TECHNICAL UNIVERSITY OF DORTMUND

REIHE COMPUTATIONAL INTELLIGENCE

COLLABORATIVE RESEARCH CENTER 531

Design and Management of Complex Technical Processes and Systems by means of Computational Intelligence Methods

PG511 - CI in Games - Zwischenbericht

Holger Danielsiek, Christian Eichhorn, Tobias Hein, Edina Kurtić, Georg Neugebauer, Nico Piatkowski, Michael Puchowezki, Jan Quadflieg, Sebastian Schnelker, Raphael Stüer, Andreas Thom, Simon Wessing

No. CI-236/07

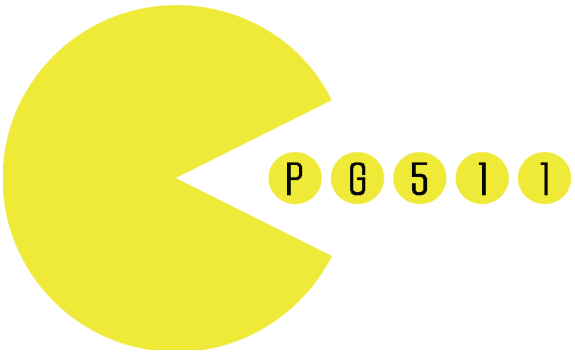
Technical Report ISSN 1433-3325 December 2007

Secretary of the SFB 531 · Technical University of Dortmund · Dept. of Computer Science/LS 2 · 44221 Dortmund · Germany

This work is a product of the Collaborative Research Center 531, "Computational Intelligence," at the Technical University of Dortmund and was printed with financial support of the Deutsche Forschungsgemeinschaft.

UNIVERSITÄT DORTMUND
FACHBEREICH INFORMATIK
Projektgruppe 511: CI In Games

CI in games



Zwischenbericht

Holger Danielsiek, Christian Eichhorn,
Tobias Hein, Edina Kurtić,
Georg Neugebauer, Nico Piatkowski,
Michael Puchowezki, Jan Quadflieg,
Sebastian Schnelker, Raphael Stüer,
Andreas Thom, Simon Wessing
27. Oktober 2007

INTERNE BERICHTE

Betreuer:
Nicola Beume
Boris Naujoks
Mike Preuß

Inhaltsverzeichnis

1 Einleitung

Die Projektgruppe 511 (PG511) beschäftigt sich mit dem Einsatz von Methoden der Computational Intelligence (CI), insbesondere evolutionären Algorithmen (EA), künstlichen neuronalen Netzen (KNN) und Fuzzy-Systemen, in Computerspielen. Der Einsatz dieser Methoden soll es den vom Computer gesteuerten Gegnern ermöglichen, ihr Verhalten nicht aus einem vorher durch die Spieldesigner vorgegebenen Determinismus, sondern durch Lernverfahren zu entwickeln. Die PG511 verspricht sich davon, herkömmliche Methoden der künstlichen Intelligenz (KI) in für den Spielspaß relevanten Punkten wie Ausgeglichenheit des Schwierigkeitsgrades, empfundene „Natürlichkeit“ des Verhaltens, aber auch in Fairness zu übertreffen. Die Aufgabe der PG511 ist es zu untersuchen, ob und inwieweit sich die genannten Methoden auf Computerspiele anwenden lassen, um dieses Ziel zu erreichen. Diese vage Zielsetzung wird durch zwei Teilaufgaben konkretisiert:

Das erste Ziel ist, die CI-Methoden an die Aufgabe anzupassen, sie zu implementieren und gegeneinander zu testen. Computergegner, die auf einer der genannten Methoden basieren, werden erwartungsgemäß unterschiedliches Verhalten aufweisen. In diesem Projekt soll untersucht werden, ob der menschliche Spieler diese Unterschiede wahrnimmt und gegebenenfalls verschieden bewertet; genauer, ob gewisse Methoden einen höheren Spielspaß erzeugen als andere.

Die Frage nach dem unterschiedlichen Spielspaß führt zur zweiten Teilaufgabe: Wieviel Spaß ein Spiel einem Spieler macht, scheint in erster Linie ein rein subjektives Empfinden zu sein, welches sich der objektiven Messung und dem Vergleich entzieht. Es ist aber, unter anderem den Humanwissenschaften, immer wieder gelungen, für scheinbar subjektive Phänomene Gesetzmäßigkeiten zu finden. Es bietet sich also an, ein Maß für den Spielspaß zu suchen, das zu vergleichen und vielleicht sogar ohne das Dazutun von menschlichen Testspielern zu gewinnen ist. Zu diesen Zwecken greifen wir auf das von Yannakakis und Hallam [YH04] aufgestellte Maß für das Spiel Pac-Man (Kapitel 3) zurück, welches Spielspaß als Funktion fasst, bestehend aus Schwierigkeit, Unvorhersehbarkeit der Geisterbewegungen und einer der Gleichverteilung angenäherten Verteilung der Geister über dem Spielfeld. Ob dieses Maß tatsächlich den Spielspaß ausdrückt, soll mit Ergebnissen einer von der PG511 durchgeführten Befragung in Relation gesetzt und somit auf seine Korrektheit geprüft werden.

In Kapitel 2 dieses Berichts finden sich die Grundlagenarbeiten und eine Übersicht der CI-Methoden, die zur Vorbereitung und zum besseren Verständnis des Pacman-Projekts dienen. Kapitel 3 beschreibt die Aspekte des Pacman-Projekts, von der Wahl dieses bestimmten Spiels, über die Organisation der zwölf Teilnehmerinnen[1] bis hin zur Wahl von C++ als Programmiersprache des ersten Teilprojektes.

Die Kapitel 4, 5 und 6 beinhalten eine Beschreibung der Arbeiten und Ergebnisse, der in Abschnitt 3.2 vorgestellten Teilgruppen. Hierbei soll sowohl auf den theoretischen Hintergrund der jeweiligen Methoden als auch auf die Implementierungsdetails eingegangen werden, insbesondere auf die Besonderheiten der konkret verwendeten Methoden und Lernverfahren. Schwerpunkte werden sowohl auf die theoretischen Grundlagen, die eine

[1] Es wird die weibliche Form für Teilnehmerin und Teilnehmer verwendet. Diese Konvention wurde von allen Mitgliedern der PG511 zu Beginn des Projekts beschlossen und in die PG-Ordnung (Anhang N) aufgenommen.

bestimmte Implementierung verlangt, als auch auf die spezifischen Merkmale in der eigentlichen Implementierung der jeweiligen Methoden gelegt.

Die Ergebnisse der Implementierungsphase wurden auf dem „Campusfest“ der Universität Dortmund präsentiert. In dieser Umgebung wurden den freiwilligen Spielern zu ihren Erfahrungen einige Fragen gestellt, um das zweite genannte Teilziel der Evaluation des Spielspaßmaßes nach Yannakakis/Hallam zu erfüllen. Die genaue Methodik dieser Umfrage und ihre Ergebnisse finden sich in Kapitel 7.

Die Erfahrungen der PG511, die im Laufe dieses einsemestrigen Projekts gesammelt wurden, sind im Kapitel 8 zusammengefasst. Dieser Bericht schließt mit einer Zusammenfassung (Kapitel 9) und einem Ausblick (Kapitel 10) in die Arbeit der Projektgruppe im zweiten Projekt-Semester mit einem weiteren Spiel.

2 Grundlagen

In diesem Kapitel werden die Grundlagen, der von uns verwendeten CI-Methoden vermittelt, die notwendig sind, um die von uns entwickelten Methoden nachzuvollziehen. Im Anschluss wird ein Einblick in ähnliche Forschungsprojekte gegeben.

2.1 Einführung in CI-Methoden

Im Folgenden führen wir kurz in die verwendeten CI-Methoden ein. Ausführlichere Informationen über diese Methoden sind den Seminararbeiten „Evolutionäre Algorithmen“ von Jan Quadflieg (Anhang A) und „Neuronale Netze“ von Georg Neugebauer (Anhang C) zu entnehmen.

2.1.1 Evolutionäre Algorithmen

Aufgrund der Inspiration durch den Evolutionsprozess der Natur wurden viele der im Zusammenhang mit evolutionären Algorithmen auftauchenden Begriffe direkt aus der Biologie übernommen. Evolutionäre Algorithmen operieren auf einer Multimenge von *Individuen*, die *Population* genannt wird. Ein Individuum ist ein Element des Suchraums des zu lösenden Problems, stellt also eine mögliche Lösung dar. Seine Kodierung, *Repräsentation* genannt, wird von der Struktur des Suchraums bestimmt. Beispiele für mögliche Suchräume sind der Boolesche ($\mathbb{B}^n$), der reele ($\mathbb{R}^n$), der Raum der Permutationen (S_n) und Bäume.

Der Suchraum S, auf dem ein evolutionärer Algorithmus arbeitet, muss nicht zwingenderweise mit dem Suchraum A des zu lösenden Problems übereinstimmen. Die Individuen sind also Elemente von S, der auch *Genotypraum* genannt wird. A bezeichnet man als *Phänotypraum* und die Abbildung $S \rightarrow A$ als *Genotyp-Phänotyp-Abbildung*. Um die gefundenen Lösungen zu verbessern, werden iterativ aus den alten Individuen neue Individuen (*Nachkommen*) erzeugt, bis eine vorgegebene Güte oder eine maximale Anzahl von Iterationen erreicht ist. Ein solcher Iterationsschritt heißt *Generation*. Für die Erzeugung der Nachkommen greifen evolutionäre Algorithmen auf Methoden zur Reproduktion zurück, wie sie in der Natur bei geschlechtlicher und ungeschlechtlicher Fortpflanzung vorkommen: neue Individuen werden als Klon eines Individuums oder aus mehreren Elternindividuen erzeugt (*Rekombination* oder auch *Crossover*). Zusätzlich können auch zufällige Veränderungen auf die Individuen einwirken (*Mutation*).

Die *Selektion* wählt an zwei Stellen eines evolutionären Algorithmus Individuen aus: Zum einen die Individuen aus der alten Population, aus denen Nachkommen erzeugt werden. Zum anderen die Individuen der Folgepopulation aus den Nachkommen und gegebenenfalls der alten Population. Die *Fitnessfunktion* bewertet ein Individuum hinsichtlich der Qualität der Lösung, die es repräsentiert. Der Wert der Fitnessfunktion kann die *Selektion* beeinflussen, zum Beispiel bei der *fitnessproportionalen Selektion*. Fügt man all dieses zusammen, ergibt sich der in Listing 1 dargestellte Ablauf eines evolutionären Algorithmus. Im Folgenden werden die einzelnen Module detaillierter vorgestellt.

```
t:=0
Initialisierung der Anfangspopulation P(0)
Bewertung der Population P(0)
WHILE Terminierungskriterium nicht erfuellt
  Selektiere die Eltern
  Erzeuge aus den Eltern Nachkommen
          mittels Mutation und gegebenenfalls Rekombination
  Bewerte die Nachkommen
  Selektiere die Nachfolgepopulation P(t+1)
  t:=t+1
END WHILE
```

Listing 1: Pseudocode eines evolutionären Algorithmus

Initialisierung Die Initialisierung erzeugt die Individuen der Ausgangspopulation. Häufig wird hier die Gleichverteilung gewählt, wenn über die zu erreichende Lösung wenig bekannt ist oder Vorwissen nicht verwendet werden soll. Generell kann man beliebige Startpunkte als Ausgangspopulation nehmen. Wenn durch Vorkenntnisse jedoch viel versprechende Individuen bekannt sind, können auch diese als Anfangspopulation genutzt werden. Der EA wird dann schnell gegen ein lokales Optimum konvergieren [Eng02].

Fitnessfunktion Die minimale Anforderung an eine Fitnessfunktion ist, dass sie den Vergleich zweier Individuen erlaubt, um eine Entscheidung darüber zu treffen, welches der beiden Individuen das bessere ist. Beschränkt man sich allein auf diese Minimalanforderung, spricht man von *relativer Fitnessbewertung*. Gerade in Spielen kommt es häufig zu einer *Stein–Schere–Papier*[2] genannten Situation, in der bei drei Strategien A, B und C Strategie A von B, B von C und C von A geschlagen wird. Da dies eine intransitive Relation ist, existiert kein globales Optimum. Daraus folgt, dass nur die relative Fitnessbewertung möglich ist.

Eine Fitnessfunktion ist allgemein eine Abbildung $f(s) : S \to \mathbb{R}, s \in S$, häufig eingeschränkt auf $\mathbb{R}_+$[3], bei der jedes Individuum einzeln bewertet wird. Die Art der Bewertung ist dabei stark von dem zu lösenden Problem abhängig, ein praktisches Beispiel wird in Abschnitt A.4 vorgestellt. Betrachtet man ein Optimierungsproblem, enthält dieses eine Zielfunktion, die es zu minimieren oder maximieren gilt, woraus sich bei einfachen Problemen auf eine natürliche Art und Weise eine Fitnessfunktion ergibt, wie man sich am Problem des Handelsreisenden (TSP) leicht klar macht: Aufgabe ist es, eine kostengünstigste Rundreise durch gegebene Städte zu finden, zum Beispiel die Route mit den geringsten Benzinkosten. Die Benzinkosten sind die Zielfunktion, die zu minimieren ist, die Fitness eines Individuums die Benzinkosten, die es verursacht. Für theoretisch motivierte Fitnessfunktionen sei an dieser Stelle auf das Vorlesungsskript von Thomas Jansen [Jan04] verwiesen.

[2] *Stein–Schere–Papier* ist ein weltweit bekanntes Glücksspiel, das mit den Händen gespielt wird, wobei Handhaltungen Symbole zugeordnet sind, die sich gegenseitig schlagen können. In Deutschland kennt man es auch unter den Namen *Schnick, Schnack, Schnuck* oder *Ching, Chang, Chong*.

[3] Voraussetzung für die fitnessproportionale Selektion

Reproduktionsselektion Wie zu Beginn von Abschnitt 2.1.1 erwähnt, wählt die Selektion an zwei Stellen Individuen aus, zum einen als Eltern für die Reproduktion und zum anderen als überlebende Individuen für die Folgepopulation. Die dabei jeweils eingesetzten Selektionsoperatoren können durchaus unterschiedlich sein. Eine einfache Variante, Individuen zufällig gleichverteilt zu wählen, wird *uniforme Selektion* genannt.

Wenn man jedoch gute Individuen bevorzugen will, liegt es nahe, die Selektion direkt an der Fitness zu orientieren. Bei der fitnessproportionalen Selektion wird ein Individuum $s \in S$ mit der Wahrscheinlichkeit $\frac{f(s)}{\sum_{x \in S} f(x)}$ gewählt. Doch gerade diese starke Abhängigkeit vom Fitnesswert erweist sich als Nachteil: Bei großen Unterschieden in den Fitnesswerten verhält sich die fitnessproportionale Selektion fast deterministisch, es wird mit sehr großer Wahrscheinlichkeit das Individuum mit dem größten Fitnesswert ausgewählt und das unter Umständen immer wieder. Sind die Unterschiede in den Fitnesswerten dagegen sehr klein, unterscheidet sich die fitnessproportionale Selektion kaum von der uniformen Selektion.

Die *Boltzmann–Selektion* versucht diesen Nachteil zu umgehen, indem sie die Idee des *simulated annealing* ([AK89]) aufgreift und statt der Funktionswerte $f(x)$ den Ausdruck $e^{f(x)/T}$ verwendet. T ist dabei die aus dem simulated annealing bekannte *Temperatur*. Man beginnt mit einem großen Wert für T und senkt diesen von Generation zu Generation. Erst bei niedrigen Werten für T, also bei späteren Generationen, gewinnt der Wert der Fitnessfunktion an Einfluss auf den Selektionsprozess. Im Gegensatz zu den bisher vorgestellten Verfahren werden bei der *Schnitt-Selektion* nicht eines sondern eine vorher festgelegte Anzahl von Individuen mit der besten Fitness ausgewählt.

Selektion der Folgepopulation Die Auswahl der Individuen für die Folgepopulation kann natürlich ebenfalls mit den zuvor vorgestellten Operatoren erfolgen. Es ergibt sich jedoch die Frage aus welcher (Multi-)Menge die Individuen gewählt werden sollen. Bei der *Plus-Selektion* werden die μ Individuen mit dem größten Fitnesswert aus der Vereinigung der alten Individuen und deren Nachkommen deterministisch ausgewählt. Dieses wird als $(\mu + \lambda)$–Selektion notiert, wobei μ für die Größe der Population und λ für die Anzahl der Nachkommen steht. Bei der *Komma-Selektion* werden die Individuen der neuen Population ausschließlich aus den Nachkommen ausgewählt; die alte Population wird komplett verworfen. Das impliziert natürlich, dass die Anzahl der Nachkommen mindestens so groß sein muss wie die Größe der Population. Die Auswahl geschieht wieder deterministisch. Es werden die μ Nachkommen mit der größten Fitness gewählt. Analog zur Plus–Selektion wird die Komma-Selektion als (μ, λ)–Selektion notiert.

Variation Die Variation erzeugt mit Hilfe von Zufallselementen aus den vorhandenen Individuen einer Generation neue Individuen für die nachfolgende Generation. Je nachdem, ob ein neues Individuum aus einem oder aus mehreren Individuen der alten Population erzeugt wird, spricht man von *Mutation* oder *Rekombination*. Während nicht alle evolutionären Algorithmen Rekombination verwenden, wird Mutation immer eingesetzt, gegebenenfalls nach der Rekombination.

Rekombination Bei der Rekombination, auch *Crossover* genannt, wird aus mehreren, meist jedoch zwei Individuen ein Nachkomme erzeugt. Der Nachkomme erbt dabei Eigenschaften seiner Eltern. Bei Repräsentationen, die sich als Vektoren auffassen lassen (zum

Beispiel im $\mathbb{B}^n$ oder $\mathbb{R}^n$), teilt man dazu einfach den Vektor: Beim *k–Punkt–Crossover* werden k verschiedene Positionen zufällig gleichverteilt ausgewählt und die Vektoren der Eltern an diesen Stellen zerlegt. Der Nachkomme wird dann alternierend aus den $k+1$ Teilen zusammengesetzt. Es ist üblich, k sehr klein zu wählen, häufig $k = 1$ oder $k = 2$.

Beim *uniformen Crossover* wird aus zwei Eltern für jede Komponente des Vektors des Nachkommen mit Wahrscheinlichkeit jeweils $1/2$ die Komponente des einen oder des anderen Elter kopiert. Sowohl uniformes- als auch k-Punkt-Crossover, bei denen die Teile der Eltern unverändert kopiert werden, werden auch *diskrete* Rekombination genannt. Dagegen steht im $\mathbb{R}^n$ das *arithmetische* Crossover, bei dem der Nachkomme y als gewichtete Summe $y = \alpha_1 x_1 + ... + \alpha_n x_n$ aus den n Eltern $x_1, ..., x_n$ berechnet wird. Sind alle Gewichte $\alpha_i = 1/n$, spricht man von *intermediärer* Rekombination. Der Nachkomme ist dann der Schwerpunkt der Eltern.

Mutation Die Mutation verändert ein Individuum zufällig, wobei der Grad der Veränderung wie in der Natur im Allgemeinen recht klein gehalten wird. Die Art der Veränderung hängt stark von der Darstellung ab.

Typische Mutationsoperatoren im $\mathbb{B}^n$ sind die *Standard–Bit–Mutation* und die *1–Bit–Mutation*. Bei ersterer wird jedes Bit mit einer Wahrscheinlichkeit p_m geflippt, wobei p_m in der Praxis relativ klein gewählt wird. Oft verwendet man bei einem Bitstring der Länge n $p_m = 1/n$, so dass im Erwartungswert nur ein Bit verändert wird. Bei der *1–Bit–Mutation* wird immer genau ein zufällig ausgewähltes Bit verändert.

Im $\mathbb{R}^n$ bieten sich ähnliche Mutationsoperatoren wie im $\mathbb{B}^n$ an, wobei eine Komponente des Vektors $x \in \mathbb{R}^n$, der ein Individuum repräsentiert, die Rolle eines Bits im $\mathbb{B}^n$ einnimmt. Das Verfälschen der einzelnen Komponenten geschieht dann durch Addition eines Mutationsvektors $m \in \mathbb{R}^n$, dessen Komponenten bei einfachen evolutionären Algorithmen in der Regel alle gleich sind.

Parameter Evolutionäre Algorithmen lassen sich über verschiedenste Parameter beeinflussen; einige wurden bereits in den vorangegangenen Abschnitten erwähnt. Legt man diese Parameter vor dem Start des evolutionären Algorithmus fest und ändert sie danach nicht mehr, spricht man von *statischen Parametereinstellungen*. Dieses Vorgehen kann dazu führen, dass der EA bei seiner Suche langsamer fortschreitet, als eigentlich möglich wäre: Parameterwerte, die zu Beginn der Ausführung angemessen waren, können im weiteren Verlauf unpassend werden, wenn sich der EA der optimalen Lösung nähert. Es kann also sinnvoll sein, Parameter abhängig von der Anzahl durchlaufener Generationen zu verändern. Dieses Prinzip, Parameter des evolutionären Algorithmus als Funktion der Zeit zu definieren, wird *dynamische Parametereinstellung* genannt.

Der Einsatz von *dynamischer Parametereinstellung* ist jedoch bei evolutionären Algorithmen eher unüblich, da es wesentlich lohnenswerter ist, nicht nur die Zeit sondern auch andere Informationen in die Veränderung der Parameter einfließen zu lassen, wie zum Beispiel die Fitnesswerte und die Verteilung der Population im Suchraum. Man spricht dann von *adaptiver Parametereinstellung*. Dabei kann ein Feedback-Mechanismus benutzt werden, der den evolutionären Prozess überwacht und Parameter danach beurteilt, wie gut sie in jedem Schritt den Fitnesswert der Population verbessern [BFM00b].

Alle bisherigen Verfahren zur Anpassung der Parameter arbeiten deterministisch und der Benutzer hat die Kontrolle über die Veränderung der Parameter. Bei statischen Parametern ist dies offensichtlich, bei dynamischer und adaptiver Parametereinstellung erfolgt der Einfluss auf die Veränderung über die Vorgabe der gewünschten Funktionen. Bei der *Selbstadaption* unterliegt schließlich auch die Menge der Parameter, die den evolutionären Algorithmus steuern, dem evolutionären Prozess. Jedes Individuum repräsentiert dann wie gehabt einen Punkt des Suchraumes, enthält aber darüber hinaus auch eine Menge von Parametern. Bezeichnet man den Raum der möglichen Parameter mit $\mathbb{P}$, so arbeitet der EA auf dem Kreuzprodukt $S \times \mathbb{P}$ aus dem Suchraum S und dem Parameterraum $\mathbb{P}$, wobei die Fitnessfunktion weiterhin eine Abbildung $f(s) : S \rightarrow \mathbb{R}, s \in S$ ist. Bei der Erzeugung der neuen Population werden dann zuerst mit speziellen Operatoren neue Parameter erzeugt und dann mit diesen Parametern die neuen Individuen.

