oder mehrerer Schiedsrichter. Ob diese Schiedsrichter immer korrekt entscheiden sollen, ist für das Gameplay eine schwerwiegende Entscheidung. Simuliert die künstliche Intelligenz des Schiedsrichters die Entscheidungsfehler realer Unparteiischer zieht man sich eventuell den Unmut manches Spielers auf sich. Entscheidet der

Schiedsrichter in einem Sportspiel hingegen immer korrekt, so wirkt der so entstehende Spielverlauf nicht realitätsnah. Nahezu alle Mannschaftssportsimulationen und auch einige andere Sportspiele simulieren eine Fernsehübertragung, indem zusätzlich zum Verhalten der am Spiel direkt beteiligten Akteure die KI einen Kommentator mit passenden Kommentaren und eine Soundkulisse mit jubeln oder auspfeifen auf das Spielgeschehen reagieren lässt. In einigen Sportarten muss die künstliche Intelligenz zusätzlich die Rolle des Trainers einer Mannschaft übernehmen und die Taktik und Aufstellung im Hinblick auf die Spielsituation verändern. Hinsichtlich der Anforderungen an die Simulation unterschiedlicher Sportarten gibt es große Unterschiede. Kaum ein Spieler kann die Unterschiede zwischen zwei Rennfahrern erkennen, die auf einer Rennstrecke die Ideallinie befahren. Dahingegen fällt dem Spieler bei Sportarten mit Mannschaften, die gewisse Taktiken verfolgen um ein Spielziel zu erreichen, schnell auf, wenn die Sportler sich nicht wie in der Realität verhalten. Reale Sportmannschaften verbringen unzählige Trainingseinheiten damit, die individuellen Fähigkeiten der einzelnen Athleten zu verbessern und aus einer Gruppe von Individuen eine funktionierende Einheit zu bilden. Dabei wird so trainiert, dass die einzelnen Sportler auf die aktuelle Spielsituation angemessen und in einer für die Mannschaftskollegen vorhersehbaren Weise instinktiv reagieren. In Mannschaftsballsportarten ist es das Ziel, den gegnerischen Spielern auszuweichen, sie zu umspielen und sie gerade nicht an den Ball kommen zu lassen.

2 KI in Rennspielen

Bei Rennspielen versuchen mehrere Teilnehmer eine bestimmte Strecke in möglichst kurzer Zeit zu absolvieren und die bestmögliche Platzierung zu erreichen. Dabei beschränkt sich die Art des Fortbewegungsmittels bei aktuellen Titeln nicht auf Autos und Motorräder, sondern umfasst auch Boote, Skateboards und viele weitere Möglichkeiten. Zu unterscheiden sind bei den Rennspielen die Simulationen, in denen es hauptsächlich um das Renngeschehen korrekt und realistisch wiederzugeben, und die Arcade-Rennspiele, in denen der Realismus eine untergeordnete Rolle spielt und der schnelle Einstieg und das einfache Spiel im Vordergrund stehen. Bei Arcade-Rennspielen kann es zusätzlich zur besten Rundenzzeit noch andere Randbedingungen wie Bonuspunkte und Waffen geben, die auch Auswirkungen auf die Entwicklung der künstlichen Intelligenz haben müssen, auf die hier aber nicht weiter eingegangen wird. Für reale wie virtuelle Rennfahrer ist der Schlüssel zur schnellen Umrundung eines Rennkurses die Ideallinie, eine gedachte Linie, die die schnellste Bahn durch die Kurven der Strecke repräsentiert. Abbildung 1 zeigt ein Beispiel für eine Ideallinie durch eine Kurve. Dabei muss das Fahrzeug bis zum Ende des ersten Pfeils auf die Geschwindigkeit heruntergebremst sein, die dem Fahrzeug gerade noch ermöglicht, in die Kurve zu fahren, ohne durch die Fliehkraft herausgedrückt zu werden. Diese Geschwindigkeit muss der Fahrer bis zum Ende des zweiten Pfeils am Scheitelpunkt der Kurve halten und kann dann wieder beschleunigen. Die Ideallinie ergibt sich durch ein komplexes Zusammenspiel der physischen Eigenschaften der