2.1.2 Neuronale Netze

Künstliche neuronale Netze (KNN) sind eine Abstraktion des menschlichen Gehirns. Sie werden vor allem zur Klassifikation und Funktionsapproximation eingesetzt. Eine Nervenzelle *(Neuron)* im KNN hat dabei eine Nervenzelle des Gehirns als Vorbild. Obwohl die einzelnen Neuronen sehr einfach aufgebaut sind, können KNNs komplexe Aufgaben lösen. Für die Eingabe KNN wird meist eine Kodierung der Umwelt gewählt, welche mittels Sensoren aufgenommen und zur Eingabe verarbeitet werden. Das KNN berechnet anhand dieser Eingaben problemspezifische Ausgaben, welche häufig weiter verarbeitet werden können. In unserem Fall wird die Spielsituation als Eingabe kodiert, wobei die berechnete Ausgabe das Verhalten einer Spielfigur entspricht. Ein entscheidender Vorteil der KNNs gegenüber traditionellen Algorithmen ist die Lernfähigkeit des Netzes. Nach dem erfolgreichen Training des Netzes, ist dieses in der Lage zu generalisieren und auf nicht trainierte Eingaben „intelligent“ zu reagieren. Das Netz wird durch eine Trainingsmenge, welche aus vorgefertigten Eingaben und den dazu gewünschten Ausgaben besteht, offline trainiert.

Häufig werden KNNs in drei Schichten unterteilt.

- Die Eingangsschicht: Die Signale, die an der Eingangschicht anliegen, repräsentieren den Zustand der Umwelt in geeigneter Kodierung und werden unverfälscht an die nachfolgenden Schichten weitergeleitet.
- Die versteckte Schicht: Kann aus mehreren Schichten bestehen.
- Die Ausgabeschicht: Die berechneten Ergebnisse werden an die Umwelt zurückgegeben.

Die Verbindungen zwischen zwei Neuronen erhalten ein Gewicht, wodurch eingehende Informationen entweder verstärkt oder abgeschwächt werden. Beim Training der KNN werden die Gewichte an den Kanten angepasst. Sie sind die Komponenten, die für den Lernerfolg verantwortlich sind. Abbildung 1 stellt ein Beispiel für ein neuronales Netz dar.

Die Aufgabe der Neuronen in einem KNN besteht darin, die ankommenden Signale zusammenzufassen, mittels einer Aktivierungsfunktion zu entscheiden, ob das Neuron aktiviert wird (*feuern* genannt) oder nicht, um anschließend die neue Ausgabe zu berechnen.

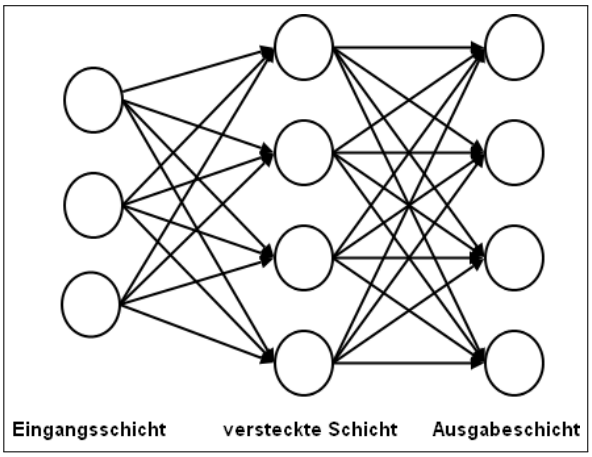


Abbildung 1: Einfaches neuronales Netz bestehend aus drei Neuronen auf der Eingabeschicht, vier Neuronen auf der versteckten Schicht und vier Neuronen auf der Ausgabeschicht

Im Allgemeinen werden hierzu die gewichteten Eingangssignale des Neurons einfach aufsummiert, so dass die Netzfunktion $f_{net} : (\mathbb{R} \times \mathbb{R})^U \rightarrow \mathbb{R}$ (Gleichung (1)) entsteht. Hierbei bestimmt die Gewichtung w_i den Einfluss der jeweiligen N Eingaben x_i bei der Berechnung der Aktivierungsfunktion [AP02]

$$f_{net}(\vec{x}, \vec{w}) = \sum_{i=1}^{N} x_i w_i = net. \tag{1}$$

Die Aktivierung des Neurons wird durch die Netzeingabe *net* und einen Schwellenwert (threshold) Θ beeinflusst. Die Aktivierungsfunktion ist in den meisten Fällen eine der folgenden.

1. Identitätsfunktion: $f(x) = x$
2. binäre Schwellenwertfunktion: $f(net) = \begin{cases} 1, & \text{wenn } net \geq \theta \\ 0, & \text{sonst} \end{cases}$
3. sigmoide Funktion: $f(net) = \frac{1}{1+exp(-net)}$

Die Ausgabefunktion berechnet auf Basis der Aktivierungsfunktion den Ausgabewert des Neurons: In den meisten Fällen wird diese als binäre Zuweisung realisiert. Wenn das Neuron feuert, wird eine 1 ausgegeben.

Jedem Neuron wird zusätzlich ein Verschiebungsterm (bias) hinzugefügt, wodurch die Aktivierungsfunktion jedes Neurons in einem anderen Wertebereich sensitiv sein kann. Dieser Term wird beim Trainieren des Netzes genau wie das Gewicht der Kanten angepasst.

Bei den KNN werden verschiedene Netzmodelle unterschieden. Diese geben vor, welche Verbindungen innerhalb eines Netzes zulässig sind. Die bekanntesten Netztypen sind [Lip07]:

- vorwärtsgerichtete (feedforward) Netze: Hierbei ist jedes Neuron einer Schicht mit jedem Neuron der nachfolgenden Schicht verbunden.
- rückgekoppelte (rekurrente) Netze: Hierbei werden auch Schleifen und gerichtete Kreise zugelassen.

Wie schon oben gesagt wurde, ist die Lernfähigkeit der neuronalen Netze die wichtigste Eigenschaft. Die prinzipliellen Möglichkeiten einem Neuronalen Netz Wissen beizubringen sind [Zel94]:

- Entwicklung neuer Verbindungen,
- Löschen esistierender 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 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 [Lip07] (Fehlerrückgabe) verwendet.

Backpropagation Backpropagation ist ein Verfahren der Fehlerrückgabe. Auf der Basis einer passenden Kodierung der Eingänge werden Testmuster erzeugt, zu welchen eine gewünschte Ausgabe bekannt ist. Auf Basis der vorgegebenen Eingaben erzeugt das neuronale Netz Ausgaben, die von den gewünschten Ausgabewerten abweichen. Diese Abweichung wird im Netz zurückpropagiert, indem die Gewichte an den Kanten des Netzes angepasst werden. Das Lernverfahren kann in drei Phasen unterteilt werden ([Lip07]):

1. Forward Pass: Dem Netz wird ein beliebiger Eingabevektor $\vec{x}$ aus der Trainingsmenge präsentiert. Das Netz berechnet Schicht für Schicht die jeweiligen internen Ausgaben und die internen Eingaben, bis die Ausgabeschicht erreicht ist, in der der Ausgabevektor $\vec{o}$ zurückgegeben wird.
2. Bestimmung des Fehlers: Zu jeder Eingabe $\vec{x}$ aus der Trainingsmenge ist die jeweilige gewünschte Ausgabe bekannt (Supervised Learning siehe Abschnitt C.4.1). Nun wird mit Hilfe einer Fehlerfunktion der Fehler des Netzes bestimmt. Falls dieser Fehler oberhalb eines „threshold“ (Güteschwelle) liegt, wird das Netz durch den Backward Pass modifiziert, ansonsten wird das Training beendet. Als Fehlerfunktion wird der mittlere quadratische Fehler (mean squared error - kurz: MSE, siehe Gleichung (2)) benutzt:

$$F(\vec{w}) = \lim_{N \to \infty} \frac{1}{N} \sum_{k=1}^{N} Q(\vec{x}_k, \vec{w}) \tag{2}$$

Der MSE ist der Grenzwert von einer möglichst großen Anzahl von Eingabevektoren $\vec{x}_k \in N$, gemittelt über die Anzahl der betrachteten Trainingsmuster (N). $F(\vec{w})$ wird als Fehler des Netzes bezüglich des Gewichtsvektors $\vec{w}$ bezeichnet. Die Funktion Q wird in Gleichung (3) definiert.

$$Q(\vec{x}, \vec{w}) = |f(\vec{x}) - o_{net}(\vec{x}, \vec{w})|^2 \tag{3}$$

Dies ist der quadratische Fehler (siehe auch [AP02] Seite 22), der bei genau einem speziellen Eingabe- und Gewichtsvektor entsteht. $f(\vec{x})$ ist hier die tatsächlich gewünschte Ausgabe des Netzes bei Eingabevektor $\vec{x}$, $o_{net}(\vec{x}$ und $\vec{w})$ ist die tatsächliche Ausgabe des neuronalen Netzes.

3. Backward Pass: In diesem Schritt werden rückwärtsgerichtet die Verbindungen zwischen den Neuronen modifiziert, das heißt, als erstes werden die Verbindungen zwischen der letzten versteckten Schicht und der Ausgabeschicht aktualisiert und als letztes werden die Verbindungen zwischen der Eingabeschicht und der ersten versteckten Schicht modifiziert. Dabei erfolgt die Modifikation mit Hilfe der in den Gleichungen (4) und (5) beschriebenen Lernregel.

$$\nabla_{\vec{w}} F(\vec{w}) = \lim_{N \to \infty} \frac{1}{N} \sum_{k=1}^{N} \nabla_{\vec{w}} Q(\vec{x}_k, \vec{w}) \tag{4}$$

$$\vec{w}(t+1) = \vec{w} - \eta \cdot \nabla_{\vec{w}} F(\vec{w}) \tag{5}$$

Backpropagation ist ein Gradientenabstiegsverfahren (siehe auch [AP02] Seite 37 ff.). Es wird versucht durch Verschieben des Gewichtsvektors $\vec{w}$ in Richtung $-\nabla_{\vec{w}} F(\vec{w})$ das Minimum der Fehlerfunktion zu bestimmen. Dabei bezeichnet $\nabla_{\vec{w}}$ den Gradienten (siehe auch [AP02] Seite 22 f.) der Fehlerfunktion bezüglich des Gewichtsvektors $\vec{w}$ und Q ist der quadratische Fehler bezüglich des Eingabevektors x_k und dem Gewichtsvektor $\vec{w}$. Nun wird versucht, ähnlich wie bei der MSE, gemittelt über die Anzahl der Eingabemuster eine Minimierung der Fehlerfunktion bezüglich möglichst vieler Eingabevektoren $\vec{x}_k$ zu erreichen. Der Gewichtsvektor $\vec{w}$ zum Zeitpunkt (t+1) ist so definiert, dass man vom aktuellen Gewicht den Gradienten $\nabla_{\vec{w}} F(\vec{w})$ multipliziert mit einem konstanten Faktor η (Schrittweite) subtrahiert. Danach erfolgt wieder Schritt 2 (Fehlerbestimmung), bis ein lokales (im besten Fall das globale) Minimum erreicht ist.

Für weiterführende Literatur bezüglich Backpropagation sei auf [AP02] und [Lip07] verwiesen. Dabei stellt die erste Quelle das Verfahren mathematisch formal dar, während die zweite Quelle einen illustrativen Ansatz verfolgt und mit vielen Beispielen zum Verständnis beiträgt. Letztlich wird in beiden Quellen auf mögliche Probleme beim Einsatz von Backpropagation eingegangen. In Abschnitt 5.1 werden diese Probleme, die während der Testphase auftraten, näher erläutert.

2.2 Seminare (Abstracts)

Zur Vorbereitung der Projektgruppe wurden im Rahmen eines Blockseminars zwölf Vorträge gehalten, deren Ausarbeitungen im Anhang zu finden sind. Es folgen die Abstracts zur Übersicht.

2.2.1 Evolutionäre Algorithmen

Die Seminarausarbeitung zu evolutionären Algorithmen (Anhang A) bietet eine knappe Einführung in das Thema. Nach einer kurzen Motivation zum Einsatz dieser, von der Natur inspirierten, randomisierten Suchheuristiken, werden die einzelnen Module eines evolutionären Algorithmus und deren mögliche Implementierung vorgestellt. Nach einer kurzen Einordnung historisch begründeter Namen für verschiedene Varianten evolutionärer Algorithmen wird die Ausarbeitung von einem praktischen Beispiel abgeschlossen: Vorgestellt wird die Arbeit von Rick Strom, der mit Hilfe eines evolutionären Algorithmus Wegfindungsalgorithmen für Strategiespiele erzeugt hat.

2.2.2 Fuzzy-Systeme

In der Seminarausarbeitung zu Fuzzy-Systemen (Anhang B) wird die Fuzzy-Logik motiviert. Darauf aufbauend wird die Fuzzy-Logik eingeführt, angefangen bei der Begriffsdefinition über t-Norm, s-Norm und Komplement bis hin zu Fuzzy-IF-THEN-Regelbasen. Alle Definitionen werden an Beispielen nochmals verdeutlicht. Den Abschluss der Arbeit bildet ein Kapitel über mögliche Modelle und Strukturen zur Konzeption von Fuzzy-Systemen, sowie die Beschreibung eines umfangreicheren Praxis-Einsatzes von Fuzzy-Systemen.

2.2.3 Neuronale Netze

Das Nervensystem ist eines der wichtigsten Bestandteile eines Säugetiers, da es auf die Aufnahme und Verarbeitung von Signalen spezialisiert ist, was erst intelligentes Verhalten erlaubt. Aus diesem biologischen Vorbild sind künstliche neuronale Netze entstanden. Das Seminar über künstliche neuronale Netze (Anhang C) erläutert zuerst den biologischen Bezug. Danach wird systematisch erst ein einzelnes künstliches Neuron, danach verschiedene neuronale Netze und letztlich der Lernvorgang beschrieben. Es werden sowohl formale Definitionen, als auch anschauliche Beispiele verwendet, um dem Leser den Bereich der künstlichen neuronalen Netze näher zu bringen. Motiviert durch das Projektgruppen-Thema „CI in Games“, wird im letzten Teil der Seminararbeit der Bezug zu Spielen aufgegriffen und der Einsatz von neuronalen Netzen in den Spielen Counter-Strike ([Wik07b]), Pacman ([Wik07c]) und Nero ([Aus07]) demonstriert.

2.2.4 Machine Learning, Regelbasierte Steuerung

Die Seminararbeit (Anhang D) klärt zu Beginn kurz einige grundlegende Begriffe in Bezug auf logisches Schließen. Außerdem werden der Begriff *Regel* definiert und Beispiele dafür gegeben.

Daraufhin folgt ein Überblick über maschinelles Lernen. Verfahren auf diesem Gebiet sind im Wesentlichen in unüberwachtes, verstärkendes und überwachtes Lernen einzuteilen. Als Beispiel für unüberwachtes Lernen wird Clustering behandelt. Entscheidungsbäume sind eine wichtige Anwendung auf dem Gebiet der überwachten Lernverfahren.

Das Ableiten von Regeln aus Entscheidungsbäumen liefert eine Überleitung zu regelbasierten Systemen. In diesem Teil werden Praktiken zum Einsatz von Regeln in Spielen

beschrieben. Dazu gehören unter anderem Algorithmen zur Wissensverarbeitung durch Vorwärts- und Rückwärtsverkettung von Regeln und zur Beschleunigung der Vorwärtsverkettung durch Preprocessing. Abschließend wird die Umsetzung eines Regelsystems in einem Spiel erklärt.

2.2.5 Learning Classifier Systems

Der Seminarvortrag „Learning Classifier Systems“ (LCS, Anhang E) thematisiert die Grundgedanken des Konzeptes der LCS nach J. H. Holland [BK05] und die auf Basis des LCS entstandene Weiterentwicklung des „Zeroth Level Classifier System“ durch Steward W. Wilson [Wil94].

Das Konzept der LCS stellt eine Technik des maschinellen Lernens dar, welches evolutionäre Algorithmen, reinforcement learning, überwachtes Lernen, unüberwachtes Lernen und heuristische Mittel kombiniert, um ein adaptives System zu erzeugen. Auf einer zufällig oder durch den Entwickler erzeugten Regelbasis, soll das System mit der Umwelt interagieren und sein Verhalten anpassen. Nachdem eine Veränderung in der Umwelt durchgeführt wurde, werden die Regeln, die an dieser Veränderung beteiligt waren, belohnt oder gegebenenfalls bestraft.

2.2.6 Multi-Agenten Systeme

Die Seminararbeit „Multi-Agenten Systeme“ (Anhang F) bietet einen kleinen Überblick über das recht komplexe Forschungsgebiet. Neben den Grundlagen (Softwareagenten) werden die Ideen und Mechanismen hinter dem Konzept agentenbasierter Systeme genauer beleuchtet. Der Schwerpunkt liegt hier vor allem auf den Vorgängen und Abläufen der Kommunikation, Koordination und Kooperation.

2.2.7 Wahrscheinlichkeitsrechnung, Statistik

Diese Seminararbeit (Anhang G) vermittelt Basiskenntnisse über grundlegende Methoden der Wahrscheinlichkeitsrechnung und mathematischen Statistik. Sie umfasst Konzepte diskreter und kontinuierlicher Wahrscheinlichkeitsrechnung wie Verteilungsfunktionen, bedingte Wahrscheinlichkeiten, Unabhängigkeit von Ereignissen und Zufallsvariablen. Einige werden zum besseren Verständnis an einer beispielhaften Analyse evolutionärer Algorithmen motiviert. Des Weiteren gibt die Arbeit einen Überblick über statistische Begriffe sowie Verfahren zum Testen von Hypothesen. All diese Methoden sind wichtiger Bestandteil der Computational Intelligence und Voraussetzung für ein gutes Verständnis der CI-Methoden.

2.2.8 Experimentelle Analyse von Algorithmen

Der Seminarbericht „Experimentelle Analyse von Algorithmen“ (Anhang H) gibt einen Überblick über die gegenwärtige Anwendung von Experimenten in der Informatik.

Im Unterschied zu der experimentellen Praxis in den Naturwissenschaften werden in der Informatik Experimente überwiegend zum Vergleichen von Implementierungen eingesetzt;

dies schöpft aber die Möglichkeiten der experimentellen Resultate nicht vollständig aus. Experimente als Simulationen können auch wichtige Aussagen über die Eigenschaften eines einzelnen Algorithmus liefern und damit zu effizienteren Implementierungen führen. Das Planen und der Aufbau von Experimenten geschehen in Anlehnung an die Naturwissenschaften und die Resultate werden mit gängigen statistischen Methoden analysiert und bewertet. Dabei kann man die klassischen Modelle, wie sie im Design of Experiments (DOE) postuliert werden, oder aber die moderne Methode DACE (Design and Analysis of Computer Experiments) wählen. Abschließend ist anzumerken, dass es noch kein standardisiertes Vorgehen gibt, über die Experimente zu berichten. In dieser Ausarbeitung wird eine Vorgehensweise vorgestellt, die sich bisher als sinnvoll erwiesen hat.

2.2.9 Spieltheorie

Die Spieltheorie ist eine mathematische Disziplin, die sich historisch aus der Analyse von Gesellschaftsspielen entwickelt hat. Heute wird sie in vielen Bereichen, wie unter anderem der Ökonomie, den Politikwissenschaften, der Evolutionsbiologie und Soziologie eingesetzt.

Die vorliegende Seminararbeit (Anhang I) führt in die Grundlagen der

Spieltheorie ein und beschreibt gängige Lösungskonzepte, wie neben weiteren das berühmte Gleichgewicht von John Nash, gibt eine Einführung in die Theorie des Nutzens und die Analyse von konkreten Spielen. Abschließend wird die Anwendbarkeit der Spieltheorie auf das sich der Projektgruppe im ersten Projekt stellende Problem der Verbesserung der Computergegner im Spiel Pacman diskutiert.

2.2.10 Experimentelle Bewertung/Vergleich von Spielstrategien und Spielspaß

In dieser Seminarausarbeitung (Anhang J) geht es vorzüglich um den Versuch ein Maß aufzustellen, anhand dessen man den Spielspaß von Spielen messen kann. Die Motivation des Papers von Yannakakis und Hallam [YH05], welche Faktoren das Maß aufgreifen soll, werden verdeutlicht. Das zugrunde liegende Paper sollte im Rahmen der Feldstudie während des Campusfestes validiert werden (Kapitel 7).

2.2.11 Module von intelligenten Systemen in Spielen

In der Seminarausarbeitung „Module für Intelligente Systeme“ (Anhang K) wird das Konzept des Aufbaus einer künstlichen Intelligenz (KI) in einem siebenstufigen Modell vorgestellt. Jede Stufe stellt eine weitere Verfeinerung in Aufbau der KI dar, wobei in jede Stufe eine Verbesserung des Vorgängers erfolgt. Dabei wird stets auf die Probleme der aktuellen Stufe hingewiesen, welche im nächsten Schritt durch weitere Optimierung behoben werden.

Dieses Vorgehen wird beispielhaft am Aufbau der Künstlichen Intelligenz von Pacman und NERO vorgestellt. Bei der Betrachtung der Künstlichen Intelligenz der Geister bei Pacman werden die FSM und der Pathfinding Algorithmus, der die Bewegung der Geister steuert, beschrieben. Bei der Betrachtung der Künstlichen Intelligenz von NERO werden die neuartigen Konzepte vorgestellt, so zum Beispiel das „real-time NeuroEvolution of Augmenting

Topologies“ genannte Konzept, nach welchem die neuronalen Netze (Gehirne) der Roboter mit jeder nachfolgenden Generation durch Kombination der Methoden neuronaler Netze (Abschnitt 2.1.2) und evolutionärer Strategien (Abschnitt 2.1.1) dazulernen.

Um einen Überblick über die bisherige Entwicklung von KIs in Computerspielen zu geben, gibt das Seminar einen historischen Überblick von Spielen mit den zu ihrer Zeit innovativen KI-Konzepten, gefolgt von einem Ausblick in die zu erwartenden Entwicklungen und Forschung in diesem Bereich.

2.2.12 Kommunikation Spiel/Spielstrategie, Spiel/Spielanalysesysteme

In diesem Seminarbeitrag (Anhang L) wird die Kommunikation in Computerspielen betrachtet. Ein besonderes Augenmerk wird auf die Ansätze in Multiplayer-Spielen, zusammen mit Methoden, den Kommunikationsaufwand zu verringern, gelegt.

Für die Kommunikation stehen die zwei grundlegend unterschiedlichen Konzepte des *Event-Driven-Behavior* und *Polling* im Vordergrund, deren Unterschiede und unterschiedlicher Kommunikationsaufwand diskutiert werden.

Die Betrachtung vom eigentlichen Spiel unabhängiger Kommunikationsmethoden führt zur Betrachtung von generellen Spielarchitekturen, gerade die von großer Kommunikation Massive Multiplayer Online Real-Time Strategy Games (MMORTs) und die unterschiedlichen Ansätze, den Kommunikationsaufwand zu bewältigen, wobei insbesondere Peer-To-Peer Netze, *Statistical Client Prediction* und *World segmentation* als Methoden betrachtet werden. Als letzter Punkt in diesem Seminarbeitrag wird mit Influence Maps (räumliche zur Laufzeit erzeugte Entscheidungshilfen für die künstliche Intelligenz) eine Methodik erklärt, die aufzeigen soll, wie eine Sichtweise auf die aktuelle Spielwelt für einen Non-Player-Character (NPC) verwaltet werden kann.

2.3 Verwandte Arbeiten

Neuronale Netze Im Bereich der Spielpraxis sind neuronale Netze (immer noch) ein zaghaft eingesetztes Mittel. Ein interessantes Merkmal im Gebrauch neuronaler Netze ist ihre Anwendung zu unterschiedlichsten Zwecken. So schlagen Graham et al. [GMS04] einen auf neuronalen Netzen basierenden Algorithmus zur Wegfindung vor. Die Autoren halten ihren Ansatz insbesondere in sich dynamisch verändernden Umgebungen als eine sinnvolle Alternative zu den auf statischen Umgebungsinformationen basierenden Algorithmen. Der Dynamik des System begegnet ein Agent im Spiel durch die ihm gegebenen Sensoren. Die Aufgabe, die Informationen, die die Sensoren aufnehmen, auf ihren Nutzwert für den Agenten hin zu filtern, übernimmt ein feedforward neuronales Netz. Das Netz wird, falls es einfache Regelmengen lernen soll, mit Backpropagation trainiert, ansonsten mit Reinforcment Learning.

Kalyanpur und Simon [KS01a] beschreiben einen hybriden Ansatz aus evolutionären Algorithmen und neuronalen Netzen. Das neuronale Netz verwendet zum Lernen den Backpropagation Algorithmus und wird dazu benutzt, die Mutations- und Crossoverwahrscheinlickeiten zu optimieren. Dieser Ansatz wurde speziell für Pacman entwickelt und auf Spielen mit Teams von drei Geistern getestet.

In ihrer Arbeit zur Spielspaßmessung [YH05] stellen Yannakakis und Hallam ebenfalls einen hybriden Ansatz vor. Das neuronale Netz ist dabei für die Bewegungen der Geister zuständig, was unserem Ansatz am nächsten kommt. Der evolutionäre Teil ist für die initiale Belegung der Netzgewichte zuständig.

Evolutionäre Algorithmen Evolutionäre Algorithmen finden häufig Einsatz in Computerspielen, wobei der Trend zu Mischalgorithmen aus evolutionären Algorithmen und neuronalen Netzen deutlich ist. Stellvertretend für die vielen Arbeiten auf diesem Gebiet lassen sich die folgenden anführen:

Thompson et al. [TLH07] stellen einen co-evolutionären Ansatz zur Entwicklung von Computergegnern vor, wobei die Anwendung auf einem speziell entwickelten Spiel, EvoTanks, getestet wird. Die „Tanks“ werden durch Ansammlung von Chromosomen repräsentiert und zufällig initialisiert. In der Fitnessfunktion wird berücksichtigt, wie schnell (effizient) ein Tank den Gegner besiegt hat und wieviel er dabei an Kraft verloren hat. Die Auswertung der Fitnessfunktion findet nach jeder Runde statt, wobei ein Kampf bis zum Sieg als eine Runde gilt. Die Tanks benutzen auch ein neuronales Netz, das aus 12 Neuronen besteht.

Determinismus In diesem Bereich werden oft *Finite State Machines* (FSM erklärt in Abschnitt K.3) eingesetzt. Diese stellen die endlichen Automaten mit einer Menge von Zuständen, Zustandsübergängen und Aktionen dar. Bei den endlichen Automaten handelt es sich um einen Sonderfall aus der Menge der Automaten. Ein Zustand speichert die Information über die Vergangenheit, das heißt er reflektiert die Änderungen der Eingabe seit dem Systemstart bis zum aktuellen Zeitpunkt. Ein Beispiel für eine FSM bei Pacman gibt die folgende Abbildung 2.

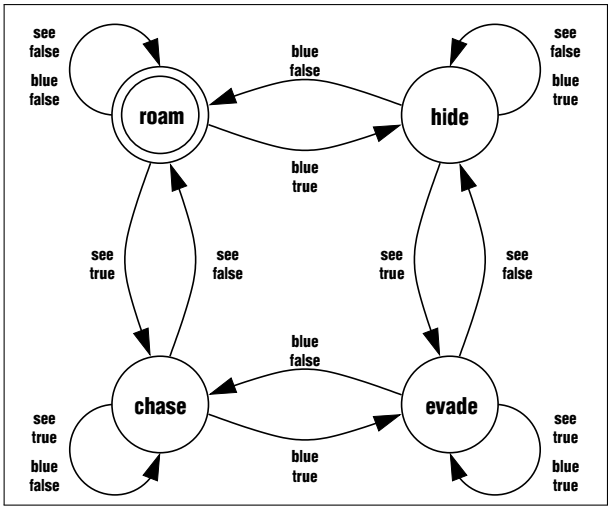


Abbildung 2: Finite State Machine für Geistersteuerung in Pacman. Der Pacman kann je nach Spielsituation zwischen den Zuständen: Herumlaufen, Jagen, Verstecken und Weglaufen wechseln.

3 Erstes Projekt: Pacman

Zum Erlernen des Einsatzes der CI-Methoden wurde als erstes Spiel der Klassiker „Pac-Man“ von den Betreuern der PG vorgeschlagen.

Pac-Man ist ein Arcade- und Videospiel der japanischen Firma Namco, das 1980 unter dem Namen „Puc-Man“ und dann 1981 in den USA unter dem (bekannteren) Namen „Pac-Man“ veröffentlicht wurde. In diesem Spiel tritt der Spieler als „Pac-Man“ gegen vier Geister an. In einem Labyrinth (Abbildung 3, links) versucht der Spieler, alle Futterpillen (im Folgenden auch als Pellets bezeichnet) zu fressen, ohne sich dabei seinerseits von den Geistern fangen zu lassen. Um dem Spieler einen kleinen Vorteil zu gewähren, gibt es vier besondere Kraftpillen, die den Spieler für einen kurzen Zeitraum in die Lage versetzen, die Geister zu jagen und zu fressen. Diese werden dann für einige Sekunden aus dem Spiel genommen, um anschließend die Jagd erneut aufzunehmen.

In Pacman[4] können sowohl der Pacman als auch die Geister automatisiert werden. Auch kann die Anbindung der implementierten CI-Methoden an ein einfaches Spiel und damit auch an eine einfachere Umgebung für einen hoffentlich beschleunigten Einstieg sorgen.

Eine weitere interessante Möglichkeit ist der Versuch einer Verfikation beziehungsweise Falsifikation des Papers von Yannakakis und Hallam, die speziell für Pacman die These aufgestellt haben, dass der Spielspaß mittels einer Formel messbar sei. Dementsprechend wurde das Originallabyrinth in dem von uns verwendeten Klon „Njam“ (Abschnitt 3.1) nachgebaut (Abbildung 3, rechts).

3.1 NJam

NJam [Bab03] ist ein unter der GPL verfügbarer Klon des Spieleklassikers Pac-Man von Milan Babuskov. Die Wahl auf NJam fiel in zwei Schritten: In einer ersten Diskussion einigten sich die Teilnehmerinnen der Projektgruppe auf die Programmiersprache C++. Das entscheidende Argument zugunsten von C++ war die Tatsache, dass im zweiten Semester wahrscheinlich ein in C++ geschriebenes Spiel bearbeitet werden wird. Es erschien der Projektgruppe unter diesem Umstand einfacher, bereits das erste Semester dafür zu nutzen, dass sich alle Teilnehmerinnen in C++ einarbeiten können.

Nachdem die Wahl auf die Programmiersprache C++ gefallen war, wurde in einem zweiten Schritt eine Implementierung von Pacman ausgewählt, die im Quellcode verfügbar ist und somit als Ausgangsbasis für alle weiteren Arbeiten dienen konnte. Durch Einschränkung auf C++ blieben von den im Tutorium Pacman vorgestellten Programmen nur NJam und Pac^2 (Pac-Squared) übrig. Nach Studium der Quelltexte fiel dann die Wahl auf NJam, da dessen Quelltext bei erster Durchsicht den Eindruck machte, wesentlich besser strukturiert zu sein. Somit wären die geplanten Änderungen einfacher durchzuführen. Dieser erste Eindruck hat sich im Laufe des Projekts weitestgehend bestätigt.

Implementierungsdetails Zentraler Bestandteil von NJam sind die Klassen NJamEngine und NJamMap. NJamEngine enthält die Initialisierung der Simple DirectMedia Layer

[4] Sofern nicht Namcos Originalspiel gemeint ist, wird in der vorliegenden Arbeit der Begriff „Pacman“ für das Spiel „Pac-Man“ und seine Nachfolger und Nachbauten verwendet. Ebenso wird die Spielfigur des „Pac-Man“ analog dazu als „Pacman“ bezeichnet.

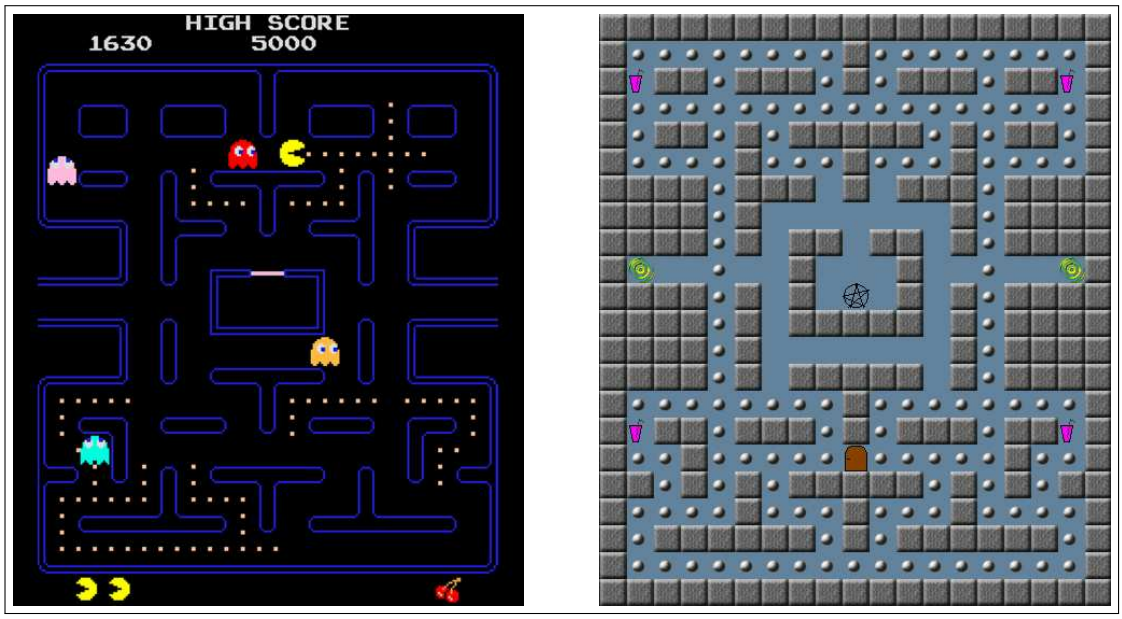


Abbildung 3: Das ursprüngliche Pac-Man Labyrinth von Namco (links) und der Nachbau dessen in NJam (rechts). Der „Tunnel“ zwischen den Spielfeldseiten des Originals wurde in NJam durch „Teleporter“ ersetzt, die nur von Pacman und nicht von den Geistern passiert werden können.

Bibliothek (SDL), das Laden sämtlicher Resourcen (Grafiken, Musik und Geräusche), Code für die Darstellung der grafischen Benutzeroberfläche (GUI) und die komplette Spiellogik. NJamMap modelliert das Spielfeld, welches in einem zweidimensionalen Array gespeichert ist. Ein Element des Arrays entspricht einer Kachel (Tile) des Spielfelds und kann entweder eine Wand (nicht passierbar), ein leeres Feld oder ein besonderer Gegenstand sein (Teleporter, Pille, etc).

Die aktuelle Position einer Spielfigur (ein Geist oder Pacman) wird über die Koordinate der Kachel bestimmt, auf der sich die Figur gerade befindet (Abbildung 4). Da die Kacheln recht groß sind, wäre jedoch keine optisch flüssige Bewegung der Spielfiguren möglich, wenn ein einzelner Schritt die Bewegung von einer Kachel zur nächsten wäre. Aus diesem Grund unterteilt NJam jede Kachel in 5 Subpositionen: ein diskreter Schritt einer Spielfigur ist die Bewegung von einer Subposition zur nächsten. Wird dabei der Rand der Kachel erreicht, wechselt die Spielfigur im gleichen Schritt auch zur nächsten Kachel.

Aus dem mangelnden Verständnis für den Ablauf der Bewegung und für das Koordinatensystem ergaben sich im Laufe des Projektes immer wieder Schwierigkeiten: Erste Versionen der Geister bewegten sich durch Wände, weil die von uns veränderte Methode NJamEngine::moveGhosts die Geisterbewegungen nicht mehr korrekt auswertete. Als Konsequenz wurde zuerst diese Methode und später auch die Methode NJamEngine::movePlayers (wertet die Bewegung von Pacman aus) von uns komplett neu geschrieben.

Aus dem Konzept der Subpositionen ergibt sich unmittelbar, dass die Spielfiguren nur dann an einer Kreuzung die Richtung wechseln können, wenn sie sich genau auf der Mitte der Kachel befinden. Andernfalls würden die Figuren in die sie umgebenden Wände hineinragen. Unsere Geister entschieden aber über einen möglichen Richtungswechsel nur

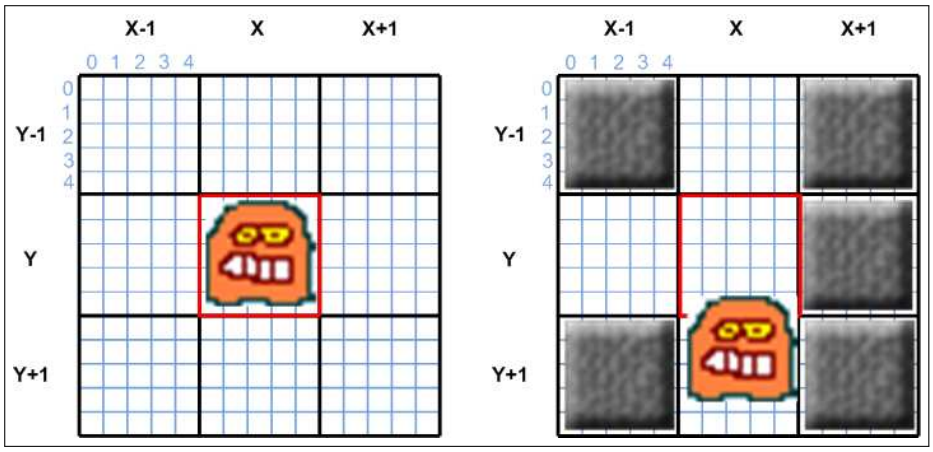


Abbildung 4: Das Koordinatensystem von NJam. Fett gedruckt sind die Koordinaten der Kacheln zu sehen, in blauer Schrift die Subpositionen innerhalb einer Kachel. Die Kachel, auf der sich der Geist befindet, ist rot umrandet. Im linken Bild befindet sich der Geist genau mittig auf der Kachel (Koordinate (x,y), Subpositionen (0,0)), im rechten Bild dagegen nicht (Koordinate (x,y), Subposition (0,4). Daraus entsteht im rechten Bild eine Situation, in der der Geist nicht nach links abbiegen kann, obwohl die Kachel links von ihm (Koordinate x-1, y) frei ist.

anhand der Tatsache, ob in einer bestimmten Richtung eine Wand ist (Richtungswechsel nicht möglich) oder nicht (Richtungswechsel möglich). NJamEngine::moveGhosts lehnt dann jedoch die gewünschte Richtungsänderung im Rahmen der Kollisionserkennung ab, als Resultat blieben die Geister an Kreuzungen einfach stehen. Die langfristige Lösung bestand aus einer kleinen Hilfsmethode, die im Bewegungscode der Geister aufgerufen wird, und die zurückgibt, ob eine Richtungsänderung möglich ist oder nicht.

3.2 Organisation

Eine der ersten Aufgaben der PG war die Erstellung einer PG-Ordnung, die die Arbeit und den Umgang mit der PG regeln sollte. Schwierig hierbei waren die Entscheidungen über die Abstimmungsmodi, um festzulegen, wie Entscheidungen in der PG getroffen werden. Dabei war insbesondere zu beachten, dass auch in Abwesenheit einzelner Mitglieder die PG als solche beschlussfähig bleibt, aber bei wesentlichen Entscheidungen trotzdem niemand übergangen wird. Die PG-Ordnung ist in Anhang N zu finden. Ein weiteres wichtiges Thema war der Punkt „Erreichbarkeit“. Da die Sitzungen der PG in kurzen Zeitabständen aufeinander folgen, wurde mit der PG-Ordnung festgelegt, dass eine Reaktionszeit von 24 Stunden gewährleistet wird. Um dies zu erreichen wurde festgelegt, dass mindestens alle 24 Stunden die E-Mails abgerufen werden müssen. Um einen zentralen Übersichtspunkt über alle Aktivitäten und bisherigen Ergebnisse zu haben sowie um aktuelle Themen zu diskutieren, hat die PG ein Wiki [LC01] eingerichtet. Über einen Strafenkatalog für eine Kaffeekasse wurde zwar diskutiert, der Vorschlag wurde jedoch abgelehnt, da zu Beginn der PG nicht der Eindruck entstand, als ob dies notwendig sei. Zu Beginn des zweiten Teilprojekts wird diese Annahme nochmals überdacht werden.

Nachdem die Projektgruppe sich für das Spiel „NJam“ als Pacman-Implementierung entschieden und sich damit endgültig auf die Programmiersprache C++ geeinigt hat, wurde eine Schnittstelle entworfen, die es allen CI-Methoden erlaubt, auf die benötigten Spielinformationen direkt zuzugreifen ohne spezielle Kenntnisse über die Abläufe im Spiel zu haben.

Zur Implementierung und dem weiteren Vorgehen wurden Arbeitsgruppen zu den Themen „NJam“(Jan Quadflieg, Raphael Stüer, Simon Wessing), „Neuronale Netze“ (Tobias Hein, Georg Neugebauer, Raphael Stüer, Andreas Thom), „Evolutionäre Algorithmen“(Nico Piatkowski, Sebastian Schnelker, Simon Wessing), „Deterministische Geister“(Holger Danielsiek, Christian Eichhorn, Edina Kurtić, Michael Puchowezki, Raphael Stüer) und „Auswertungen“(Holger Danielsiek, Christian Eichhorn, Edina Kurtić) gebildet. Im weiteren Verlauf der PG wurden noch die Gruppen „Campusfest“ (Andreas Thom, Christian Eichhorn, Holger Danielsiek) sowie „Zwischenbericht“ (Holger Danielsiek, Christian Eichhorn, Tobias Hein, Jan Quadflieg, Raphael Stüer, Andreas Thom) gebildet, um schnell auf die kurzfristigen Anforderungen der Themenbereiche eingehen zu können.

Basis der Entwicklung war der Zeitplan (Abbildung 5) mit dem Campusfest[5] als Stichtag, der zu Beginn erstellt wurde. Dabei ist im Zeitplan zu sehen, dass zunächst einfache Basisimplementationen der jeweiligen CI-Methoden geplant wurden. Diese sollten nach einem Auswertungstestlauf erweitert werden. Der Zwischenbericht wurde die ganze Zeit als laufende Dokumentation erstellt, indem Texte direkt passend aufbereitet im Wiki untergebracht wurden.

Für die tägliche Arbeit der PG wurde ein zusätzlicher Raum sowie ein Schrank zur Unterbringung von Netzwerkequipment und anderen Sachen organisiert, wodurch die PG auch außerhalb der regulären Sitzungen gemeinsam schnell und effizient arbeiten konnte. Zusätzlich zu den bereits erwähnten regulären Sitzungen wurde von der PG eine weitere Sitzung eingeführt, in der Dinge besprochen wurden die aus Zeitgründen in den regulären Sitzungen nicht mehr entschieden werden konnten. Dabei ist zu erwähnen, dass bei der zusätzlichen Sitzung die Betreuer nicht anwesend waren.