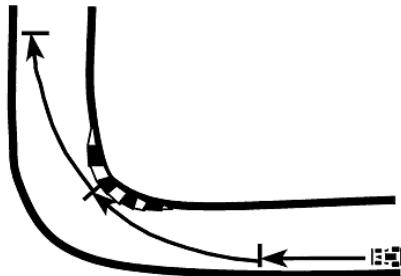


Abbildung 1. Ideallinie einer Rennstrecke (Quelle: www.club100.co.uk)

verwendeten Fahrzeuge, der Form und des Bodenbelags der Rennstrecke. Für den Spieler einer Rennsimulation stellt die Ideallinie die größte Herausforderung dar. In [1] stellt John Manslow einen Algorithmus zur schnellen und effizienten Approximation der Ideallinie einer Rennstrecke vor. Die Aufgabe der künstlichen Intelligenz in einem Rennspiel geht natürlich weit über die Aufgabe hinaus, das Fahrzeug auf der Ideallinie zu halten. Im Fall das langsamere Fahrzeuge überholt werden können wechselt die KI auf eine der so genannten Überhollinien. Je mehr Fahrzeuge gleichzeitig auf der sind, desto mehr Möglichkeiten müssen der KI gegeben werden, eine alternative Fahrspur zu wählen. Bei der Analyse der Überholmöglichkeiten muss abgeschätzt werden, wo sich das eigene und die Fahrzeuge der Kontrahenten in naher Zukunft befinden. Für diese Aufgabe werden in Rennspielen häufig so genannte *Dead-Reckoning*-Algorithmen verwendet, deren Funktionsweise ausführlicher in 4.1 beschrieben wird. Zur Steuerung des Fahrzeugs muss die Beschleunigung bzw. Verzögerung und die Möglichkeit zur Richtungsänderung berücksichtigt werden. Da die künstliche Intelligenz bei ihren Reaktionen reale Rennfahrer imitieren soll, muss sie weiche Übergang bei der Steuerung des Gas- und Bremspedals sowie der Lenkung simulieren. Der Übergang zwischen von Vollgas auf Bremse soll als Beispiel nicht innerhalb eines Frames passieren. Als Lösung wurden in Rennspielen so genannte *Feedback Loops* implementiert, die aus der Steuerungs- und Regeltechnik adaptiert wurden. *Feedback Loops* bieten die Möglichkeit, einen Ist-Wert über eine bestimmte Zeit einem Soll-Wert anzupassen. Des Weiteren werden in Rennspielen endliche Zustandsautomaten verwendet, die die Reaktionen der virtuellen Fahrer beispielsweise während der Situationen des Fahrens auf der Ideallinie, des Überholens oder auch eines Pit Stops regeln. Ein wichtiger Aspekt, der bei der Entwicklung einer KI für ein Rennspiel beachtet werden sollte ist die Etiquette unter den Rennfahrern. Je nach Thema des Spiels, wie beispielsweise Simulation in der *Formel 1* oder *NASCAR*, sollen die von der KI gesteuerten Fahrer Kollisionen eher ausweichen oder sie eventuell gerade forcieren. Prinzipiell sollte in Kurven das innere Fahrzeug bereits dann den Vorzug erhalten, wenn sich sein Zentrum vor dem vorderen Ende des äußeren Fahrzeugs befindet. Um den Spieler nicht zu verärgern, wird ihm in vielen Rennspielen durch die künstliche Intelligenz der Vorzug gewährt.

3 Aufbau einer Sportspiel-KI-Architektur

Am Beispiel der Basketball-Simulationsserie *NBA Inside Drive*, deren KI-Architektur von Terry Wellmann in [2] beschrieben wird, sollen in diesem Abschnitt die Konzepte und der Aufbau einer KI-Architektur für kommerzielle Sportsimulationstitel vorgestellt werden. Die erste Aufgabe des Designer einer KI, die Wellmann anspricht, ist die Identifizierung der von der KI gesteuerten Spieleragenten auf höchster Ebene zu treffenden Entscheidungen, die er als Pläne bezeichnet. Dabei soll zuerst ein grober Überblick über die zu treffenden Entscheidungen entstehen, wobei zusammengehörige Entscheidungen gruppiert werden. Beispiele für die Pläne beim Basketball sind die Offensive, Defensive, Pass, Wurf, Positionierung des Verteidigers, einen Pass abfangen oder einen Wurf blocken. Die Details der einzelnen Pläne wird im Folgenden nicht weiter betrachtet. Die in *NBA Inside Drive* verwendeten Pläne leiten sich von programmiertechnischer Sicht von der Klasse *AgentPlan* ab, die im folgenden Codebeispiel skizziert wird:

```
class AgentPlan
{
    ...
    float EvaluateInitiation();
    float EvaluateContinuation();
    void Initiate();
    void Update();
    ...
}
```

Die Funktion *EvaluateInitiation()* wird benutzt, um zu berechnen, wie sinnvoll es ist, den Plan auszuführen. Die *EvaluateContinuation()*-Funktion wird hingegen verwendet, um zu entscheiden, ob es sinnvoll ist, den Plan weiter zu verfolgen, falls er gerade aktiv ist. Eine Unterscheidung zwischen der Entscheidung, einen Plan überhaupt auszuführen oder ihn weiter zu verfolgen ist von großer Bedeutung, da in beiden Fällen vom Designer oft unterschiedliche Voraussetzungen zu Grunde gelegt werden. Die *Initiate()*-Funktion wird bei jeder Ausführung des Plans zu Anfang ausgeführt, um einmalige Entscheidungen zu treffen. Die Funktion *Update()* ist für die eigentliche Durchführung des Plans zuständig und wird bei jedem Berechnungsdurchlauf von der KI ausgeführt, falls der Plan aktiv ist. Die beiden Funktionen für die Auswertung geben jeweils Fließkommawerte an die aufrufende Funktion zurück. Somit ist es möglich, mehrere Pläne miteinander zu vergleichen und zu entscheiden, welcher ausgeführt werden soll. Wellmann schlägt für diese Ergebniswerte eine Spanne zwischen -1.0 und 1.0 vor, wobei nur Pläne mit einem Ergebniswert von größer als 0.0 überhaupt in Betracht gezogen werden. Nachdem für die einzelnen Agenten Pläne identifiziert sind, wird ein Gerüst benötigt, das steuert, in welchen Situationen welche Pläne benutzt werden sollen.

3.1 Endliche Zustandsautomaten

Wie in vielen anderen Sportsimulationen werden zu dieser Aufgabe des Teammanagements in *NBA Inside Drive* endliche Zustandsautomaten für diese Aufgabe verwendet. Die Vorteile von endlichen Zustandsautomaten Die Menge der Zustände wird bei *NBA Inside Drive* in offensive und defensive Zustände geteilt und jeder Zustand erfüllt einen anderen Zweck im übergeordneten Ziel des Zusammenspiels der Mannschaft. Dabei gibt es in Wellmanns Architektur für jeden offensiven Zustand, wie beispielsweise den Zustand bei Freiwürfen, einen entsprechenden defensiven Zustand. Die offensiven und defensiven Zustände sind:

- **Inbound:** Zuständig den Ball beim Einwurf ins Spiel zu bringen
- **Transition:** Zuständig den Ball in die gegnerische Hälfte zu bringen
- **Frontcourt:** Zuständig um das Spiel in der gegnerischen Hälfte zu machen, um in Wurfposition zu kommen
- **Rebound:** Zuständig den Versuch des „Rebound“ zu steuern, nachdem der Ball geworfen wurde
- **Recover loose ball:** Zuständig einen Ball zu erobern, der durch keinen Spieler kontrolliert wird (z.B. nach einem Rebound)
- **Free throw:** Zuständig für die Freiwurfversuche

Zusätzlich zu den offensiven und defensiven Zuständen gibt es noch allgemeine Zustände, die in neutralen Situationen, wenn der Ball nicht im Spiel oder kein Team in offensive oder defensive ist, zuständig sind:

- **Time-Out:** Zuständig während Time-Outs
- **Quarter Break:** Zuständig während der Pausen zwischen den Spielvierteln
- **Substitution:** Zuständig für Spielerauswechselungen
- **Post game:** Zuständig für Jubelszenen nach dem Spiel

Die Zustandsübergänge werden durch Spielereignisse wie beispielsweise einen Wurfversuch, ein Foul, ein Time-Out oder dem Ende eines Viertels ausgelöst. Nachdem die Pläne identifiziert und in Zuständen organisiert sind, verbleibt die Frage, wie die KI eines Agenten nun entscheidet, ob ein Plan zu einem gewissen Zeitpunkt sinnvoll ist oder nicht. Als Beispiel sei hier der Wurfplan genannt, der für einen Agenten in Ballbesitz auswerten soll, ob dieser einen Wurfversuch von der aktuellen Position wagen soll. Es muss also berechnet werden, wie hoch die potentielle Erfolgsrate bei einem Wurf von dieser Position ist. Dies hängt in Sportsimulationen wie im realen Sport von den Fähigkeiten und der aktuellen Fitness des Sportlers und seiner direkten Gegenspieler ab. Des weiteren könnte man sich auch vorstellen, dass psychischer Druck auf den Schützen einwirkt, der eine positive oder negative Auswirkung haben kann. In Sportspielen werden die Fähigkeiten der Sportler häufig durch Bewertungen in Form von Zahlenwerten repräsentiert. Mit Hilfe dieser Werte kann nun die benötigte Erfolgsrate errechnet werden. Damit die virtuellen Sportler sich nicht immer gleich verhalten und wie von Wellmann bezeichnet „zum Leben erweckt“ werden, sind die Entscheidungen, die die künstliche Intelligenz in *NBA Inside Drive* trifft, bis zu einem gewissen Grad von Pseudozufallszahlen abhängig. Hierdurch erscheint das Verhalten der virtuellen Basketballer für den Spieler komplexer als es in Wahrheit ist.