[5] Seit 1991 feiert die Universität Dortmund einmal im Jahr ihr Campusfest, einen „Tag der offenen Tür“, an dem sich die Hochschule allen Interessierten präsentiert.
Neben dem wissenschaftlichen Bereich, mit zahlreichen Mitmach-Experimenten und Schnuppervorlesungen kommt auch das leibliche Wohl nicht zu kurz. Rund um die Mensabrücke kann man Köstlichkeiten aus aller Welt genießen, zubereitet von deutschen und ausländischen Studierenden der Universität Dortmund. (Quelle: `http://www.uni-dortmund.de/uni/Uni/Campusleben/Campusfest/index.html`, online, 10. Oktober 2007)

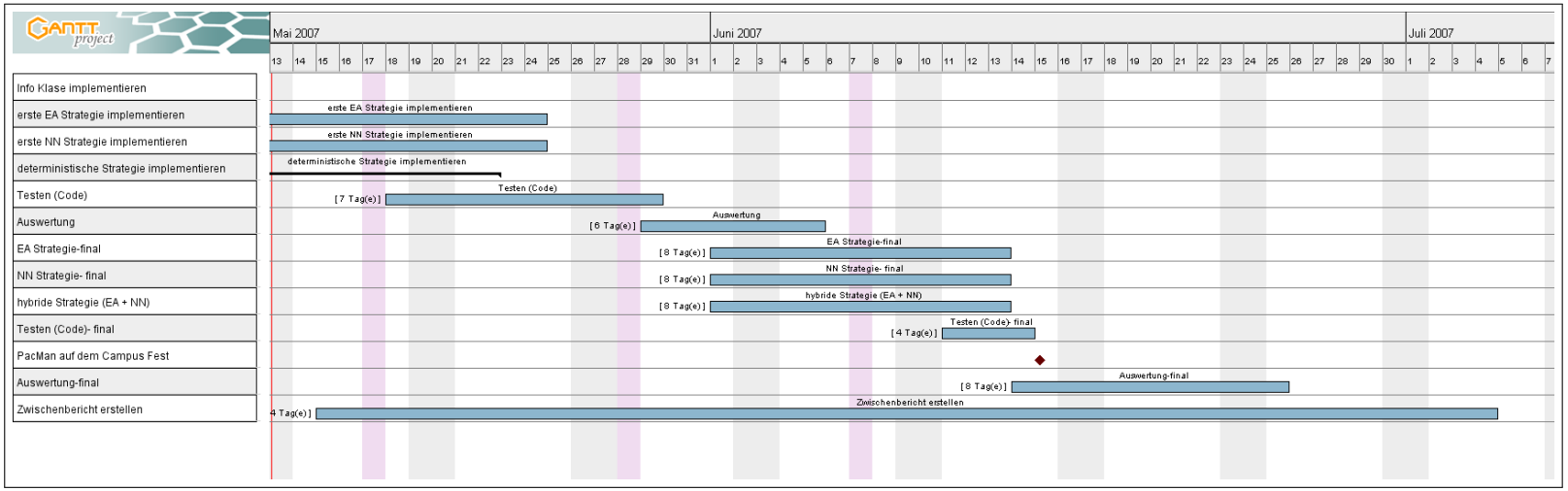


Abbildung 5: Die Projektplanung zu Pacman. Die Grafik zeigt unseren ersten Entwurf des Zeitplans des Projekts.

3.3 Schnittstelle zu NJam

Die Idee, in der Schnittstelle alle Informationen an zentraler Stelle abrufbar zu machen, wurde durch die Klasse „GameInfo“ (Abbildung 6) realisiert. Hierzu werden aus dem Spielablauf heraus alle relevanten Spielinformationen sofort im GameInfo Objekt gespeichert.

Die Klasse GameInfo wurde als Singleton [GHJV95] implementiert, um einen einfachen Zugriff auf das Objekt von allen Stellen des Quelltextes aus zu ermöglichen. Die einzige Instanz von GameInfo hält die Referenz auf alle Akteure (Geister und Pacman) in Form der Klasse „NPC“. Zur Kopplung der CI-Methoden an das Spiel werden die abstrakten Implementierungen mittels einer Adapterklasse [GHJV95] gekapselt, die von „NPC“ erbt. Im Spielablauf wird immer, wenn das Spiel eine Aktion von einem der Akteure erwartet, die Methode „move()“ des „NPC“ aufgerufen. Die CI-Methoden übernehmen dann an dieser Stelle die Berechnung des nächsten Zugs.

Da einige Methoden auf Graphen arbeiten, um zum Beispiel bekannte Methoden der Wegfindung zu nutzen, wird ein Graph zur Karte erstellt und separat gespeichert.

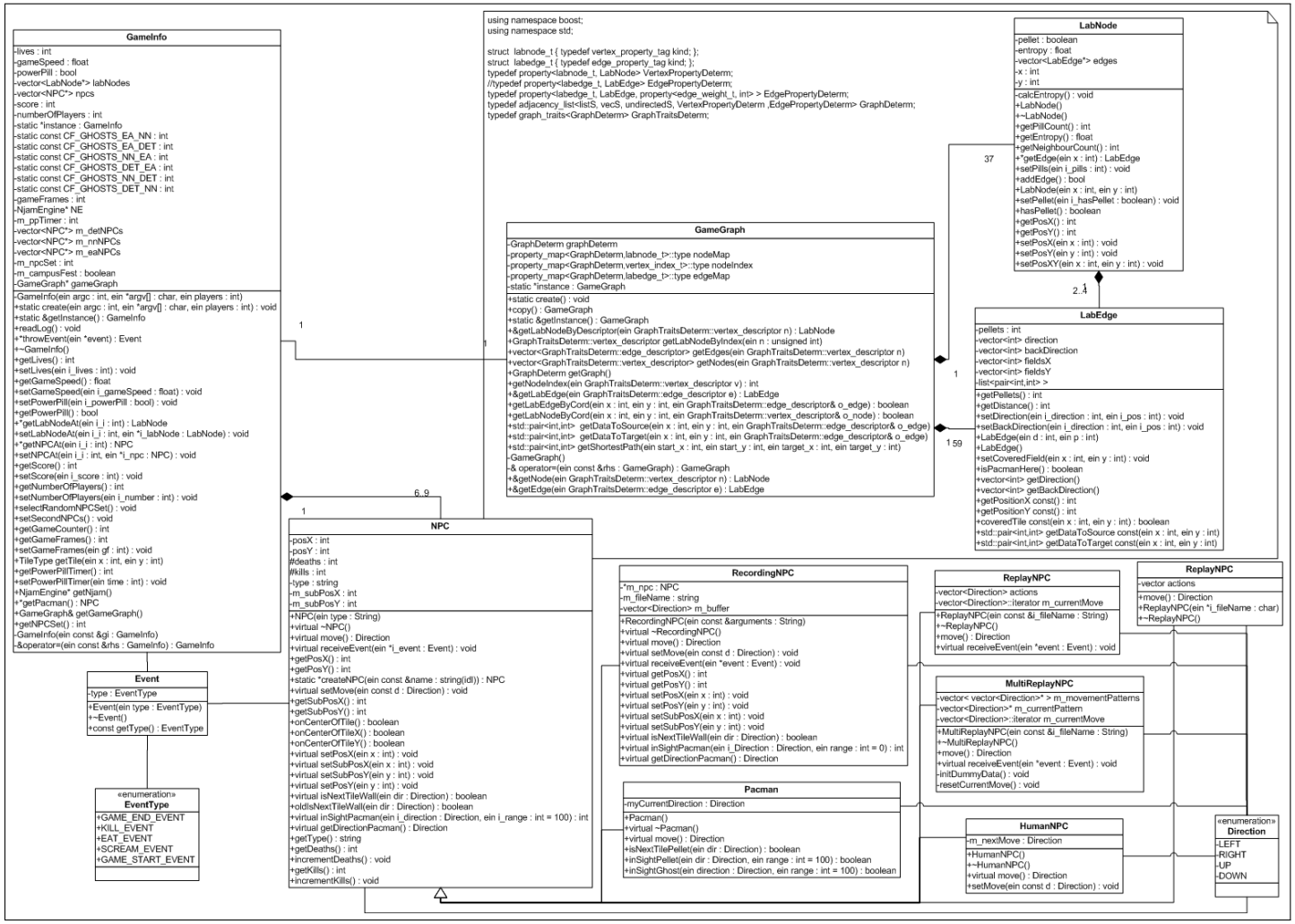


Abbildung 6: Die Schnittstellen-Implementierung zu Pacman. Das Klassendiagramm zeigt die Klassen der Schnittstelle mit ihren Variablen, Methoden und den jeweiligen Sichtbarkeiten sowie die Abhängigkeiten der Klassen untereinander.

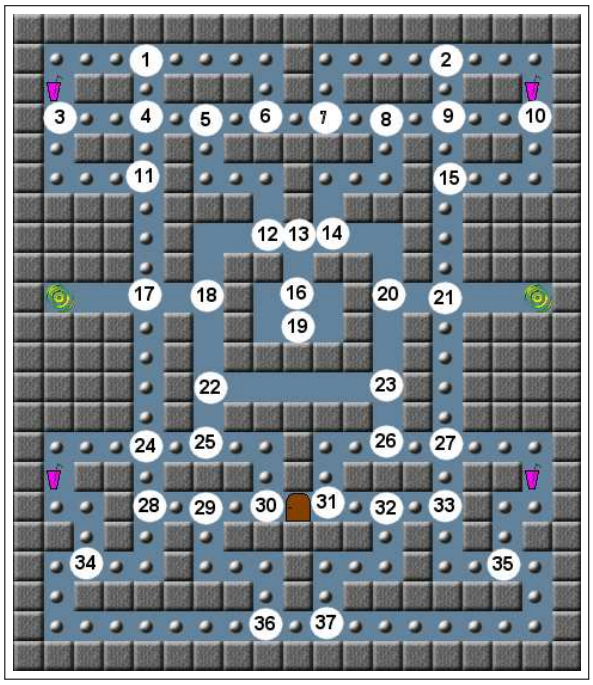


Abbildung 7: Das Labyrinth mit für den Spielgraphen nummerierten Kreuzungen.

3.4 Spielgraph

Da viele der besprochenen Gegner Wissen über das Labyrinth besitzen oder ansammeln und innerhalb des Labyrinths Wege berechnen, haben wir uns für eine Darstellung des Labyrinths als Graph entschieden. Hierbei sind Kreuzungen im Labyrinth als Knoten und Wege von einer Kreuzung zur anderen als Kanten repräsentiert. Die Kanten sind jeweils mit Wegkosten der Länge des Pfades in diskreten Spielfeldkacheln gewichtet (Abbildungen 7 und 8) und speichern die Schrittfolgen, die dem Verlauf der Kante auf dem Spielfeld in beiden Richtungen entsprechen.

Programmtechnisch ist der Graph im Graphkonstrukt der Boost-Bibliothek ([Boo07]) erstellt, die zu dem Graphen effiziente Algorithmen für das Auffinden kürzester Wege[6] bereitstellt.

Als problematisch im Umgang mit den Graphen hat sich das Arbeiten mit Templates und Iteratoren herausgestellt, da diese Vorgehensweise einigen Teilnehmerinnnen der PG fremd war. Die Wegfindung selbst war, wie von der Boost-Library versprochen, ein einfacher Aufruf; die Definitionen, die dafür jedoch notwendig waren, waren aufwendiger zu erstellen.

[6] namentlich der Algorithmus von Dijkstra [Dij59] und der A*-Algorithmus [HNR68]

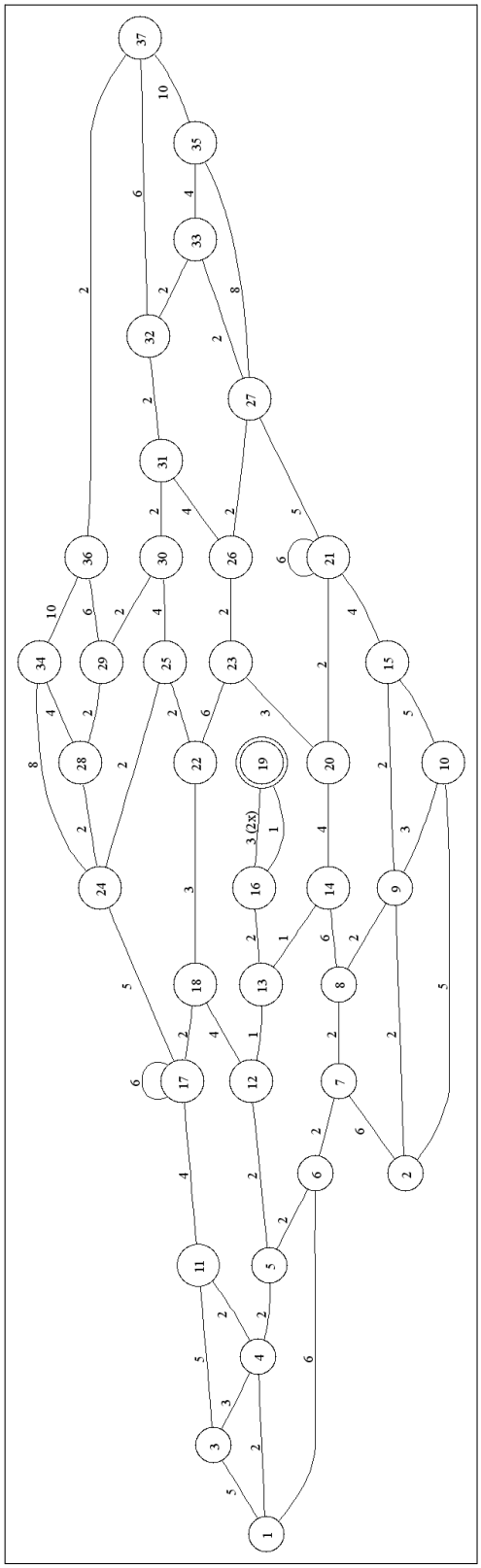


Abbildung 8: Der zum beschriebenen Labyrinth gehörende Wegegraph mit Wegkosten in diskreten Kacheln an den Kanten.

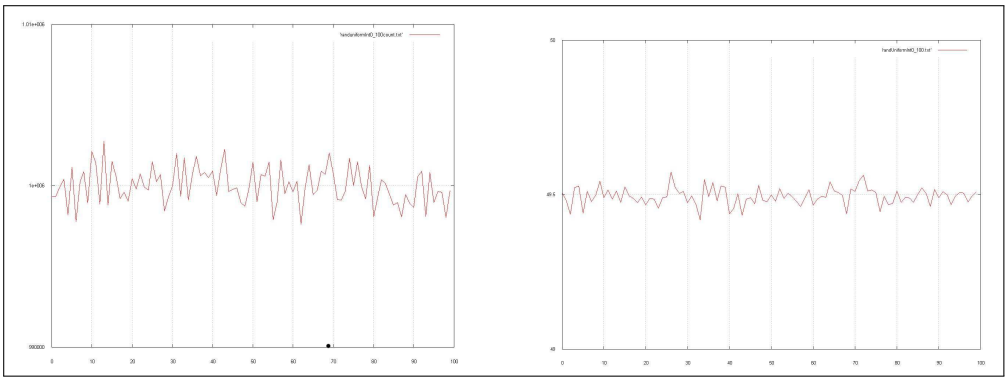


Abbildung 9: Aufruf des Zufallszahlengenerators für ganzzahlige Werte im Intervall $[0, 99]$ links: absolute Anzahl der einzelnen Werte von 0 bis 99 bei 100 Millionen Testaufrufen, wobei 1 Million der erwartete Mittelwert ist; rechts: Mittelwert von 100 verschiedenen Durchläufen mit jeweils 1 Million Zufallszahlen, wobei der erwartete Mittelwert $49,5$ ist.

3.5 Tests

In der Projekt-Planungsphase wurden Klassentests mit CppUnit [Rob00] als Aufgaben festgelegt. Nach der Umsetzung der ersten Tests zeigte sich aber schnell, dass klassische Unit Tests, in denen jede einzelne Klasse auf ihre gesamte Funktionalität unabhängig von allen anderen Klassen getestet wird, bei einem Spiel nicht ohne Weiteres möglich sind. Der Grund dafür ist, dass zum Test jeweils das gesamte GameInfo-Objekt mit der gewünschten Spielsituation nötig und die Erstellung dieser Momentaufnahmen extrem zeitaufwendig ist.

Ausführlich getestet wurde hingegen der Zufallsgenerator, da dessen korrekte Arbeitsweise grundlegende Voraussetzung für die Methoden der CI ist. Die erste zu testende Funktion war uniformInt(x, y), die eine ganzzahlige Zufallszahl aus dem Intervall $[x, y]$ liefert. Hier haben wir die gleichmäßige Verteilung über das gesamte Intervall sowie den erwarteten Mittelwert erfolgreich getestet (Abbildung 9 für das Intervall $[0, 99]$ und Abbildung 10 für das Intervall $[-10, 10]$). Grundlage war jeweils eine Anzahl von 1 Million Aufrufen, die 100-mal unabhängig voneinander ausgeführt wurden. Abschließend wurden die Randfälle getestet, mit folgenden Ergebnissen:

- uniformInt$(3, 4)$ liefert korrekte Ergebnisse
- uniformInt$(10, 10)$ liefert immer die Zahl 10
- uniformInt$(4, 3)$ wirft eine Fehlermeldung

Auch die Variante mit einer übergebenen Variable (uniformInt(x)) lieferte die gewünschten ganzzahligen Zufallszahlen aus dem Intervall $[0, x]$. Für die zweite zu testende Funktion uniformReal(), die reelle Zufallszahlen aus dem Intervall $[0, 1]$ liefern soll, haben wir sogenannte Runs-Tests durchgeführt, bei denen man die Länge aller aufsteigenden bzw. absteigenden Folgen der Zufallszahlen misst. Hierbei sollte die Anzahl der Folgen gleicher Länge auch etwa gleich häufig vorkommen. Wie in der folgenden Tabelle 1 ersichtlich ist, sind die Folgen bei einer Testsequenz von 1 Million Werten im gewünschten Bereich.

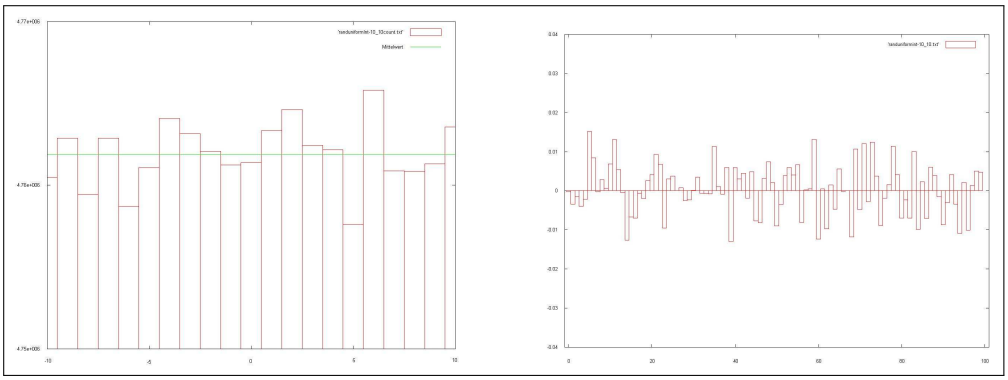


Abbildung 10: Aufruf des Zufallszahlengenerators für ganzzahlige Werte im Intervall $[-10, 10]$ links: absolute Anzahl der einzelnen Werte von -10 bis 10 bei 100 Millionen Testaufrufen, wobei $4,762$ Millionen (100 Millionen /21) der erwartete Mittelwert ist; rechts: Mittelwert von 100 verschiedenen Durchläufen mit jeweils 1 Million Zufallszahlen, wobei der erwartete Mittelwert 0 ist.

Wir haben auf explizite Tests der Varianten uniformReal(x) für das Intervall $[0, x]$ und uniformReal(x,y) für das Intervall $[x, y]$ verzichtet, da es sich bei beiden Funktionen nicht um eine neue Zufallszahlengenerierung handelt, sondern nur um eine Verschiebung der Grenzen durch Skalierung.

Tabelle 1: Runs-Test bei einer Folge von einer Million Werten

Länge	Anzahl aufsteigend	Anzahl absteigend
2	208325	208284
3	91850	92019
4	26362	26317
5	5623	5631
6	1083	1022
7	155	135
8	25	16
9	4	2

4 Gegner auf Basis deterministischer und randomisierter Strategien

Deterministische Computerspielgegner sind durch ihre Unfähigkeit zu lernen noch mehr abhängig von einer konkreten Situation als solche, die aus CI-Methoden hervorgegangen sind, da diese auf abstrahierten Umgebungsinformationen arbeiten. Im Falle der deterministischen Pacman-Geister ist diese Situation das Labyrinth, in dem das Spiel stattfindet. Um dies für die zu beschreibenden Geister (und das zu erwartende Spiel) festzulegen, wurde das klassische Pacman-Labyrinth von Namco in Njam nachgebaut (Abbildung 3). Der Tunnel, der den Spielfiguren erlaubt, schnell von einer Seite des Spielfeldes zur anderen zu gelangen, wurde durch Teleportfelder ersetzt, die allerdings nur von Pacman und nicht von den Geistern benutzt werden können.

Im Gegensatz zu den Strategien der Computational Intelligence, deren Aufgabe es ist, Geister zu erschaffen, die ihre Strategie selbst erlernen, lag die Herausforderung in der Generierung von deterministisch/randomisierten Strategien (im Folgenden kurz: dr-Strategien genannt) darin, Regeln und Verhalten für die Geister zu generieren, die weder zu gut noch zu einfach sein durften (Vgl. Abschnitt 7.1 zur Gewinnung des Spielspaßmaßes). Der Einsatz von globaler Sicht und Kommunikationsfähigkeit würde dazu führen, dass der Spieler in kürzester Zeit gefangen werden würde und keine Chance hätte, das Spiel siegreich zu beenden, weshalb beide Fähigkeiten nicht in die Geister integriert wurden. Andererseits sind auch zu leichte Gegner dem Spielspaß hinderlich und somit ebenso unerwünscht.

Die dr-Strategien sollten die Möglichkeit geben, das Verhalten der Geister mit Strategien auf Basis Neuronaler Netze (Abschnitt 5) und evolutionärer Allgorithmen (Abschnitt 6) dzu testen. Daher wurden die Geister „Randegho“ und „Firou“ (Abschnitt 4.1) unter dem Gedanken, möglichst leichte Gegner zu erstellen, geschrieben. Um einige Geister zu erstellen, die etwas mehr den Eindruck von intelligentem Verhalten erwecken, werden die Geister „Old Shimer“, „Darmok“ und „Puck“ (Abschnitt 4.1) durch eine kurze Geschichte über ihre Stärken und Schwächen beschrieben, so dass sie ausgewogene Gegner darstellen.