4 Verwendete Techniken

4.1 Dead Reckoning

In Sportspielen ist es für ein erfolgreiches Vorgehen wichtig, die Bewegungen der am Spiel beteiligten Sportler und Objekte wie Rennfahrzeuge oder Bälle vorauszusagen. Dabei sind die Bewegungen von Mitspielern und Gegenspielern nicht zu unterscheiden, da die künstliche Intelligenz eines virtuellen Sportlers nicht den Vorteil haben sollte, durch versteckte Kommunikation die Bewegungen der anderen vom Computer gesteuerten Sportler vorherzusehen. Bei der Vorhersage der Bewegung helfen so genannte *Dead Reckoning (DR)*-Algorithmen, die ursprünglich als Hilfsmittel für Navigatoren auf Schiffen verwendet wurden. So kann ein Seemann die aktuelle Position seines Schiffs abschätzen, indem er die letzte bekannte Position, den beabsichtigten Kurs und die aufgezeichnete Geschwindigkeit in die Rechnung miteinbezieht, ohne allerdings Strömungen oder Winde zu betrachten. Je größer die Auswirkungen der unbekannten Außenfaktoren ist, um so schlechter wird die errechnete Vorhersage. Die Abschätzung der Bewegung kann nun natürlich auch für jedes Objekt durchgeführt werden, dessen aktuelle Position, Geschwindigkeit und Bewegungsrichtung bekannt sind. Ist die Geschwindigkeit eines Objekts nicht bekannt, so kann sie durch Beobachtung berechnet werden, indem die Position des Objekts zu zwei Zeitpunkten überwacht wird. Als Beispiel für den Nutzen der Anwendung von *Dead Reckoning* bei der künstlichen Intelligenz in Sportspielen kann man ein Mannschaftsspiel geben, bei dem ein vom Computer gesteuerter Spieler den Ball zu einem Mitspieler passen soll und vor dem Abspiel überprüft, ob dieser Pass abgefangen werden kann. Im Gegensatz zu einem Rennspiel, in dem es für die KI ausreicht die Bewegungen der Fahrzeuge in zwei Dimensionen vorauszusagen, muss bei Sportarten mit Bällen die dritte Dimension in die Berechnung miteinbezogen werden.

In der einfachsten Form wird die Berechnung auf *Newton's first law of motion* reduziert. Dieses besagt, dass man mit dem Wissen der Position eines Objekts und seiner Geschwindigkeit annehmen kann, dass es sich in der Zukunft in einer geraden Linie in derselben Geschwindigkeit weiterbewegt. Mit dieser einfachen Methode lassen sich die zukünftigen Positionen mit der Gleichung

$$P_t = P_0 + v * t$$

berechnen, wobei P_t die Position des Objekts zum Zeitpunkt t , P_0 die ursprüngliche Position und v die Geschwindigkeit bezeichnen. Für Objekte, deren Bewegungen nicht oder nur wenig von äußeren Einflüssen abhängen, ist diese Berechnungsmethode sicherlich ausreichend. Für Sportspiele wird andererseits eine Methode benötigt, die auch beispielsweise die Erdanziehungskraft oder den Luft- und Rollwiderstand berücksichtigt um mit diesen Faktoren die Beschleunigung eines Objektes abschätzen zu können. Ist nun zusätzlich zur ursprünglichen Position P_0 und der Geschwindigkeit v eines Objekts der Beschleunigungsvektor a bekannt, so kann man mit der kinematischen Gleichung

$$P_t = P_0 + v_0 * t + 0,5 * a * t^2$$

eine bessere Annäherung für den Positionsvektor P_t zu einem gewählten Zeitpunkt bestimmen.