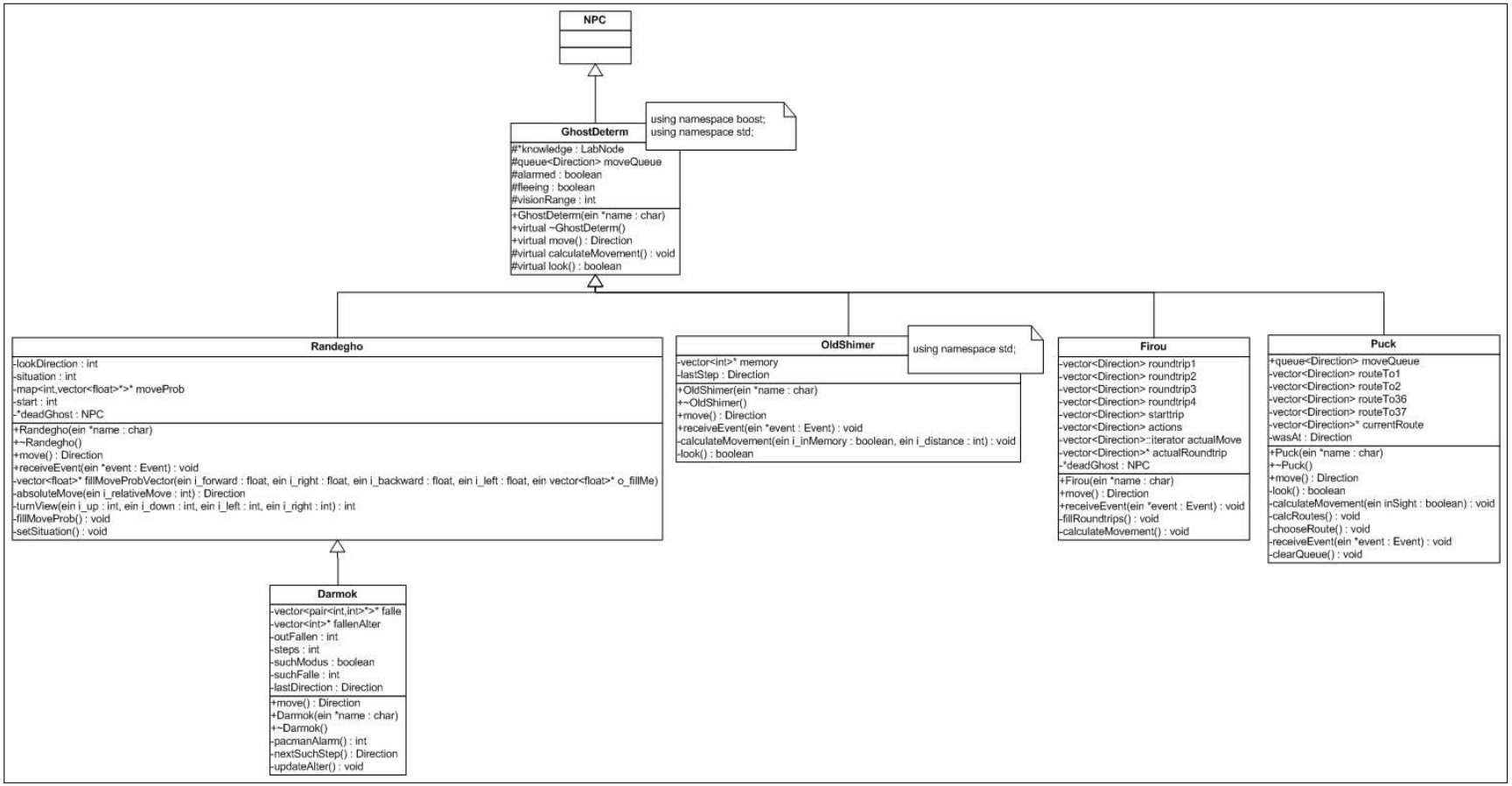


Abbildung 11: Das Klassendiagramm der determinsitisch-randomisierten Geisterstrategien zeigt die Klassen der Geister mit ihren Variablen, Methoden und den jeweiligen Sichtbarkeiten sowie die Abhängigkeiten der Klassen voneinander. NPC ist eine Klasse der Njam-Schnittstelle (Abschnitt 3.3) und nur als Platzhalter eingefügt, die genaue Beschreibung ist diesem Abschnitt und der Abbildung 6 zu entnehmen.

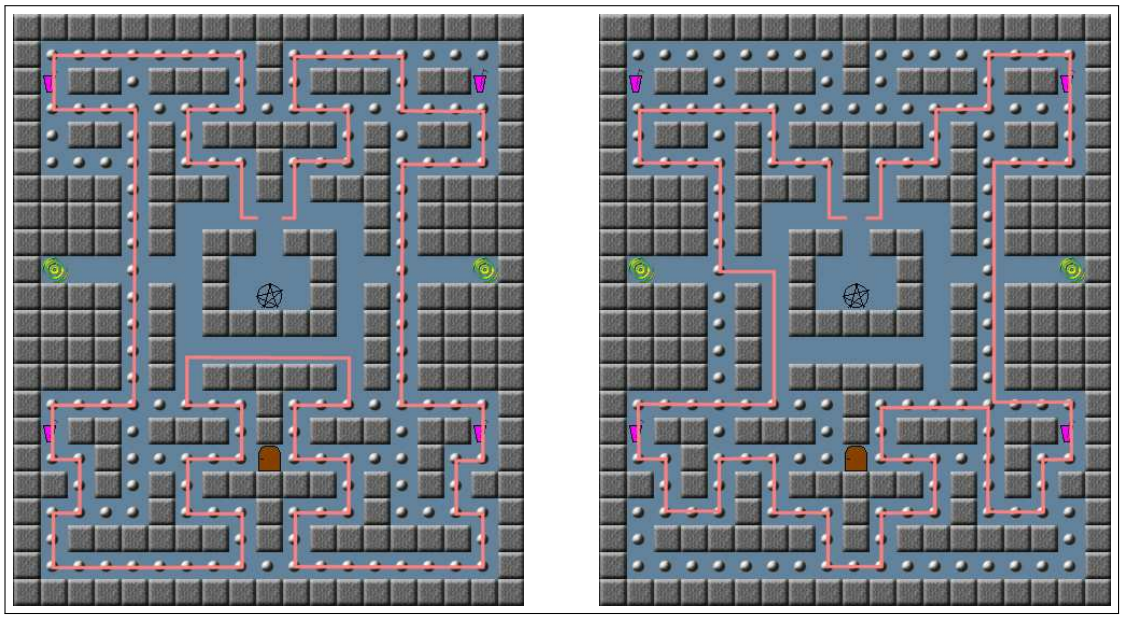


Abbildung 12: Routen, die Firou im Labyrinth zurücklegt, beide Routen können in beide Richtungen abgelaufen werden.

4.1 Deterministisch-randomisierte Gegnertypen

Wie eingangs beschrieben, entwickelten wir unterschiedliche auf dr-Strategien beruhende Gegner. Um das Rad nicht jedes Mal neu zu erfinden, wurde, wie aus dem Klassendiagramm (Abbildung 11) ersichtlich wird, eine dr-Oberklasse *GhostDeterm* als abgeleitete Klasse von NPC (siehe auch Abschnitt 3.3 beziehungsweise Abbildung 6) erstellt, die die von allen dr-Strategien benötigten Methoden enthält. Die einzelnen Implementierungen wurden dann von dieser Klasse abgeleitet.

Randegho Randegho (Akronym von **Ran**dom **de**terministic **gho**st) ist ein sich rein zufällig bewegender Geist. Für seine Bewegungen wird eine gleichverteilte Zufallszahl $[0, 1]$ gezogen, seine tatsächliche Bewegung bestimmt er durch die ihn umgebenden Wände und einen zur aktuellen Wandkonfiguration gehörenden Richtungsvektor $\vec{v}$ mit den Wahrscheinlichkeiten $p_n | n \in 1, 2, 3, 4$, $\vec{v} = (p_1, p_2, p_3, p_4)$ und $\sum_{i=1}^{4} p_i = 1$. Diese Wahrscheinlichkeiten sind so gewählt, dass Randegho, sofern dies der ihn umgebenden Wände wegen möglich ist, mit höherer Wahrscheinlichkeit die Richtung beibehält als abzubiegen und mit höherer Wahrscheinlichkeit abbiegt als umzudrehen.

Firou Der Geist, der immer weiß, wo er lang will. Firou (Akronym von **Fi**xed **rou**te ghost) wählt eines von vier festen Bewegungsmustern, das ihn vom Startpunkt aus einmal durch das ganze Labyrinth führt. Erreicht er wieder den Startpunkt, wird wieder eines der vier Muster gewählt. Firou läuft deterministisch und reagiert auch nicht auf Pacman. Die Routen können in Abbildung 12 eingesehen werden.

Old Shimer Old Shimer (pseudeo-anglifizierte Version der morbus alzheimer (Alzheimer-Krankheit, [Wik07a]), humoristisch bezogen auf die sich im frühen Stadium der Erkrankung ausbildende Vergesslichkeit) ist ein recht betagter Geist und kann sich nicht mehr viel merken, dafür bringt er aber eine recht große Erfahrung und viel Einblick in das Geschehen im Labyrinth mit. Er hat vollständige Information über denjenigen Teil des Spielfeldes, an den er sich erinnern kann. Die Sichtweite von Old-Shimer deckt gerade Sichtlinien ausgehend von seiner Position bis zu fünf Felder weit (Old Shimer ist ein wenig kurzsichtig) sowie den ihm bekannten Teil des Labyrinths ab.

Algorithmus: Um die abstrakte Idee umzusetzen, arbeitet Old Shimer auf dem im GameInfo Objekt verankerten Graphen. Dabei wird in jedem Knoten entschieden, welcher Knoten als nächstes Ziel von Old Shimer angesteuert werden soll. Jeder besuchte Knoten wird dabei in seinem Gedächtnis gespeichert, welches über eine maximale Größe von zehn Knoten verfügt. Dabei muss nun der Fall unterschieden werden, ob Old Shimer Pacman sieht, oder nicht.

Ist Pacman nicht in Sicht, schlägt Old Shimer immer den Weg zu einem Knoten ein, der noch nicht in seinem Gedächtnis gespeichert ist. Stehen mehrere Knoten zur Auswahl, so wird zwischen diesen zufällig ausgewählt. Sind alle Knoten bereits bekannt, so wird zwischen allen diesen Knoten zufällig ausgewählt, dies bedeutet aber, dass sich Old Shimer diesen Knoten zweimal „merkt“ und sich sein Gedächtnis daher effektiv um einen Knoten verringert. Durch diesen Kunstgriff ist gewährleistet, dass nach spätestens acht Zügen Old Shimer wieder anfängt, die Karte zu erforschen, wenn er zum Beispiel ins Geisterhaus gelaufen ist und für diesen Fall keine Sonderprogrammierung benötigt wird.

Ist Pacman in Sicht, verfolgt Old Shimer ihn, sofern nicht die Powerpille aktiv ist. Da "in Sicht" hierbei sowohl die direkte Sichtlinie als auch den bekannten Teil des Labyrinths bedeutet, wird der Algorithmus von Dijkstra verwendet, um den kürzesten Weg zu Pacman zu ermitteln. Ist die Powerpille aktiv, so wählt Old Shimer zufällig einen möglichen Weg, der nicht der empfohlenen Bewegung durch den Dijkstra Algorithmus entspricht.

Schwierig in der Umsetzung waren die Richtungswechsel von Old Shimer auf den Kanten des Graphen (Abschnitt 3.4), da die Bewegungen des Geistes aufgrund der Verwendung der gespeicherten Bewegungsfolgen umgerechnet werden mussten, damit der Geist zum Ausgangsknoten zurückfindet.

Puck Puck (da, wie im Folgenden beschrieben, dieser Geist ein gewisses „Heimweh“ zum Geisterhaus verspürt, wird er als Hausgeist (englisch: „Puck“) bezeichnet) bewegt sich auf den Wegen, die ihm vertraut sind, und scheut jedes Abenteuer. Seine bekannten Routen führen ihn zu allen vier Ecken der Welt. Sollte es passieren, dass er von seinem Weg abkommt, findet und benutzt er immer den Weg nach Hause.

Dabei kann ihn nur Pacman von seinem Weg abbringen: sieht der heimische Geist den schutzlosen Pacman, rennt er ihm nach, bis er ihn gefangen oder aus den Augen verloren hat. Vergisst er im Eifer des Gefechts, wo er ist, rennt er schnellstmöglich nach Hause und setzt seine Weltreise neu an. Sollte der Pacman ihm über den Weg laufen, wenn dieser unter dem Einfluss der Kraftpille steht, wählt der Geist den kürzesten Weg zum Haus und flieht dorthin.

Algorithmus: Die Idee zum Verhalten von Puck wurde folgendermaßen umgesetzt: Nach der Generierung des Geistes bewegt er sich auf direktem Weg aus dem Haus. Diese Schritte

sind festgelegt und sorgen dafür, dass der Geist nicht im Haus herum irrt. Der Knoten Nummer 13 ist der Startknoten zu jeder Route, die dieser Geist wählen kann. Die Routen führen, wie in der obigen Beschreibung angegeben, zu den vier den äußeren Ecken des Labyrinthes am nähesten liegenden Knoten $1, 2, 36$ und 37 (gemäß der Nummerierung im Graphen, Abschnitt 3.4). Vom Knoten 13 aus wird für die Bewegung eine dieser vier Routen zufällig mit Hilfe des Zufallszahlengenerators gewählt. Eine Route wird durch einen Vektor dargestellt, in dem die Richtungen (vom Typ Direction) UP, DOWN, LEFT oder RIGHT, die zum Zielknoten führen, festgehalten werden.

Im Normalmodus, wenn also Pacman nicht in Sicht ist, folgt Puck der gewählten Route bis zum Zielknoten. Am Zielknoten angekommen, wird der kürzeste Weg zurück zum Knoten 13 mit der in der Graph-Klasse verfügbaren Methode zur Berechnung kürzester Wege berechnet. Im Anschluss wird erneut eine der vier Routen gewählt und der Ablauf wiederholt sich.

Mit jedem Aufruf der move-Methode wird überprüft, ob Pacman in Sicht ist. Sollte dies der Fall sein, unterscheiden sich zwei Fälle im Bewegungsalgorithmus von Puck. Zum einen kann der Pacman eine Powerpille gesammelt haben. In diesem Fall ist der Geist verwundbar und soll vor Pacman fliehen. Der Fluchtmodus vom Puck sieht vor, dass er sich auf dem schnellsten Weg zurück zum Knoten 13 begibt. Daher wird der kürzeste Weg zum Haus berechnet und Puck folgt dieser Route zurück. Sollte sich im Fluchtmodus Pacman im Gang vor dem Haus bewegen, flieht Puck zurück ins Haus (Knoten 19) und prüft mit jedem folgenden move-Aufruf, ob Pacman immer noch im Gang ist und wenn ja, ob die Powerpille wirksam ist. Je nach Ausgang dieser Prüfung bleibt er im Fluchtmodus und geht zurück ins Haus oder aber wechselt in den Jagd- bzw. Normalmodus. Der Jagdmodus ist der zweite Sonderfall im Bewegungsablauf. Der Geist wechselt in diesen Modus, wenn er Pacman ohne Powerpille gesichtet hat. In diesem Fall wird erneut auf die Methode zur Berechnung kürzester Wege zurückgegriffen. Zur Berechnung des Weges werden die aktuellen Koordinaten des Geistes und die aktuellen Koordinaten des Pacman gewählt. Der Geist bewegt sich dann auf dem berechneten Pfad zu Pacman hin. Sollte sich der Pacman aus der Sichtweite des Geistes entfernt haben, wird der kürzeste Weg zum Knoten 13 von der aktuellen Position aus berechnet und der Geist kann von diesem Knoten aus erneut eine der vier Routen wählen. Falls Pacman jedoch in Sichtweite bleibt, aber sich von der ursprünglich berechneten kürzesten Route entfernt hat, wird einfach der kürzeste Pfad immer wieder berechnet.

Darmok Darmok (der Name ist entlehnt von einer mythische Jägerfigur des fiktiven Volkes der Tamarianer, beschrieben in der Fernsehsendung „Darmok“, der zweiten Folge der fünften Staffel der Science Fiction Fensehserie „Star-Trek Raumschiff Enterprise: Das nächste Jahrhundert“) ist ein schlauer Jäger. Er bewegt sich wie Randegho (Abschnitt 4.1) über das Spielfeld, legt dabei aber für den Spieler unsichtbare Fallen aus, die ihm, wenn Pacman sie betritt und damit auslöst, die Position des Spielers verraten, worauf er auf kürzestem Weg dorthin läuft. Außerdem verfolgt er Pacman, wenn er ihn sieht.

Algorithmus: Darmok bewegt sich zufällig und legt mit einer Wahrscheinlichkeit von 20% in jedem Schritt eine unsichtbare Falle in den Weg. Dabei speichert er die Koordinaten der Fallen in einer Tabelle ab. Insgesamt verfügt er über fünf Fallen. Jede dieser Fallen besitzt eine Lebensdauer von fünfzig Spielschritten, danach wird sie gelöscht und Darmok kann eine weitere Falle auslegen. Wenn sich Pacman auf ein Feld bewegt, auf der sich eine

Falle befindet, ruft der Geist die Koordinaten der betreffenden Falle auf, berechnet den kürzesten Weg zu ihr und bewegt sich dorthin, die Falle wird dann gelöscht. Jedes Mal, wenn eine Falle gelöscht wird, kann Darmok sie neu auslegen.

Falls der Pacman sich in Sichtweite des Geistes befindet, ruft der Geist den Shortest-Path-Algorithmus auf. Dieser liefert den kürzesten Weg zwischen der aktuellen Position des Geistes und des Pacmans. Durch diesen Schritt wird die Verfolgung Pacmans ausgelöst.

Für den Fall, dass der Pacman die Kraftpille geschluckt hat und nicht in Sichtweite ist, bewegt sich Darmok weiterhin zufällig. Ansonsten wird der kürzeste Weg zum Pacman bestimmt und die Ausgabe für die jeweiligen Bewegungen invertiert.

4.2 Deterministische Pacman-Implementierung

Beim Training der durch evolutionäre Algorithmen gesteuerten Geister stellte sich heraus, dass aufgezeichnete Bewegungen von Pacman sich hierzu nicht eigneten. Das Problem lag darin, dass ein Pacman, der nur eine Aufzeichnung eines menschlichen Spielers ablief, nicht vor den Geistern fliehen konnte. Somit bestand der Bedarf für einen intelligenten NPC für Pacman. Um den Geistern einen besseren Gegner zu bieten als statische Aufzeichnungen von Spielen, wurden einige einfache Regeln zur Steuerung von Pacman aufgestellt. Pacman bewertet an seiner aktuellen Position alle vier Richtungen und wählt die Richtung mit der höchsten Bewertung. Eine Richtung erhält einen Wert gemäß Tabelle 2.

Die Paare aus Bewertung und Richtung werden in eine Prioritätswarteschlange eingestellt und dann die höchstbewertete Richtung gewählt. Dieses Vorgehen funktioniert sehr gut, solange noch Pellets in Sicht sind. Sobald die Karte relativ leer gefressen ist, wird es für Pacman schwierig, die verbleibenden Pellets zu finden, da er nicht über eine globale Sicht verfügt.

Tabelle 2: Richtungsbewertungen für durch dr-Strategie gesteuerten Pacman. Die Richtung, in die sich Pacman vorher bewegt hat, erhält zusätzlich noch $+0,25$.

Wert	Bedingung
5.5	Pellets in Sicht und benachbartes Feld ist auch Pellet, Geist in Sicht Pacman hat Powerpille
5.0	Pellets in Sicht, Geist in Sicht, Pacman hat Powerpille
4.5	Pellets in Sicht und benachbartes Feld ist auch Pellet, kein Geist in Sicht
4.0	keine Pellets in Sicht, Geist in Sicht, Pacman hat Powerpille
4.0	Pellets sind in Sicht, kein Geist in Sicht
3.0	keine Pellets in Sicht, kein Geist in Sicht
2.5	Pellets in Sicht und benachbartes Feld ist auch Pellet, Geist in Sicht
2.0	Pellets in Sicht, Geist in Sicht
1.0	keine Pellets in Sicht, Geist in Sicht
0.0	Wand

5 Gegner auf Basis neuronaler Netze

Der Entwicklungsprozess der Teilgruppe neuronale Netze (NN) begann mit einer Planungsphase, in der zunächst Ideen für die spätere Umsetzung gesammelt wurden. Zu den Themen, die während der Planungsphase angedacht wurden, zählen unter anderem:

- die zu verwendende Datenstruktur der Netze,
- die zu verwendende Datenstruktur der Knoten und Kanten,
- die relevanten, beschreibenden Attribute eines vollständigen Netzes,
- die Möglichkeit zum Speichern und Laden von Netzen sowie
- die Implementierung eines Lernverfahrens, zum Beispiel Backpropagation (siehe 2.1.2).

Diese Punkte werden in den folgenden Abschnitten genauer betrachtet.

Datenstruktur für Netze Bei der Suche nach einer geeigneten Datenstruktur für die Repräsentation künstlicher neuronaler Netze (KNN) fiel die Wahl schnell auf die Verwendung von Graphen. Um ein hohes Maß an Performance gewährleisten zu können und gleichzeitig die knappe Zeit optimal auszunutzen, wurde auf die Entwicklung einer eigenen Graphenstruktur verzichtet. Stattdessen entschloss sich die Gruppe, die bereits vorhandene Graphen-Bibliothek der Boost C++ Libraries [Boo07] zu verwenden. Eine mehrjährige Entwicklungszeit, Qualitätskontrolle sowie der Template-basierte Ansatz sprachen für die Entscheidung. So konnte die geforderte Stabilität und Performance garantiert werden. Die zusätzlich zur Verfügung stehenden Methoden und Algorithmen erleichterten zudem die späteren Arbeiten der Implementierungsphase. Dennoch sei an dieser Stelle kurz erwähnt, dass die Benutzung der Boost C++ Libraries zu Beginn einige Probleme bereitet hat, da die intensive Einarbeitung in die Referenz innerhalb der ersten Wochen die Implementierungsarbeiten verzögerten.

Datenstruktur für Neuronen und Kanten Aufgrund des Template-basierten Ansatzes des Boost-Graphen, konnte die Entwicklung einer geeigneten Datenstruktur zur Repräsentation der Knoten (beziehungsweise Neuronen) und Kanten zunächst parallel und unabhängig voneinander verlaufen. Schließlich wurde die generische Graphenstruktur um die von der Teilgruppe NN entwickelten Eigenschaften für KNN erweitert. Beide Objekttypen wurden bereits im Vorfeld so abstrakt wie möglich beschrieben, um ein hohes Maß an Flexibilität zu gewähren. Außerdem wurde die Möglichkeit zur Verwaltung beliebiger KNN Konfigurationen durch eine spätere Modifikation des PacNN Moduls offen gelassen.

Interface Trotz der großen Auswahl an Methoden und Algorithmen der Graphen-Bibliothek war es nötig ein geeignetes Interface für die Verwendung des PacNN Moduls zu implementieren. Dies beinhaltete in erster Linie Methoden zur Bereitstellung aller möglichen Kombinationen von Eingangsdaten. Zusätzlich wurde die Idee eines Kommandozeilenmodus, welcher unabhängig von der Testumgebung Njam arbeitet, umgesetzt (Abschnitt 5.2.3).

Speichern und Laden Aufgrund des eingesetzten Offline-Lernverfahrens war es nötig, trainerte Netze für den späteren Einsatz zunächst abzuspeichern. Zudem mussten die verwendeten Trainingsmuster effizient verarbeitet werden können. Neben der Implementierung der Methoden zum Speichern und Rekonstruieren gültiger Netze bestand die Aufgabe darin, ein geeignetes Dateiformat zur Repräsentation von Netzen und Trainingsmustern zu entwickeln (Abschnitte 5.2.1 und 5.2.2).

Implementierung des Lernverfahrens und erstes Testen Als Lernverfahren wurde der Backpropagation-Algorithmus implementiert, welcher in Abschnitt 2.1.2 beschrieben wird. Es wurde versucht, diesen in mehreren verschiedenen Varianten zu implementieren. Es wurde beispielsweise Backpropagation mit Momentum-Funktion und Threshold-Update implementiert. Dies führte jedoch nicht zu den gewünschten Ergebnissen, so dass dieser Ansatz verworfen werden musste und zunächst eine vereinfachte Version des Backpropagation-Algorithmus entwickelt wurde. Diese Version (siehe auch 2.1.2) führte letztendlich zu den gewünschten Ergebnissen.

MrNeuro: Schnittstelle Njam Da das KNN entkoppelt vom konkreten Spiel Pacman und der Implementierung NJam entwickelt wurde, war eine Schnittstelle notwendig, die die Daten aus dem Spiel in Eingaben für das neuronale Netz sowie die Ausgaben des neuronalen Netzes in Aktionen für das Spiel umrechnet.

Das neuronale Netz verfügt über vier Input Neuronen, welche folgende Informationen kodieren:

- die letzte Entscheidung des Netzes,
- durch Wände geblockte Richtungen,
- Informationen ob Pacman in Sichtweite sowie
- Informationen zum Status der Powerpille.

Die Umrechnung der aktuellen Spielsituation in gültige Eingabewerte für die vier Input-Neuronen erfolgt gemäß der Codierung in Tabelle 3. Die Ausgabe des neuronalen Netzes in Form eines Vektors von Wahrscheinlichkeiten für jede Richtung wird von MrNeuro in eine konkrete Richtung umgerechnet, indem zunächst alle nicht möglichen Züge (nicht in Wände laufen sowie nicht zurück ins Haus) auf eine Wahrscheinlichkeit von Null gesetzt werden. Anschließend wird die tatsächliche Richtung mit einer Wahrscheinlichkeit entsprechend der Empfehlung des neuronalen Netzes selektiert.

5.1 Evaluation: Testphase

Nachdem die grundlegenden Strukturen implementiert waren, wurde auf dieser Basis ein einfaches neuronales Netz erzeugt, um die Funktionsweise des Algorithmus zu verifizieren.

Das erste Netz besteht aus drei Input-Neuronen in der Eingabeschicht und vier Neuronen in der Ausgabeschicht. Die versteckte Schicht umfasst zu Beginn nur vier Neuronen. Ein

Tabelle 3: Codierung der Input Neuronen

Wand-Sensor		Pacman-Sensor		Pacman Status		Ghost Direction	
top	-2.0	right-close	2.0	power	1.0	up	0.5
left	-1.0	right-medium	1.5	midpower	0.5	right	1.0
right	1.0	right-far	1.0	nopower	0.0	down	-0.5
bottom	2.0	left-close	-2.0			left	-1.0
top-left	-1.5	left-medium	-1.5				
bottom-right	1.5	left-far	-1.0				
top-bottom	0.25	top-close	4.0				
left-right	-0.25	top-medium	3.5				
top-right	0.5	top-far	3.0				
bottom-left	-0.5	bottom-close	-4.0				
no	0.0	bottom-medium	-3.5				
		bottom-far	-3.0				
		not in sight	0.0				

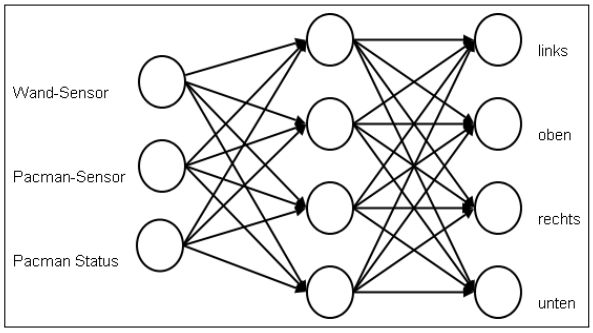


Abbildung 13: Das erste testbare neuronale Netz

guter Richtwert für die Anzahl der Neuronen in der versteckten Schicht sollte nach [BS04] Gleichung (6) liefern:

$$\sqrt{\#Inputneuronen * \#Ausgangsneuronen} = \#versteckteNeuronen \tag{6}$$

Die Anzahl der Neuronen in der versteckten Schicht wurde zu Beginn auf drei gesetzt, da diese grundsätzlich möglichst klein gehalten werden sollte, um die Komplexität und die Laufzeit zur Anpassung der Kantengewichte durch den Lernalgorithmus gering zu halten. Innerhalb der versteckten Schicht wird keine weitere Schicht benötigt, da nach [Eng02] ein neuronales Netz bestehend aus drei Schichten jede Funktion berechnen kann. Die einzelnen Schichten sind untereinander vollständig vernetzt, so dass Abbildung 13 das erste testbare neuronale Netz darstellt. Jedes Eingangsneuron repräsentiert eine für die Intelligenz der Geister relevante Beschreibung der Umgebung. Die Ausgabeneuronen repräsentieren die vier möglichen Bewegungen der Geister. Für jeden Geist wird ein eigenes neuronales Netz verwaltet.

Die Eingabe *Wand-Sensor* nimmt die Umgebung der Geister wahr, so dass die Position der Wände bestimmt werden kann. Die zweite Eingabe *Pacman-Sensor* teilt jedem Geist mit, ob und in welcher Richtung Pacman gesichtet wurde. Zusätzlich wird die Entfernung zwischen Geist und Pacman berücksichtigt. Die dritte Eingabe *Pacman-Status* codiert den

Zustand von Pacman, ob er eine Power Pille gefressen hat oder nicht. Bevor die Wirkung der Power Pille komplett aussetzt, wird dieses den Geistern über die Eingabe mitgeteilt.

Insgesamt konnten bei obiger Kodierung 263 unterschiedliche Situationen im Spiel eintreten, was auch der Anzahl aller möglichen Testpatterns entspricht. Von diesen sollte das Netz möglichst viele richtig klassifizieren, damit die Geister ein „intelligentes“ Verhalten zeigen. Jedoch haben wir uns zum Trainieren des Netzes zu Beginn auf die unserer Meinung nach wichtigsten 92 Pattern beschränkt und diese für das Training verwendet. Diese 92 Pattern setzten sich aus den Spielsituationen zusammen, bei denen die Geister nur die Möglichkeit haben, in eine festgelegte Richtung zu gehen. In den anderen drei Richtungen befinden sich also Wände, so dass die Wahl einer dieser Richtungen unnatürlich und unintelligent wirkt. Falls ein Geist mit einem so trainierten Netz auf eine ihm unbekannte Spielsituation trifft, sollte diese aufgrund der Fähigkeit zur Generalisierung von neuronalen Netzen richtig behandelt werden.

Wann wird jedoch beim Testen ein Pattern als „*richtig*“ klassifiziert? Eine selbstentwickelte *Hit*-Funktion ermöglicht das Verifizieren des Netzes. Hierbei wurden die berechneten Ausgabewerte des Netzes mit den erwünschten Werten[7] verglichen. Die einzelnen Abweichungen der berechneten Werte von den erwünschten Werten wurden aufsummiert und mit einem Toleranzfehler von $0,15$ verglichen. Dieser Fehlerwert wurde experimentell ermittelt und von den Mitgliedern der Teilgruppe NN als Toleranzfehler festgesetzt. Bei größere Abweichungen zeigten die Geister nicht das gewünschte Verhalten. War der Fehler kleiner, so hat das Netz dieses Pattern richtig erkannt.

Leider war es uns nicht möglich das oben dargestellte Netz gut zu trainieren. Die meisten Trainingspatterns wurden nicht als richtig verifiziert, was sich auch nach mehr als 50000 Lernzyklen nicht mehr veränderte. Eine mögliche Erklärung dieses Problems findet sich im Verlauf der zu optimierenden Funktion wieder. Dieser wies eine Vielzahl von lokalen Minima und flachen Plateaus auf. Da Backpropagation ein Gradientenabstiegsverfahren ist (siehe auch Abschnitt 2.1.2), kann das Verfahren in unterschiedlichen Minima enden, wenn man die Gewichtsvektoren beim Start des Verfahrens unterschiedlich wählt. Die Abbildung 14 zeigt den Fall, bei dem eine falsche Startinitialisierung der Gewichtsvektoren zu einem Abbruch des Verfahrens in einem lokalen Minimum führt.

Eine gute Randomisierung der Gewichtsvektoren beim Start des Verfahrens war die Lösung des Problems. Damit war jedoch das Problem der flachen Plateaus nicht gelöst. In flachen Plateaus ist der Gradient sehr klein und das Verfahren stagniert nahezu. Dies ist in Abbildung 15 dargestellt.

Da die Gewichtsveränderung in flachen Plateaus bei nur vier Neuronen in der versteckten Schicht sehr gering ist, lag es nahe die Anzahl der Neuronen in der versteckten Schicht zu erhöhen. In Kombination mit einer Erhöhung der Lernrate konnten Plateaus bei der Suche nach dem Optimum überwunden werden. Letztlich wurden viele verschiedene Netze mit einer unterschiedlichen Anzahl von Neuronen erstellt und trainiert. Der erste wirkliche Erfolg stellte sich bei einem Netz mit 23 Neuronen in der versteckten Schicht ein. Das Netz war in der Lage 85 der 92 Trainingspatterns zu erlernen und die restlichen möglichen Pattern, also den möglichen Spielsituationen, korrekt zu erkennen.

Im praktischen Einsatz zeigten sich jedoch weitere Schwierigkeiten, da die Geister mit dem so trainierten Netz dazu neigten, auf geraden Strecken häufig die Richtung zu wech-

[7] die zu den Eingaben korrespondierenden Ausgabewerte sind bekannt

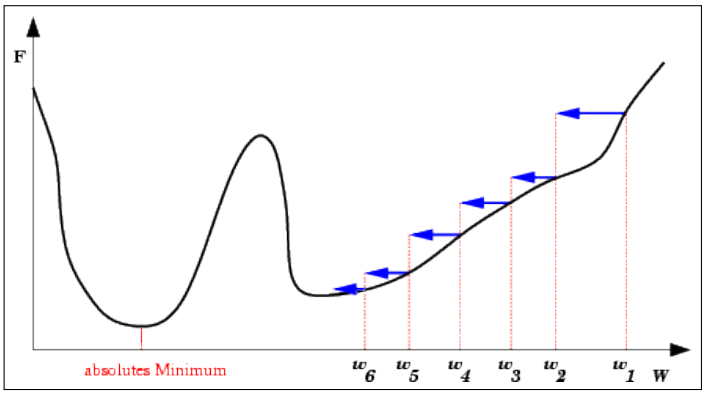


Abbildung 14: Darstellung des Problems von lokalen Minima bei Backpropagation. Es wird die Entwicklung der Fehlerfunktion gezeigt. Dabei fällt auf, dass das Lernverfahren in einem lokalen Minimum endet, anstatt das globale Minimum zu erreichen.

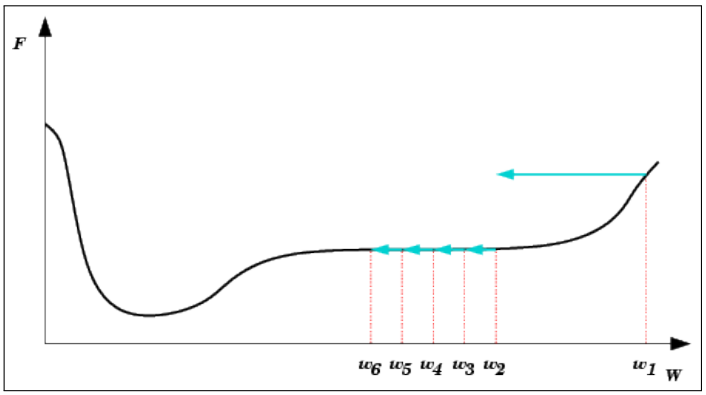


Abbildung 15: Darstellung des Problems eines flachen Plateaus bei Backpropagation. Es wird deutlich, dass sich die Fehlerfunktion sehr langsam dem globalen Minimum nähert. Es kann sogar vorkommen, dass dieses Minimum nie erreicht wird.

seln (im Folgenden als *Togglen* bezeichnet). Um diesem Problem zu begegnen wurde ein weiteres Eingangsneuron, welches die Richtung für die sich der Geist im vorherigen Zug entschieden hatte, eingefügt. Die bereits bestehenden 263 Patterns mussten überarbeitet und um weitere Patterns ergänzt werden, so dass nun 308 verschiedene Situationen im Spiel auftreten konnten. Die mit diesen Patterns trainierten Netze zeigten in der Praxis ein deutlich besseres Verhalten der Geister als die Vorgängerversionen. Das Togglen konnte minimiert werden. Das Netz bestehend aus vier Input-Neuronen, 23 Neuronen in der versteckten Schicht und vier Ausgabeneuronen wurde im Gegensatz zum ersten Ansatz mit zufällig gewählten 92 Patterns so trainiert, dass fast alle möglichen Spielsituationen richtig klassifiziert wurden.

Tabelle 4: Parameter die während des Trainings zum Einsatz kamen

cycles	$10000 - 50000$
lrate	$0.1 - 0.7$
seed	default

5.2 Referenz

Die nachfolgenden Abschnitte enthalten Informationen bezüglich der entwickelten Speicherformate und Hilfsprogramme des PacNN Moduls.

5.2.1 Dateiformat Netzspezifikation

Die Fähigkeit, Netze im Vorfeld offline zu trainieren, um sie später jederzeit einsetzen zu können, setzt die Möglichkeiten des Speicherns und Ladens voraus. Daher bestand eine Aufgabe der Teilgruppe NN darin, ein geeignetes Dateiformat für Netzspezifikationen zu entwickeln, sowie die darauf basierenden Methoden Speichern und Laden zu implementieren.

Die Wahl des Dateiformats ergab sich schnell aus der Anwendung der Graphen-Bibliothek. Diese bietet bereits Methoden zum Speichern und Laden kompletter Graphenstrukturen. Das native Format verarbeitet Daten in ASCII-Kodierung und wurde um die, für die Rekonstruktion relevanten Attribute der Netzspezifikation erweitert.

Die Syntaxelemente des Dateiformats werden im Folgenden kurz erläutert.

threshold entspricht dem Schwellenwert eines Neurons; Typ `double`

layer id entspricht der internen Schichtennummer, auf der sich das Neuron befindet; Typ `int`.

source nr entspricht der internen Knotennummer des Elterknotens einer Kante; Typ `int`

target nr entspricht der internen Knotennummer des Kindknotens einer Kante; Typ `int`

weight entspricht dem Kantengewicht einer Kante; Typ `double`

Listing 2 zeigt Syntax und Aufbau einer Netzspezifikation. Listing 3 zeigt das Beispiel einer gültigen Netzspezifikation.

5.2.2 Dateiformat Testmusterspezifikation

Eine weitere Aufgabe der Teilgruppe NN war es, ein geeignetes Speicherformat für Testmusterdaten zu entwickeln. Dies war nötig, um ein effizientes Training zu ermöglichen. So konnten Netze bereits im Vorfeld, also offline, auf beliebig vielen Trainingsdaten trainiert und optimiert werden.

```
# lines starting with # are ignored
# v indicates vertix description block
# <threshold> : double
# <layer id>  : int
v
<threshold> <layer id>
# e indicates edge description block
# <source nr> : int
# <target nr> : int
# <weight>    : double
e
<source nr> <target nr> <weight>
# eof
```

Listing 2: Syntax und Aufbau einer Netzspezifikation. Zeilen, welche mit # beginnen, sind optional zu verwenden. Die geklammerten Argumente repräsentieren den jeweils geforderten Datentyp

```
v
0.259922 0
0.490839 0
0.667639 1
0.619461 1
e
0 2  7.0821
0 3  8.18611
1 2  6.53198
1 3  -2.6788
```

Listing 3: Beispiel einer gültigen Netzspezifikation. In diesem Fall wird ein Netz auf vier Neuronen, je zwei auf Input und Output-Layer, beschrieben. Die Neuronen sind schichtweise vollständig vernetzt.

Auch hier fiel die Wahl auf ein einfaches, gut leserliches Dateiformat in ASCII-Kodierung, um das Bearbeiten der Datensätze zu erleichtern. Die Testmuster werden zeilenweise beschrieben und bestehen aus je zwei n-Tupeln von Eingangswerten und Ausgangswerten, welche durch das Zeichen # getrennt werden.

Listing 4 zeigt Syntax und Aufbau einer Testmusterspezifikation. Listing 5 zeigt das Beispiel einer gültigen Testmusterspezifikation.

5.2.3 PacNN Kommandozeilenmodus

Neben der Möglichkeit PacNN mittels MrNeuro ins Njam einzubinden, besitzt das Modul PacNN einen Kommandozeilenmodus, der zunächst unabhängig vom eigentlichen Spiel verwendet werden kann. Dieser Modus wurde zum Beispiel für das Training und die Auswertung von Netzen genutzt.

```
# lines starting with # are ignored
# each line indicates on pattern
# <input>  : double
# <output> : double
# use # as delimiter
<input> [<input>]#<output> [<output>]
# eof
```

Listing 4: Syntax und Aufbau einer Testmusterspezifikation. Zeilen, welche mit # beginnen sind optional zu verwenden. Die geklammerten Argumente repräsentieren den jeweils geforderten Datentyp

```
-2.0 2.0 0.0 0.0#0.0 0.0 1.0 0.0
-2.0 2.0 0.5 0.0#0.5 0.0 0.0 0.5
-2.0 2.0 1.0 0.0#0.4 0.0 0.0 0.6
-2.0 1.5 0.0 0.0#0.0 0.0 1.0 0.0
```

Listing 5: Beispiel einer gültigen Testmusterspezifikation. Es werden vier Testmuster mit jeweils vier Eingabewerten und vier Ausgabewerten beschrieben. Eingabewerte und Ausgabewerte werden durch # getrennt.

Der Programmaufruf kann von der Kommandozeile aus über Parameter gesteuert werden und bietet verschiedene Ausführungsmodi. Eine genaue Auflistung der zur Verfügung stehenden Befehle, sowie Beispiele zur Benutzung können dem internen Hilfebildschirm entnommen werden.

5.2.4 PacTools

PacTools war zunächst als Sammlung verschiedener Hilfsfunktionen des Moduls PacNN angedacht. Aufgrund des Zeitmangels konnte jedoch lediglich die automatische Generierung von Netzspezifikationen fertiggestellt werden.

PacTools bietet also die Möglichkeit, auf einfache Weise gültige Netze generieren zu lassen. Dies erleichterte die Abläufe während der Trainingsphase, da in kurzer Zeit Netze getestet und Ergebnisse gesammelt werden konnten.

6 Gegner auf Basis evolutionärer Algorithmen

Da das Forschungsgebiet der evolutionären Algorithmen viele zur möglichen Steuerung von NPCs denkbare Ansätze zur Verfügung stellt, beispielsweise ($\mu + \lambda$)-EAs, Evolutionsstrategien, Genetische Programmierung oder Coevolutionäre Algorithmen, wurden drei verschiedene Ansätze von uns ausgewählt, um einen möglichst breiten Einblick in die Mächtigkeit evolutionärer Generierung von Spielstrategien zu erhalten. Die Ansätze EABBGhost und ProbEAGhost entstanden aus der Idee, einem Geist die Situation, in der er sich befindet, mitzuteilen und ihn dann eine deterministisch oder randomisiert gewählte Aktion ausführen zu lassen. Welche Aktion in dieser Situation am besten auszuführen sei, war von den beiden Geistern im Vorfeld zu erlernen. Der OnlineEA wurde von der in der Praxis oft eingesetzten evolutionären Steuerung industrieller Maschinen inspiriert.

6.1 Klassenstruktur

In der Phase, als Ideen für unterschiedliche von evolutionären Algorithmen gesteuerte NPCs entstanden sind, hat die EA-Gruppe entschieden ein möglichst abstraktes Framework für die Implementierungen zu schaffen. Die Klassenstruktur sollte so flexibel gehalten sein, dass verschiedene Ansätze realisiert werden können, aber insgesamt ein einheitlicher Klassenrahmen entsteht. Um eine gute Wiederverwendbarkeit im zweiten Projekt zu gewährleisten, wurde die Klassenstruktur möglichst spielunabhängig gestaltet.

Der Kern des Frameworks, dessen Klassendiagramm in Abbildung 16 einsehbar ist, bilden die beiden Klassen EAManager und EAController. Der EAManager wurde als Singletonklasse implementiert und besitzt statische Methoden zum Erzeugen von EA-gesteuerten NPCs. Ein Aufzählungstyp EAType beinhaltet alle implementierten Typen von Geistern. Die Factorymethoden erzeugen durch Angabe eines EATypes einen entsprechenden von einem evolutionären Algorithmus gesteuerten Geist. Während des Erzeugens eines Geistes sorgt der EAManager dafür, dass stets für jeden EAType ein EAController existiert.

Der EAController ist für die Verwaltung der Population der Individuen eines bestimmten EATypes verantwortlich. Neben den Klassen zur Verwaltung gibt es für die eigentliche Implementierung der Geister eine Basisklasse EANPC, welche von der Klasse NPC erbt. Die Klasse EANPC besitzt einen bestimmten EAType und beinhaltet ein Individuum. In der von der Klasse NPC geerbten Move-Methode wurden die unterschiedlichen EA-Strategien entwickelt. Um die Klasse EAController von der NJam-spezifischen Klasse EANPC zu entkoppeln, kennt der EAController nicht die EANPCs, sondern nur die Individuen. Die EANPCs hingegen können über den EAManager ihren entsprechenden EAController aufrufen.