4.2 Gameplay-Analyse

Gameplay ist ein beliebter Ausdruck, der für die Gesamtheit der Spielerfahrungen während der Interaktion mit einem Spiel steht. Bei der Gameplay-Analyse geht es darum, das Verhalten des Spiels mit allen Regeln, Schwierigkeitsgraden, der Fairness, den Zielen und Nebenbedingungen zu analysieren. Die Präsentation des Spiels, also die Grafik und der Sound spielen hierbei eine untergeordnete Rolle. Das Gameplay hat einen großen Einfluss auf den Unterhaltungswert eines Spiels und die Hersteller aktueller Sportspiele treiben großen Aufwand, um das Gameplay zu testen und zu verfeinern. Einen unkonventionellen Ansatz erarbeitete die *GAMES-Group* der *University of Alberta* in Zusammenarbeit mit *Electronic Arts* mit den Tools *SAGA-ML* und *SoccerViz* [3]. Diese Werkzeuge sollen die Entwickler unterstützen, indem sie den Prozess der Bewertung des Gameplays teilweise automatisieren. Den Unterhaltungswert eines Sportspiels zu messen kann natürlich nicht von einem Computer übernommen werden, allerdings kann durch systematisches Testen des Gameplays Tests zu Spielsituatien durchgeführt werden, die für menschliche Tester unintuitiv und ermüdend wären. Das Ziel des Analysetools *SAGA-ML*, das im Hinblick auf die von *Electronic Arts* entwickelte *FIFA Soccer*-Reihe angepasst wurde, ist es, so genannte „sweet spots“ zu finden. Das Werkzeug *Soccerviz* besitzt die Aufgabe, die Ergebnisse, die *SAGA-ML* liefert, aufzubereiten und anzuzeigen. Diese „sweet spots“ sind Positionen, an denen der Spieler mit den gleichen Manövern mit Leichtigkeit immer wieder Tore erzielen kann. Zusätzlich können so genannte „hard spots“ gefunden werden, an denen es zu schwer ist, ein Tor zu erzielen. Die von Finnegan Southey et al. in [3] vorgestellten Szenarios umfassen die Analyse des Eckballs und eines Schusses eines Stürmers auf das Tor bei der Fußballsimulation *FIFA 99*. Die bei der Analyse des Eckballszenarios verwendete Metrik ist die Wahrscheinlichkeit ein Tor zu erzielen, nachdem der Ball von einem Spieler berührt wurde. Durch maschinelles Lernen werden die Daten aus der Simulation der Szenarios aufbereitet. Dabei werden so genannte „rule learners“ für die Lernkomponente benutzt, die den Ansatz verfolgen, für die vorgegebenen Eingaben eine Menge von Regeln aufzustellen, die den Ausgabewert, hier also die Wahrscheinlichkeit eines Treffers, vorhersagen. Als Beispiel könnte eine Regel im Stürmer-Torwart-Szenario etwa so aussehen:

WENN der Stürmer höchstens 5m vom Torwart entfernt ist
UND der Winkel zwischen Torwart und Stürmer zwischen 30° und 40° liegt
UND der Torwart höchstens 1m von der Mitte des Tors entfernt ist
DANN ist die Wahrscheinlichkeit eines Tors höher als 70%.

Die Ausgabe der Regeln erfolgt durch das Werkzeug *SoccerViz* in einer zweidimensionalen Draufsicht des Fußballfeldes. In Abbildung 2 (entnommen aus [3]) ist das Ergebnis des Eckballszenarios aufgezeigt. Die schattierten Regionen zei-

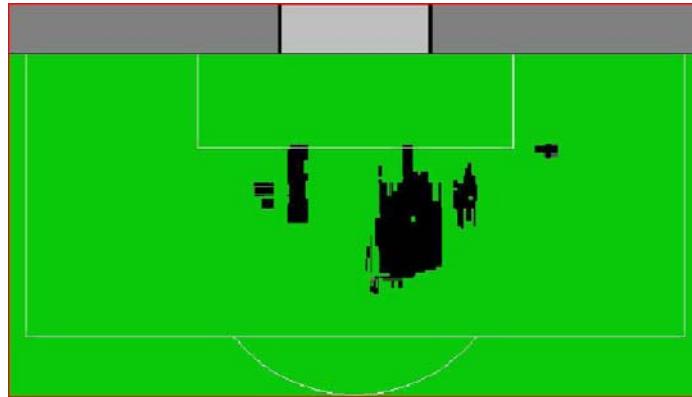


Abbildung 2. „sweet spots“ im Eckballszenario (aus [3])

gen die Positionen, für die *SAGA-ML* bei einem Eckball an diesen Punkt in der Simulation Regeln erlernt hat, die eine Torchance von mehr als 40% aufweisen. Der Eckball wurde dabei immer von der oberen rechten Ecke des Feldes ausgeführt. Für die Entwickler helfen diese Analysen, die Schwächen der künstlichen Intelligenz automatisiert zu erkennen. Während der Entwicklung neuer *FIFA Soccer*-Titel setzt *Electronic Arts* die Werkzeuge *SAGA-ML* und *SoccerViz* in angepassten Versionen ein, um das Gameplay und die KI der Fußballsimulation zu verbessern.