Die Individuen können ausgehend von der Basisklasse IndividualBase implementiert werden. Die Klasse IndividualBase besitzt virtuelle Routinen zur Mutation, Rekombination und Fitnessauswertung, welche von den verschiedenen Individuentypen überschrieben werden müssen. Die Selektion wird durch den EAController ausgeführt.

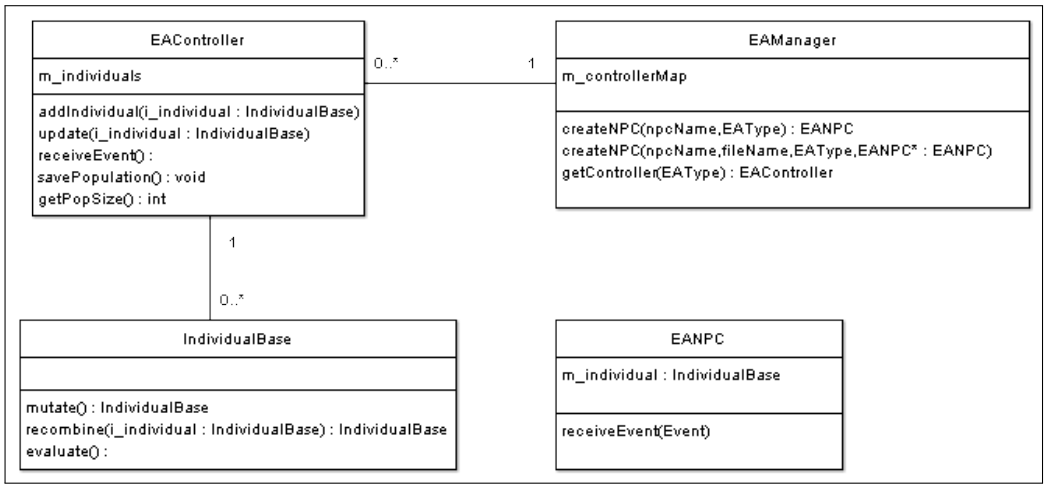


Abbildung 16: Klassendiagramm des Frameworks

6.1.1 EABBGhost - Bitvektor basierter Geist

Dieser Ansatz basiert auf einem $(\mu+\mu)$-EA zur Optimierung pseudoboolscher Funktionen. Da das verwendete Framework die Kombination verschiedener Typen von EAs erlaubt, entspricht μ stets der Anzahl der im Spiel befindlichen Geister dieses Typs. Jede EABBGhost Geisterinstanz enthält ein EABBIndividual Individuum, welches während der Laufzeit des Spiels ausgetauscht werden kann.

Der EA Der EAs wurde mit Hilfe unseres Frameworks in den Kontrollfluss des Spiels (*Gameloop*) integriert, wobei alle $t \in \mathbb{N}$ Gameloopdurchläufe ein EA Schritt ausgeführt wird. In einem EA-Schritt wird für jedes Individuum (*EABBIndividual*) die Fitness berechnet, sowie Selektion und Variation durchgeführt. Damit entspricht die Konstante t der *Lebenszeit* eines Individuums und ist einer der kritischen Parameter des EA. Eine zu geringe Lebenszeit lässt aufgrund der verwendeten Fitnessfunktion (siehe Abschnitt: Die Fitnessfunktion) keine vernünftige Bewertung eines Individuums zu, während eine zu hohe Lebenszeit eine zu geringe Varianz im Verhalten der einzelnen Individuen zur Folge hat, da die Anzahl verhaltensändernder Mutationen zu gering ist. Bei der Entwicklung hat es sich als vernünftig herausgestellt, für die Lebenszeit t einen Wert aus dem Intervall $[150, 250]$ zu wählen. In einer durchschnittlichen Runde Pacman (bestehend aus ungefähr 1350 Gameloops) und einer Lebenszeit von $t = 160$ erzeugt der EA acht Generationen. Die Lebenszeit sollte im Allgemeinen an die durchschnittliche Spieldauer angepasst werden.

Die Fitnessfunktion Die verwendete Fitnessfunktion $f : \{0,1\}^s \to \mathbb{Z}$ bewertet die zur Lebenszeit eines Individuums getroffenen Entscheidungen und ist eine Linearkombination von i Faktoren $m_i : \{0,1\}^s \to \mathbb{N}_0$, sowie Gewichten $g_i \in \mathbb{N}_0, i \in \{1,2,3,4\}$ (Tabelle 5), wobei die Werte der Funktionen m_i aus dem Gameinfo Objekt ausgelesen werden. Daraus ergibt sich folgende Fitnessfunktion:

Tabelle 5: Für NJam verwendete Faktoren und Gewichte der vom EABBIndividual genutzten Fitnessfunktion, wobei $m_i(x)$ den jeweiligen Faktor des Gameinfo Objekts ausliest und g_i dem zugehörigen Gewicht entspricht.

i	$m_i(x)$	g_i	**Beschreibung**
1	Steps	100	Anzahl der Schritte in denen sich X und Y Koordinaten des Individuums x geändert haben
2	Kills	1000	Wie oft wurde Pacman von Individuum x gefressen
3	Deaths	-750	Wie oft wurde Individuum x von Pacman gefressen
4	Score	-10	Anzahl der von Pacman gefressenen Pellets

Tabelle 6: Codierung der Informationen, die einem EABBGhost in jedem Gameloop Schritt zur Verfügung stehen. $v_d, w_d, p_d, q \in \{0,1\}$ mit $d \in \{o,u,l,r\}$. $v_d = 1$, wenn der Geist zuletzt in Richtung d gegangen ist. $w_d = 1$, falls in Richtung d eine Wand ist. $p_d = 1$, falls Sichtkontakt zu Pacman in Richtung d besteht. $q = 1$, wenn Pacman die Powerpille hat.

v_o	v_r	v_l	v_u	w_o	w_r	w_l	w_u	p_o	p_r	p_l	p_u	q

$$f(x) = \sum_{i=1}^{4} m_i(x) \cdot g_i \tag{7}$$

Allgemein kann die Fitnessfunktion, je nach gewünschtem Trainingsziel, über die Auswahl der Faktoren und Gewichte modifiziert werden.

Codierung Ein Individuum wird durch einen Bitvektor konstanter Länge kodiert. Die genaue Länge $s \in \mathbb{N}_0$ hängt von den Informationen ab, die einem Geist pro Gameloop-Schritt zur Verfügung stehen. Daher muss zuerst die Menge aller für einen Geist relevanten Informationen bestimmt werden. Für NJam bestand diese Menge aus lokalen Informationen über die Wände sowie Informationen darüber, ob und in welche Richtung Sichtkontakt zu Pacman besteht und ob Pacman die Powerpille hat. Kurz nach der ersten Implementierung wurde klar, dass jedem Geist zusätzlich Informationen über seine letzte Laufrichtung zur Verfügung gestellt werden müssen, da er sonst leicht in einen *Togglezustand* geraten kann, in dem seine Bewegungen zwischen oben und unten (links und rechts) alternieren. Vor einer Änderung der Codierung wurde vergeblich versucht, dieses Verhalten mit einer Anpassung der Fitnessfunktion zu korrigieren, indem Togglen durch einen großen, negativ gewichteten Faktor bestraft wurde. Die endgültige Codierung der Informationen ist in Tabelle 6 dargestellt. Sie verwendet für jede Richtung $d \in \{o,u,l,r\}$ vier Bits v_d für den letzten Schritt, vier Bits w_d für Wandinformationen, vier Bits p_d für Sichtkontakt zu Pacman sowie ein Bit q für den Status der Powerpille. Damit ergeben sich $s' = 2^{13} = 8192$ verschiedene Informationsvektoren, von denen allerdings nicht alle eine legale Spielsituation codieren (Siehe Abschnitt über Variation). Ein Individuum, repräsentiert durch einen Bitvektor x der Länge $s = 2 * s' = 16384$, codiert nun die Funktionstabelle der Funktion $move_x : \mathbb{N} \to \{UP, RIGHT, LEFT, DOWN\}$ wie folgt:

$$move_x(k) = \begin{cases} UP & , \text{falls } x[2*k] == 0 \wedge x[(2*k)+1] == 0 \\ RIGHT & , \text{falls } x[2*k] == 0 \wedge x[(2*k)+1] == 1 \\ LEFT & , \text{falls } x[2*k] == 1 \wedge x[(2*k)+1] == 0 \\ DOWN & , \text{falls } x[2*k] == 1 \wedge x[(2*k)+1] == 1 \end{cases} \quad (8)$$

Ein Aufruf der Methode move() (Abschnitt 3.3) auf einem EABBGhost erzeugt also den Informationsvektor, konvertiert ihn in Dezimaldarstellung und ließt den nächsten Schritt aus der Funktionstabelle (Bitvektorcodierung des Individuums) aus.

Selektion Die Population P besteht aus so vielen Individuen, wie sich EABBGhost-Instanzen im Spiel befinden. Nach Ablauf der Lebenszeit eines Individuums x, wird dieses mit Hilfe der Fitnessfunktion bewertet und die Population nach Individuen mit schlechterer Fitness durchsucht. Wird so ein Individuum gefunden, wird dieses in der Population durch x ersetzt. Anschließend wird ein neues Individuum $x^{'}$ aus x erzeugt (siehe Variation), welches x im Spiel ersetzt. Falls sich kein Individuum mit geringer Fitness in der Population befindet, wird zufällig gleichverteilt ein Individuum aus P gezogen, variiert und anstelle von x ins Spiel geschickt.

Variation Zu Anfang der Entwicklung wurde Standard-Bitmutation verwendet, was allerdings zu inakzeptablen Ergebnissen führte. Die durch Standard-Bitmutation variierten Individuen liefen oft gegen Wände und der Geist blieb mitten im Spiel stehen. Erst mit Hilfe des folgenden Reparatur-Mechanismus konnte ansprechendes Verhalten der Geister erzielt werden. Im Gegensatz zur Standard-Bitmutation werden bei dem neu entwickelten Variationsoperator keine zufällig ausgewählten Bits invertiert, sondern zufällig ausgewählte Paare von Bits durch andere Paare ersetzt, die eine *legale* Bewegung des Geistes codieren. Eine Bewegung gilt hierbei als legal, wenn der Geist nicht gegen eine Wand läuft oder togglet. Dies ist möglich, da aus dem Index eines zufällig gewählten Bitpaares die Umgebungsinformationen leicht extrahiert werden können (siehe Codierung). Des Weiteren besteht so die Möglichkeit, bereits vorhandenes Problemwissen in die Mutation einfließen zu lassen und beispielsweise einem Geist mit einer vorher festgelegten Wahrscheinlichkeit einen *pursuit-modus* einzucodieren, indem er Pacman jagt, wenn dieser in Sichtweite ist. Die Mutationswahrscheinlichkeit, in diesem Fall die Wahrscheinlichkeit mit der ein Bitpaar zufällig gewählt und durch ein anderes ersetzt wird, ist ein kritischer Parameter des EAs. Eine zu geringe Mutationswahrscheinlichkeit resultiert in einer zu geringen Diversität innerhalb der Population, eine zu hohe zerstört mit hoher Wahrscheinlichkeit gewünschtes Verhalten der Geister. Während der Entwicklung wurden mit einer Mutationswahrscheinlichkeit von $\frac{1}{5}$ gute Ergebnisse erzielt. Bei der Wahl der Mutationswahrscheinlichkeit bleibt außerdem zu beachten, dass (genau wie in der Natur) einige Felder der Codierung ungenutzt sind, was möglicherweise wirkungslose Mutationen zur Folge hat. Dies hängt jedoch stark von der Redundanz der Umgebungscodierung ab (siehe Speicherverbrauch).

Startvektor Beim erstmaligen Start des Spiels sind noch keine Informationen über vorherige Individuen vorhanden. In diesem Fall erzeugt der EA für jedes Individuum einen zufälligen Vektor der ausschließlich legale Bewegungen enthält. Diese legalen Schritte werden mit Hilfe des gleichen Verfahrens wie für den Variationsoperator berechnet. Am Ende

jedes Spiels werden die Vektoren aller sich im Spiel befindenden Individuen gespeichert und beim nächsten Spielstart weiterverwendet. Unabhängig davon, ob die Startvektoren neu erzeugt oder vom Datenträger geladen wurden, werden alle mit dem gleichen Fitnesswert initialisiert, um zu vermeiden, dass sich durch Variation neu erzeugte Individuen niemals gegen zu gut bewertete Individuen durchsetzen können. Für die oben genannte Fitnessfunktion wurde während der Entwicklung eine *Basisfitness* von 1500 für neu initialisierte und geladene Individuen verwendet. Dies stellt kein Problem dar, da ein gutes, durch den angepassten Variationsoperator erzeugtes, Individuum schnell einen höheren Fitnesswert erreicht. Es ist natürlich ebenso denkbar den tatsächlichen Wert zusammen mit dem Individuum auf dem Datenträger zu speichern und nach dem Laden proportional zur Lebenszeit abzusenken.

Performanz und Speicherverbrauch Alle Methoden des hier verwendeten EAs haben höchstens linearen Zeitaufwand in der Länge der Codierung eines Individuums. Tatsächlich ist der Mutationsoperator die einzige Methode, die ein Individuum vollständig betrachtet. Dies könnte jedoch umgangen werden, wenn die zu mutierenden Paare geschickter ausgewürfelt würden. Der Speicherverbrauch beschränkt sich auf die Bitvektoren der Individuen der Population und der sich im Spiel befindlichen Individuen. Diese codieren aber offensichtlich viele ungenutzte Situationen, was aber bei einem Spiel dieser Klasse keine Auswirkungen hatte. Aufgrund der sehr geringen Zeit- und Platzkomplexität ist dieser Ansatz für jede Form von Spielen denkbar.

Mögliche Erweiterungen Eine Möglichkeit die *gefühlte Intelligenz* der Individuen zu steigern, wäre eine Erweiterung der Umgebungscodierung (siehe Codierung) um Bits zur Kommunikation mit anderen Individuen. Diese könnten beispielsweise die Richtung codieren, in der sich weitere Geister befinden. Falls Wert auf gesteigerte Aggressivität der Geister gelegt wird, kann eine globale Sicht auf Pacman, Wissen über die Positionen der Powerpillen oder der Pellets hinzugenommen werden. Ob diese Änderungen den Schwierigkeitsgrad tatsächlich erhöhen würden, bleibt offen, allerdings muss beachtet werden, dass jedes zusätzliche Bit in der Umgebungscodierung die Länge der Codierung eines Individuums verdoppelt. Für komplexere Spiele, die einem Individuum eine Vielzahl an Informationen bieten, sollte eine möglichst platzoptimierte Umgebungscodierung gewählt werden.

6.1.2 ProbEAGhost

Dem ProbEAGhost liegt die Idee zu Grunde, Richtungsentscheidungen randomisiert zu treffen. Die Wahrscheinlichkeitsverteilung, anhand derer diese Entscheidung getroffen wird, ist Gegenstand der Optimierung durch den EA. Der Geist wurde so entworfen, dass er als Individuum eine Abbildung von seiner Situation auf Wahrscheinlichkeiten für die Bewegungsrichtungen benutzt. Die Umgebung besteht aus den vier Blickrichtungen, in die der Geist blicken kann. In jeder Richtung gibt es folgende Aspekte zu bemerken, wobei jeder Aspekt durch eine Zweierpotenz kodiert wird:

$1 = 2^0$ Das nächste Feld ist eine Wand

$2 = 2^1$ Pacman ohne Powerpille ist in Sicht

$4 = 2^2$ Pacman mit Powerpille ist in Sicht

$8 = 2^3$ Der Geist selbst bewegt sich in diese Richtung

Zusätzlich besitzt der Geist ein Gedächtnis, das drei Züge lang speichert, in welcher Richtung Pacman das letzte Mal gesehen wurde. Für jede Richtung werden die Zahlen der gegebenen Aspekte aufaddiert. So erhält man eine vierstellige Hexadezimalzahl, die eine Situation bei festgelegter Reihenfolge der Richtungen eindeutig kodiert.

Für jede neue Situation, in die der Geist gerät, wird die entsprechende Kodierung zusammen mit einer zufällig generierten, diskreten Wahrscheinlichkeitsdichte abgespeichert. Es werden also nur Situationen abgespeichert, die auch wirklich auftreten können. Die Wahrscheinlichkeitsdichte gibt die Wahrscheinlichkeiten an, mit der sich der Geist in der Situation in die jeweilige Richtung bewegt. In Listing 6 sieht man eine Liste von Situationen mit ihren Wahrscheinlichkeitsdichten. Die Selektion findet jeweils am Ende eines Spiels statt. Der schlechteste Geist wird mit dem besten rekombiniert, mutiert und tritt dann ins nächste Spiel ein. Die übrigen Geister bleiben unverändert.

```
0008    0.293175    0.0849953   0.599749    0.0220807
0009    0.0175201   0.235937    0.109365    0.637178
000A    0.316012    0.0446417   0.352972    0.286375
000C    0.16813     0.0131479   0.425819    0.392904
0018    0.167732    0.107992    0.360976    0.3633
0019    0.122501    0.50201     0.256166    0.119323
001A    0.245868    0.0215577   0.379222    0.353352
001B    0.202347    0.247959    0.132165    0.417529
001C    0.290209    0.613649    0.057337    0.0388048
0028    0.242547    0.0738313   0.448448    0.235173
0029    0.392495    0.26104     0.168777    0.177688
⋮
```

Listing 6: Auszug aus einem Individuum. Die erste Zahl einer Zeile gibt immer die Situation an. In der ersten Zeile ist beispielsweise die Situation kodiert, in der sich in allen vier Richtungen Wege befinden und der Geist sich bisher nach links bewegt. Darauf folgen die vier Wahrscheinlichkeiten für die Richtungen. Die Reihenfolge der Richtungen ist up, right, down, left. In der ersten Zeile beträgt also die Wahrscheinlichkeit, nach unten abzubiegen, ungefähr 0,6.

Mutation Die Mutation findet statt, indem von der Wahrscheinlichkeit a einer Bewegungsrichtung ein Betrag abgezogen und auf eine andere Wahrscheinlichkeit b einer Richtung in derselben Wahrscheinlichkeitsdichte addiert wird. So wird gewährleistet, dass die Summe der Dichte gleich eins bleibt. Zur Ermittlung des Betrags wird zunächst eine $N(0; 0, 1)$-verteilte Zufallszahl z_{max} gezogen. Dann wird

$$z_1 = \begin{cases} 2a - z_{max}, & \text{falls } a - z_{max} < 0 \\ z_{max}, & \text{sonst} \end{cases} \tag{9}$$

und

$$z_2 = \begin{cases} 2 - 2b - z_1, & \text{falls } b + z_1 > 1 \\ z_1, & \text{sonst} \end{cases} \tag{10}$$

ermittelt. Da z_2 der Betrag ist, der letztendlich verschoben wird, ist sichergestellt, dass das Intervall $[0, 1]$ nicht verlassen wird. Stattdessen bewirken die Berechnungen eine Art „Spiegelung an den Grenzen", falls diese überschritten würden. Diese Mutation wird für $2 \cdot \log$ (Anzahl Situationen) zufällig ausgewählte Situationen durchgeführt.

Rekombination Das neue Individuum erhält alle Situationen, die den Eltern bekannt waren. Es handelt sich dabei um eine diskrete Rekombination, das heißt für jede Situation wird immer die Wahrscheinlichkeitsdichte nur eines Elters gewählt. Dabei beträgt die Wahrscheinlichkeit, die Situation vom schlechteren Elter zu übernehmen $0,9$ und vom besseren Elter $0,1$. Vorbild für diese Rekombination ist der Genaustausch zwischen Einzellern, die kurze Fäden von RNA untereinander austauschen. Das Verhältnis wurde so ungleich gewählt, um die Vielseitigkeit in der Population zu erhalten.

Training Der erste Versuch, den ProbEAGhost zu trainieren, war wenig erfolgreich. Die Zielfunktion wurde dabei immer am Ende eines Spiels ausgewertet. In die Zielfunktion gingen über ein ganzes Spiel gesehen

- a =Anzahl der Kills,
- b =Anzahl der Deaths,
- $c = |\,\text{maxPosX} - \text{minPosX}\,|$ und
- $d = |\,\text{maxPosY} - \text{minPosY}\,|$ ein.

Die Zielfunktion war $50a - 50b + c + d$ und sollte maximiert werden. Dabei waren maxPosX und so weiter die extremalen Positionen auf dem Spielfeld, die der Geist jemals im Spiel einnahm. Diese Zielfunktion sollte diejenigen Geister belohnen, die Pacman häufig erwischen, die selten von Pacman gefressen werden und die sich aktiv auf dem Spielfeld bewegen. Als Gegner zum Training wurden vier aufgezeichnete Spiele eines menschlichen Spielers benutzt.

Das Resultat nach 2000 automatisierten Spieldurchläufen war, dass sich die Geister weniger bewegten, als mit ihrem zufällig initialisierten Verhalten. Anhand der Kodierung des Verhaltens war erkennbar, dass sich die Geister wie gewünscht im Normalfall mit höherer Wahrscheinlichkeit auf Pacman zu bewegten und sich eher von ihm entfernten, wenn er die Powerpille hatte. Allerdings war dieses Verhalten zu schwach ausgeprägt, um subjektiv im Spielverhalten erkennbar zu sein.

Problemgebiete waren vor allem folgende Felder:

- Die Zielfunktion war zu wenig spezifisch, da sie nur das Töten von Pacman belohnte, nicht aber die Annäherung.

- Das Benutzen einer Aufzeichnung für die Steuerung von Pacman führte dazu, dass keine Interaktion mit den Geistern stattfand. Pacman lief also häufig zufällig in einen Geist hinein. Dieser musste nichts aktiv für seinen Erfolg tun.
- Die Repräsentation der Situationen war zu speziell, so dass eine Situation, die nicht völlig identisch mit einer bereits bekannten war, völlig neu erlernt werden musste.

Insgesamt ist der Ablauf in einem aufgezeichneten Spiel zu statisch. Es stellen sich Überanpassungen an den Ablauf der aufgezeichneten Spiele ein.