5 Lernfähige KI in Sportspielen

Das Ziel der künstlichen Intelligenz in einem Sportspiel ist im Gegensatz zu Wettbewerben wie der Fußball-Simulationsliga im *Robocup* nicht, zu gewinnen, sondern den Spieler zu unterhalten. Nach Rijswijck [4] sollte das Ziel einer adaptiven KI nach einer Niederlage nicht sein, nie wieder zu verlieren, sondern nicht wieder das gleiche Verhalten zu zeigen. Der menschliche Spieler wird also gefordert, indem er nicht immer durch gleiche Spielzüge und Taktiken gewinnen kann, da die KI sich ähnlich einem menschlichen Mitspieler auf diese Vorgehensweisen einstellt. Eine weitere Anwendung für adaptive KI in Sportspielen ist die Zusammenarbeit mit dem Spieler, da die KI wie schon angesprochen in vielen Spielen nicht nur die Gegenspieler sondern auch die vom Spieler gerade nicht gesteuerten Mitspieler übernimmt. Da in Sportspielen die Charakter im Gegensatz zu anderen Spielegenres wie *First Person Shootern* nicht dazu neigen, während einer Partie zu sterben, hat die KI ausreichend Zeit, dem Spieler

zu zeigen, dass sie adaptiv auf sein Verhalten und seine Taktik eingeht. Für eine adaptive künstliche Intelligenz im Rahmen von Sportspielen ergibt sich eine weitere Einschränkung gegenüber der künstlichen Intelligenz im Umfeld wissenschaftlicher Forschungsarbeit. So sollen sich die virtuellen Sportler auch nach der Analyse von Spielsituationen immer noch ähnlich ihren realen Vorbildern verhalten. In einem von Rijswijck durchgeführten Versuch mit mehreren künstlichen Intelligenzen, die in einem Turniermodus in der Fußballsimulation *FIFA 99* gegeneinander antraten, gewann die KI, in die eine Regel implementiert war, nach der ein Spieler, sobald er die Kontrolle über den Ball erlangt, sofort auf das gegnerische Tor schießen sollte. Es liegt auf der Hand, dass es den meisten Spielern von Fußballsimulationen wenig Spaß bereiten würde, gegen eine vom Rechner so gesteuerte Mannschaft anzutreten.

5.1 Verhaltensmodelle in Kombination mit adaptiver KI

Wie schon erwähnt werden *Finite State Machines* in vielen Spielen für die Modellierung der Verhaltensweise virtueller Sportler genutzt. Ein Merkmal der endlichen Zustandsautomaten ist die Regel, dass sie sich zu jedem Zeitpunkt in genau einem Zustand befinden. Um einen großen Nachteil der Benutzung endlicher Zustandsautomaten aufzuzeigen, zeigt Abbildung 3 aus [4] die unterschiedlichen Einflüsse, die die Entscheidung des Spielcharakters *Pacman* beeinflussen können. Wird *Pacman* von einer KI mit Hilfe eines endlichen Zustandsautomaten gesteu-

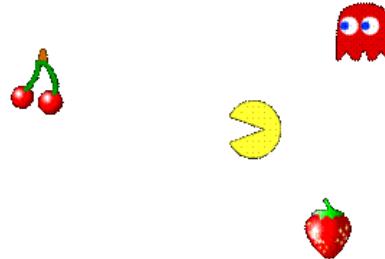


Abbildung 3. Unterschiedliche Einflüsse auf den Spielcharakter Pacman

ert, so wäre es nahe liegend Zustände wie „Ängstlich → Flucht vor dem Gegner“ und „Hungrig → Suche nach Futter“ in diesen zu integrieren. Entscheidet sich der *Pacman* zu fliehen, so wird er dies in der Gegenrichtung zu seinem Verfolger tun und sich somit auch vom Futter weg bewegen. Ist der *Pacman* im Zustand der Futtersuche so wird ihn dieser Zustand anweisen, die am wenigsten entfernte Futterquelle aufzusuchen, wobei er sich allerdings gefährlich nah an seinen Verfolger begibt. Es wird in beiden Zuständen also nur ein grundlegendes Bedürfnis der Spielfigur betrachtet und die anderen Bedürfnisse werden ausgeblendet. In

einem so genannten Verhaltensmodell sind diese Bedürfnisse alle gleichzeitig aktiv, wobei sie sich im Einfluss auf das endgültige Verhalten unterscheiden. Bei der Schema-basierten Koordination, die auch von Teilnehmern des *Robo-Cup* erfolgreich eingesetzt wird, wird von jedem Bedürfnis ein Kraftfeld mit einer bestimmten Stärke erzeugt, das die Spielfigur in eine bestimmte Richtung zieht. Die tatsächliche Bewegung der Figur ist ergibt sich aus der Summe der Einflüsse der Kraftfelder. In der in Abbildung 3 gezeigten Szene gäbe es von den Futterquellen ausgehende anziehende Kraftfelder und von der Spielfigur des Geists ein abstossendes Kraftfeld. Mit Hilfe der Schema-basierten Koordination können also die beiden Bedürfnisse nach Flucht und Hunger gleichermaßen befriedigt werden. Dieses Modell kann bei Sportspielen die in Abschnitt 3 vorgestellten endlichen Zustandsautomaten ersetzen und einen virtuellen Sportler von der KI so steuern, dass ein dynamischeres und flüssigeres Verhalten ohne ständige „Entweder-Oder“-Entscheidungen entsteht. Ein weiterer Vorteil von Verhaltensmodellen gegenüber endlichen Zustandsautomaten ist die bessere Erweiterbarkeit um neue Bedürfnisse. Während beim Hinzufügen eines neuen Zustands zu einer FSM alle Zustandsübergänge von und zum neuen Zustand hinzugefügt werden müssen hat dies beim Hinzufügen eines neuen Bedürfnisses im Verhaltensmodell keinen Einfluss auf die bestehenden Bedürfnisse.

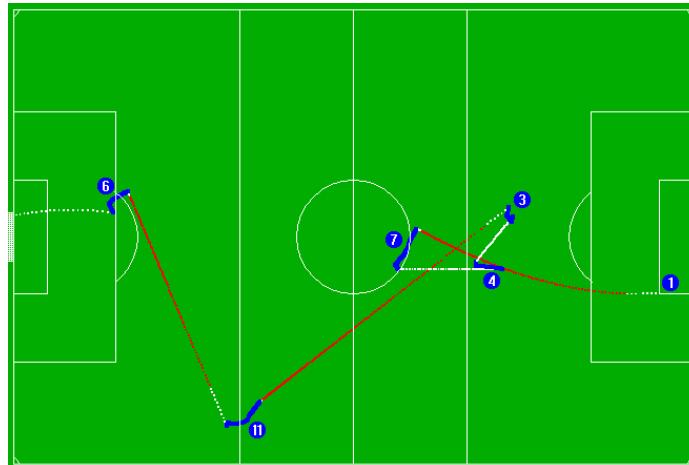


Abbildung 4. Beispiel einer Spielsequenz aus *FIFA 2002* (aus [4])

Die Grundidee hinter der adaptiven KI, die Rijswijck für *FIFA 2002* entwickelt hat, ist die Kombination des vorgestellten Verhaltensmodells der Kraftfelder mit einer adaptiven Strategie, die in der Defensive Gegentore verhindern soll. Sobald der Spieler gegen diese adaptive KI ein Tor erzielt, wird der Spielzug analysiert, der zum Tor führte. Dabei kann die KI auf die Aufzeichnungen

der Bewegungen des Balls und der eigenen und gegnerischen Spieler zurückgreifen. Die Betrachtung der Spielszene erfolgt ab der Balleroberung der ein Tor erzielenden Mannschaft. Sollte allerdings ein Eckball oder Freistoß ausgeführt worden sein, so analysiert der Algorithmus den Spielzug ab dem Moment der Ausführung. Ein Beispiel der vereinfachten Simulation von Spielszenen der Fußballsimulation *FIFA 2002* aus [4] wird in Abbildung 4 gezeigt. Der verwendete Algorithmus sucht die Stationen des Spielaufbaus, in denen ein Verteidiger den Spielaufbau entscheidend hätte stören können. Für die verteidigende Mannschaft ist es nicht wichtig, dass ein bestimmter Verteidiger in einer bestimmten Position steht um das Gegentor zu verhindern. In Verbindung mit dem Verhaltensmodell mit Kraftfeldern können die durch die Lernbeispiele gewonnenen Erkenntnisse als Bedürfnisse mit Hilfe von Kraftfeldern, die die Spieler der Mannschaft in gewissen Situationen in Richtung der markanten Verteidigungspositionen ähnlich magnetischen Kraftfeldern lenken, modelliert werden. Die Kraftfelder sind dabei nur aktiv, wenn sich eine Spielsituation ergibt, die ähnlich zu der vorher gelernten Situation ist. In Abbildung 5 ist das aus der Anwendung des von Rijswijck benutzten Algorithmus auf die in Abbildung 4 gezeigten Spielsituation entstehende Kraftfeld gezeigt.

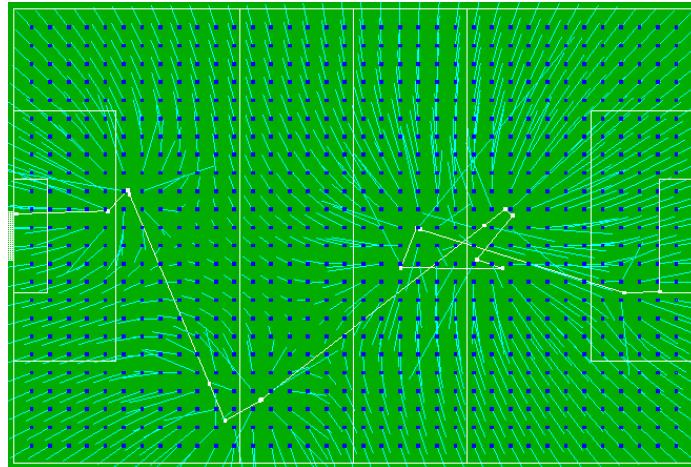


Abbildung 5. Resultierendes Kraftfeld aus dem Lernbeispiel (aus [4])

5.2 Neuronale Netze zur Rennwagensteuerung

Einen weiteren Ansatz der Verwendung von lernfähiger künstlicher Intelligenz in Sportspielen ist die Verwendung von neuronalen Netzen, die auf spezielle Aufgaben in diesem Kontext trainiert werden. Da hier nur auf die spezifischen Anpassungsaufgaben von neuronalen Netzen im Kontext von Rennspielen eingegangen wird, sei hier für Grundlagen der Technik der neuronalen Netze auf

[5] verwiesen. Die KI in der von *Codemasters* herausgegebenen Rennsimulation *Colin McRae's Rallye 2* wurde vom für die künstliche Intelligenz verantwortlichen Entwickler Jeff Hannan [6] mit einem neuronalen Netz so trainiert, dass sie auch auf unbekannten Strecken über den Kurs fahren konnte. Die Schwierigkeit bei der Rallyesimulation *Colin McRae* liegt in der Steuerung des Fahrzeugs auf verschiedenen Bodenbelägen wie Asphalt, Kies und Matsch. Auf rutschigem Untergrund muss der Wagen schon vor der Kurve eingelenkt werden, um wie für Rallyfahrzeuge charakteristisch durch die Kurve zu gleiten. Nachdem Hannan bei anfänglichen Versuchen mit Regelsätzen für die KI der vom Computer gesteuerten Fahrer keine Fortschritte erzielte, entschied er sich für Versuche mit mehrlagigen Perzeptronen. Die größte Schwierigkeit lag laut Hannan in der Identifikation der besten Eingaben und der Größe für das neuronale Netz. Die Menge der verwendeten Eingabedaten wählte er durch Experimente als Mittelweg zwischen Performanz und resultierendem Fahrverhalten der KI. Dabei galt es, sich zwischen Eingabedaten, die wie beispielsweise die Fahrbahninformationen im Abstand von 5m, 10m und 20m vor dem Fahrzeug redundante Informationen beinhalten können, zu entscheiden. Die Trainingsdaten erzeugte Hannan, indem er die von ihm gedrückten Tasten und Streckeninformationen aufzeichnete während er *Colin McRae's Rallye 2* spielte. Seiner Aussage zufolge waren wenige tausend Trainingsrunden ausreichend, um die verschiedenen Fahrmodelle zu trainieren, bei denen zwischen den unterschiedlichen Bodenbelägen unterschieden wurde. Das neuronale Netz ist bei dieser Rennsimulation allerdings nur für das schnellstmögliche Umrunden der Strecken zuständig, wenn keine Fahrzeuge überholt werden müssen. Im Fall des Überholens schaltet die KI auf eine regelbasierte Methode um, solange ein Kontrahent in der Nähe des Fahrzeugs ist. Die Ausgabewerte des für die künstliche Fahrerintelligenz verwendeten neuronalen Netzes ergeben simple Ein-/Aus-Befehle für den Controller. Da aus den Trainingsdaten eine Ideallinie errechnet wird benötigt die KI für das Befahren unbekannter Strecken als Eingabe die Ideallinie, kann dann aber ohne neue Trainingsdaten diese Strecken befahren. Da es sich bei *Colin McRae's Rallye 2* um ein kommerzielles Produkt handelt, gab Hannan in den Veröffentlichungen keine genauen Informationen zu den Parametern des verwendeten neuronalen Netzes preis.

Literatur

1. Manslow, J.: Fast and efficient approximation of racing lines. In Rabin, S., ed.: AI Game Programming Wisdom II 2004. 485–488
2. Laramee, F.D.: Dead reckoning in sports and strategy games. In Rabin, S., ed.: AI Game Programming Wisdom II 2004. 499–504
3. et al., F.S.: Semi-automated gameplay analysis by machine learning. In: Proceedings of the 2005 Conference on Artificial Intelligence in Interactive Digital Entertainment (AIIDE-05). 123–128
4. van Rijswijck, J.: Learning goals in sports games. In: Proceedings of the 2003 Game Developers Conference
5. Kriesel, D.: Neuronale netze

6. Buckland, M.: Interview mit jeff hannan zur verwendung von neuronalen netzen in colin mcrae's rally 2