Verbesserung des Ansatzes Aufgrund der oben genannten Probleme wurden einige Vereinfachungen am Individuum vorgenommen. Das Individuum wurde so abgeändert, dass es nicht mehr gesamte Situationen aus vier Richtungen, sondern nur noch einzelne Richtungen mit einer festen Gewichtung enthielt (Abbildung 7). Aus diesen Gewichten wurde dann eine Wahrscheinlichkeitsdichte erzeugt. Die Richtungen i wurden aufsteigend nach ihren Gewichten g_i sortiert. Dann wurde jedes Gewicht mit seinem Rang potenziert. Anschließend wurden die Ergebnisse dieser Operation so normiert, dass ihre Summe 1 ergab. Damit hat man eine Wahrscheinlichkeitsdichte erhalten.

```
0   0.32143
1   0.0122396
2   0.645948
3   0.229656
4   0.376831
5   0.105484
8   0.688019
9   0.0494717
```

Listing 7: Ein komplettes, vereinfachtes Individuum. Die erste Zahl einer Zeile gibt dabei die Eigenschaften einer Richtung an. Die Kodierung ist dabei genauso, wie zu Beginn dieses Abschnitts beschrieben. Die zweite Zahl gibt die Gewichtung dieser Aspekte an. Zum Beispiel beschreibt die $9 = 2^0 + 2^3$ in der letzten Zeile eine Richtung, in die der Geist gerade läuft und in der das nächste Feld eine Wand ist. Entsprechend unserer Erwartung ist das Gewicht dafür, und somit die Wahrscheinlichkeit gegen die Wand zu laufen, gering.

Die Zielfunktion wurde so abgeändert, dass sie auch die Annäherung an Pacman belohnte. Außerdem wurde ein kleiner Strafterm eingeführt, der den Richtungswechsel in die entgegengesetzte Richtung bestrafte. Die verbesserte Zielfunktion bestand also aus

- $a =$ Anzahl der Kills,
- $b =$ Anzahl der Deaths,
- $c = |\,\text{maxPosX} - \text{minPosX}\,|$,
- $d = |\,\text{maxPosY} - \text{minPosY}\,|$,
- $e =$ durchschnittl. Manhattendistanz zu Pacman und
- $f =$ Anz. der Richtungswechsel in die entgegengesetzte Richtung.

Die zu maximierende Zielfunktion war $2a - 3b + c + d - e - 0,001f$. Zusätzlich wurde noch ein NPC für Pacman implementiert, der sich nach einfachen deterministischen Regeln auf dem Spielfeld bewegt. Dieser NPC ist in Abschnitt 4.2 näher beschrieben.

Mit diesem Individuum verlief das Training deutlich erfolgreicher. Dies dürfte vor allem der stark vereinfachten Abbildung geschuldet sein, da so ähnliche Situationen automatisch mitgelernt wurden. In Listing 7 sieht man das beste Resultat des Lernprozesses nach ca. 500 Spielen. Man erkennt, dass die Gewichtung zum gegen die Wand zu laufen sehr gering ist (1 und 9) und die Annäherung an Pacman ohne Powerpille (2) höher bewertet wird, als die Annäherung an Pacman mit Powerpille (4). Die Differenz zwischen den Fällen 2 und 4 erweist sich aber im Spiel als nicht groß genug für ein intelligent wirkendes Jagd- und Fluchtverhalten.

6.1.3 OnlineEA

Evolutionäre Algorithmen werden im industriellen Einsatz häufig zur on-line Steuerung von Prozessen benutzt. Der Grund liegt in der Tatsache, dass evolutionäre Algorithmen schon während der Laufzeit Lösungen einer Optimieraufgabe bereitstellen. Des Weiteren werden auch nach wenigen Iterationen oft schon brauchbare Lösungen erzeugt, die bereits eingesetzt werden können. Dies hat den Vorteil, dass auf eintretende Ereignisse im Steuerungsprozess schnell reagiert werden kann.

Basierend auf dieser Idee wurde ein NPC entwickelt, der lokal und zur Laufzeit – also on-line – seine Richtungsentscheidungen durch einen evolutionären Algorithmus evaluiert. Dieser NPC sollte die Fähigkeit besitzen, den Pacman *zu jagen*. Dazu erhielt der sogenannte OnlineEA eine globale Sicht auf das Spielfeld, um kürzeste Wege auf dem Spielfeldgraphen zu errechnen. Die Repräsentation des Individuums wurde sehr einfach gehalten und besteht nur aus zwei Koordinaten, welche zusammen eine Position auf dem Spielfeld auszeichnen. Eine Mutation auf das Individuum ermittelt eine neue Position, in dem ganzzahlige Zufallszahlen aus dem Intervall $[-5, 5]$ auf die Koordinaten addiert werden. Das Intervallgrenzen wurden aus empirischen Gesichtspunkten gewählt. An dieser Stelle war es wichtig zu überprüfen, ob diese zufällig ermittelten Koordinaten auch eine gültige Spielfeldposition beschreiben und keine Position einer Wand ermittelt wurde oder sogar das Spielfeld verlassen wurde.

Die Bewertung des Individuums betrachtet hauptsächlich die Distanz der Position des Individuums zu der Position von Pacman. Da der NPC den Pacman jagen soll, ist natürlich diese Distanz zu minimieren. Im Verlauf des Spiels wird versucht die Position von Pacman zu approximieren, da diese die geringste Distanz besitzt. Damit der NPC den Pacman verfolgt, berechnet der Geist zu der Position seines besten Individuums einen kürzesten Weg und bewegt sich auf diesem Pfad. Die Fitnessfunktion ist in Gleichung (11) beschrieben.

$$\text{Fitnessfunktion} = d \longrightarrow \min! \text{ mit } d \in \mathbb{N} \text{ der kürzester Weg auf dem Spielfeld zu Pacman.} \tag{11}$$

Während der Entwicklung wurde die Erkenntnis gewonnen, dass mehrere von einem OnlineEA gesteuerte Geister sich sehr häufig gleich verhalten und dieselben Wege auf dem Spielfeld gehen. Um dies zu verhindern, wurde die Fitnessfunktion dahingegend erweitert,

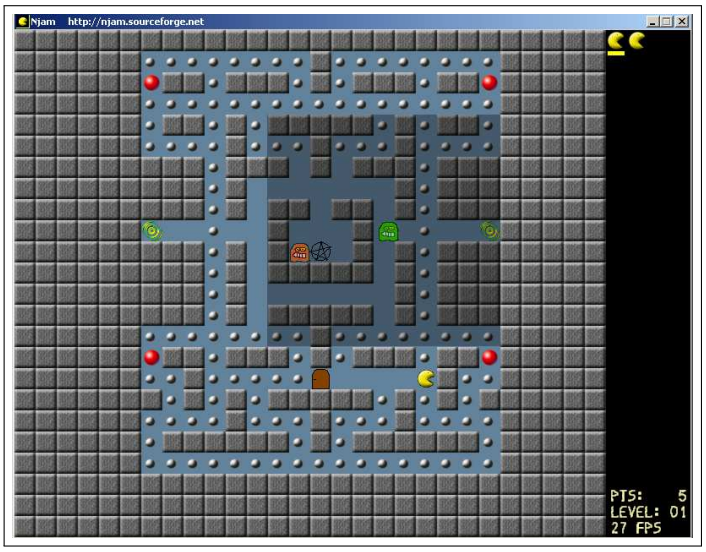


Abbildung 17: Der dunkle Bereich kennzeichnet die Spielfelder, die für den grünen Geist durch eine Mutation erreicht werden können

dass die Geister untereinander, trotz der Nähe zum Pacman, einen möglichst großen Abstand voneinander halten sollen. Die Einführung des Strafterms θ (12) hatte zum einen den Vorteil, dass die Geister versuchten Pacman von mehreren Seiten einzukesseln. Zum anderen ist diese indirekte Kommunikation mit den anderen Geistern unabhängig von den übrigen Geistertypen und funktioniert zum Beispiel genauso mit MrNeuro (siehe Abschnitt zu Beginn von Kapitel 5).

$$\text{Strafterm}\,\theta = \frac{1}{4}\sum_{i=1}^{4} d_i \text{ mit } d_i \in \mathbb{N} \text{ der Distanz zu Geist i.} \tag{12}$$

Die Gewichtung des Strafterms war eine kritische Einstellung. Wurde die Gewichtung zu hoch gewählt, hat der Strafterm die zu minimierende Distanz zu Pacman dominiert und die Geister näherten sich nicht mehr Pacman an. Bei zu niedriger Gewichtung wurde kaum ein verbesserter Effekt festgestellt. Zusätzlich wurde ein weiterer Faktor in die Fitnessfunktion integriert. Die Geister sollten die Positionen bevorzugen, auf denen noch eine Pille zur Verfügung steht, weil diese Pacman höchstwahrscheinlich als nächstes ansteuern wird (boolscher Parameter p nimmt den Wert 1 an, fall auf dem Feld eine Pille vorhanden ist). Der letzte nicht weniger wichtige Faktor in der Fitnessfunktion steuert das Verhalten des OnlineEA im Fall, dass Pacman die Powerpille besitzt (Parameter α). In diesem Szenario soll der NPC die Distanz zu Pacman möglichst maximieren. Für die Fitnessfunktion bedeutet dies, dass die Distanz zu Pacman subtrahiert wird und damit eine größere Distanz die Fitnessfunktion verringert. Abschließend ergibt sich die Fitnessfunktion zu Formel (13):

$$\text{Fitnessfunktion} = d*\alpha + 2.2*p - 1.5*\theta \longrightarrow \text{min! wobei } \alpha \in \{1,-1\} \text{ und } p \in \{0,1\} \tag{13}$$

Die Gewichtung der einzelnen Parameter wurde empirisch ermittelt.

Vorgehensweise Wie im vorherigen Abschnitt erklärt, soll der OnlineEA Pacman möglichst geschickt verfolgen. Um diese Vorgehensweise durchzuführen, benötigt einerseits der OnlineEA globale Sicht auf das Spiel, um seine Fitnessfunktion auszuwerten und die Möglichkeit schnell seine Bewegungen auf die neue Position von Pacman anzupassen. Die Richtungsentscheidungen sollen zur Laufzeit – also on-line – ermittelt werden. Der OnlineEA führt auf jeder Spielkachel einen $(1+1)$EA aus, der in fünf Generationen versucht ein neues gutes Individuum zu ermitteln, welches eine aussichtsreiche Position innerhalb der Umgebung von Pacman approximiert. Anschließend wird ein kürzester Weg von der aktuellen Poistion des NPCs zu der Position des Individuums berechnet und der OnlineEA schlägt für den nächsten Schritt die Richtung dieses Pfades ein. Da auf jeder Spielfeldkachel dieser Vorgang wiederholt wird, ist es ausreichend, immer in die Richtung des gerade aktuell kürzesten Pfades zu laufen und es muss kein kompletter Pfad gespeichert werden (siehe 8).

```
1. Ist Geist auf der Mitte einer Spielfeldkachel?
                Falls ja gehe zu 2, sonst ENDE
2. Starte (1+1)EA mit letztem Individuum
                (5 Generationen)
3. Berechne kuerzesten Weg zur Position des besten
                Individuums
4. Waehle Richtung, die auf dem kuerzesten Weg liegt
```

Listing 8: Algorithmus

Fazit und Probleme Der OnlineEA besitzt den Vorteil, dass dieser nicht trainiert werden muss, sondern alle Entscheidungen im Spiel evaluiert und getroffen werden. Dazu benötigt er allerdings im Gegensatz zu den anderen Geistern eine globale Sicht auf die Spielkarte. Dass er nicht trainiert werden musste, erwies sich als praktisch bei der Entwicklung des OnlineEAs. Änderungen des Verhaltens und das Anpassen der Fitnessfunktion konnten schnell getestet und verifiziert werden. Als ein Problem hat sich herausgestellt, dass der OnlinEA Pacman nicht immer fängt, wenn er und weitere NPCs sich in der Nähe von Pacman befinden. In dieser Situation toggelt der NPC in kurzer Distanz zu Pacman zwischen mehreren Kacheln, weil die mittlere Distanz zu den restlichen Geistern maximiert werden soll. Dies ließ sich auch durch Justierung der Fitnessfunktion nicht ganz beheben.

7 Feldstudie / Campusfest

Wir nahmen am Campusfest der Universität Dortmund am 16. Juni 2007 mit einem eigenen Stand teil. Diese Entscheidung fiel aus zwei Gründen: Zum Einen hatten wir in unsere Entwicklungen eine Menge Arbeit gesteckt, die wir auch gern präsentieren wollten, zum Anderen interessierte uns, ob sich das von G. N. Yannakakis und J. Hallam [YH04] postulierte Spielspaßmaß in der Praxis bewahrheiten würde.

7.1 Spielspaßmaß nach Yannakakis und Hallam

2004 entwickelten G. N. Yannakakis und J. Hallam in ihrem Artikel „Evolving Opponents for Interesting Interactive Computer Games“ [YH04] ein auf Pacman bezogenes Interessantheitsmaß. Dieses stellen wir im Folgenden kurz vor; eine ausführlichere Beschreibung ist in dem bezeichneten Artikel oder im entsprechenden Seminar über Spielspaß (Anhang J) zu finden. Die Interessantheit eines Spieles, beziehungsweise den Spielspaß, bestimmen Yannakakis und Hallam aus drei Komponenten: der *Schwierigkeit* des Spieles, sowie der *Strategievielfalt* und der *Gegneraktivität*. In ihren ursprünglichen Formeln gewichtet Yannakakis und Hallam die Einzelwerte mit jeweils einem Exponenten p_1, p_2, und p_3. Diese setzen wir für unsere Untersuchung auf 1, weil wir annehmen, dass alle drei Komponenten für den Spielspaß gleichermaßen wichtig sind; da Potenzen mit Exponent 1 den Wert ihrer Basis haben werden diese Faktoren im Folgenden weggelassen.

Schwierigkeit Ein Spiel macht Spaß, wenn es den Spieler weder über- noch unterfordert. Es sollte also weder zu leicht, noch zu schwierig sein. Hierzu werden für ein Spiel k die Anzahl der Schritte t_k gemessen, die die Gegner machen, bis sie Pacman fangen. Den über eine festgelegte Anzahl N von Spielen gemessenen Durchschnitt bezeichent man als $E\{t_k\}$. Die größte benötigte Zahl von Schritten, bis Pacman gefressen wurde, bezeichnet man als $max\{t_k\}$. Die Spielschwierigkeit T ist somit

$$T = 1 - \frac{E\{t_k\}}{max\{t_k\}}. \tag{14}$$

und $0 \leq T \leq 1$ (Beweis in Abschnitt J.5.1).

Strategievielfalt Gegner, die immer nach der gleichen Strategie vorgehen, machen ein Spiel für den Spieler schnell langweilig. Als Bewertungsgrundlage gehen Yannakakis und Hallam davon aus, dass die unterschiedliche Dauer eines Spiels auf eine unterschiedliche Strategie schließen lässt.

Das Maß S benutzt Beobachtungen aus N verschiedenen Spielen. Als Konstanten fließen sowohl die Anzahl an Schritten ein (t_{max}), die Pacman mindestens machen musste, um das Spiel siegreich zu beenden (d.h. um alle Pellets zu essen und den Geistern auszuweichen), als auch die Anzahl an Schritten (t_{min}), die Geister mindestens zurücklegen mussten, um Pacman zu fangen. Wenn nun σ die Standardabweichung von t_k über die Gesamtanzahl von N Spielen ist und σ_{max} wie folgt definiert wird:

$$\sigma_{max} = \frac{1}{2}\sqrt{\frac{N}{(N-1)}}(t_{max} - t_{min}) \tag{15}$$

dann läßt sich die Variabilität mit der folgenden Funktion S messen

$$S = \frac{\sigma}{\sigma_{max}}. \tag{16}$$

S liegt im Intervall $[0, 1]$ (Beweis in Abschnitt J.5.2).

Gegneraktivität Die Gegner sollen sich vielfältig bewegen. Ein Gegner, der an einer Stelle verharrt oder nur einen kleinen Teil des Labyrinthes bewacht ist für den menschlichen Spieler leicht zu durchschauen und auszutricksen. Es ist also wünschenswert, dass die Geister möglichst alle Felder im Labyrinth in etwa gleich oft besuchen.

Für das Maß H der Gegeneraktivität ziehen Yannakakis und Hallam die Entropie heran. v_{ik} sei die Anzahl der Besuche der Geister auf dem Feld i im betrachteten Spiel k, V_n die Gesamtanzahl von Besuchen aller Zellen ($V_n = \sum_i v_{in}$).

$$H_n = -\frac{1}{\log V_n} \sum_i \frac{v_{in}}{V_n} \log(\frac{v_{in}}{V_n}). \tag{17}$$

H liegt im Intervall $[0, 1]$ (Beweis in Abschnitt J.5.3).)

Spielspaß Der Spielspaß oder Interessantheitsgrad I ist nun die mit γ, δ und ϵ (jeweils aus $\mathbb{R}_+$) gewichtete Summe der drei gewonnen Einzelwerte

$$I = \frac{\gamma T + \delta S + \epsilon E\{H_n\}}{\gamma + \delta + \epsilon} \tag{18}$$

und liegt, ebenso wie seine Teilkomponenten, im Intervall $[0, 1]$ (Beweis in Abschnitt J.5.4). Auch hier setzen wir die Gewichte wiederum auf 1 und erhalten als Interessantheitsgrad I

$$I = \frac{T + S + E\{H_n\}}{3}. \tag{19}$$

7.2 Versuchsaufbau

Um das besprochene Interessantheitsmaß in der Praxis zu erproben, ersannen wir den folgenden Versuchsaufbau:

Mit dem zu NJam gehörenden Leveleditor bauten wir das Pacman-Originallabyrinth nach (vergleiche auch Abbildung 3 in Kapitel 3). Um die Qualität der von uns modifizierten Geister zu prüfen, stellte jede Teilgruppe ein Set aus vier Geistern zusammen, diese waren im Einzelnen:

deterministisch/randomisierte Strategien Firou, Old Shimer, Darmok und Puck

Neuronale Netze MrNeuro, basierend auf vier zuvor trainierten Netzen. Drei der Netze besitzen einen Generalisierungsfehler $< 10\%$, eines der Netze einen Generalisierungsfehler $< 50\%$.

Evolutionäre Algorithmen ProbEA, OnlineEA und zwei EABBGhost

Abbildung 18: Bildschirmfoto einer der von den Spielern nach Absolvieren der Testspiele zu beantwortenden Fragen.

Das Spiel wurde so umgeschrieben, dass ein Spieler zweimal dieses Labyrinth spielen musste. Hierbei wurde jedes Labyrinth als ein Spiel betrachtet, zu dem der Spieler drei Leben zur Verfügung hatte und das, gemäß den Regeln in Njam (Kapitel 3) gewonnen oder verloren werden konnte. In jedem Fall spielte er aber beide Spiele.

Aus den Geistersets wurden unter Berücksichtigung der Spielreihenfolge die sechs möglichen Paarungen bestimmt. Jeweils eine dieser Paarungen wurde gleichverteilt zufällig für jeden Versuch gezogen.

Am Ende des Versuchs wurde den Spielern Fragen gestellt, die mit verschiedenen Auswahlmöglichkeiten zu beantworten waren (Abbildung 18). Die einzelnen Fragen werden zusammen mit ihren Auswahlmöglichkeiten im Folgenden aufgezählt. Die Grundeinstellung war hier die oberste Antwort, in der Aufzählung durch hervorgehobene Schrift angedeutet.

- Wie alt bist Du?
  - **jünger als 14 Jahre**
  - 15 bis 17 Jahre
  - 18 bis 22 Jahre
  - 23 bis 29 Jahre
  - älter als 30 Jahre
- Geschlecht?
  - **männlich**
  - weiblich
- Spiel 1 hat mit . . . Spaß gemacht als Spiel 2!
  - **viel mehr**
  - mehr

  - genauso viel
  - weniger
  - viel weniger

- Spiel 1 war ... als Spiel 2!
  - **viel schwieriger**
  - schwieriger
  - genauso schwieriger
  - leichter
  - viel leichter

- Die Geister in Spiel 1 waren ... als in Spiel 2!
  - **viel klüger**
  - klüger
  - genauso klüger
  - dümmer
  - viel dümmer

- Wie häufig spielst Du Computerspiele?
  - **jeden Tag**
  - mehrmals pro Woche
  - einmal pro Woche
  - einmal pro Monat
  - seltener

Die Benutzerangaben sollten uns zum Einen ein Bild über das Publikum vermitteln, das an unserem Projekt interesse gezeigt hat. Zum Anderen (und wichtiger) soltte uns diese Angaben einen Eindruck über die Qualität der spielbezogenen Daten liefern, der im zweiten Teil des Fragebogens miterfragt wurde. Insbesondere der Expertisengrad hat sich als ein gewichtiger Faktor in diesem Versuchsaufbau erwiesen, wie in 7.2.2 angemerkt wird. Die spielbezogene Daten, wie beispielsweise die Anzahl der Punkte, Sieg oder Niederlage, die Anzahl der Schritte zum Sieg oder zur Niederlage, wurden im Anschluss für die Berechnung der Yannakakis-Hallam Formel benutzt.

7.2.1 Ergebnisse

Daten Auf dem Campusfest haben wir 245 Datensätze aufgezeichnet. Aus diesen wurden im Anschluss diejenigen gefiltert, die fehlerhafte Daten beinhalteten. Damit werden der Auswertung 184 Datensätze zugrunde gelegt, die sich nahezu gleichmäßig auf die einzelnen Versuchsbedingungen verteilten, so dass wir 26 bis 34 Datensätze pro Versuchsbedingung erhalten.

Die Eingaben in den Fragebögen wurden von 0 bis 4 codiert, wobei 0 für die oberste Antwort steht und 4 (oder im Falle der Frage nach Geschlecht 1) für die letzte mögliche Antwort.

Ein kurzer Überblick über die Altersgruppen, der in der Tabelle 10 im Anhang Mdargestellt ist, zeigt, dass die Mehrheit der Probanden ungefähr unserer Zielgruppe ($18 - 29$ Jahre) entsprochen hat. Wie an einem Stand des Fachbereichs Informatik zu erwarten, haben wir eine deutliche Mehrheit an männlichen Probanden. Die Verteilung über die unterschiedlichen Expertisegrade zeigt eine deutliche Orientierung zu den Extremwerten (Tabelle 10).

Vorgehen Zunächst haben wir aus den mitgeschriebenen Daten die Yannakakis-Hallam (YH) Formel 18 berechnet. Diese wurde jeweils für die ersten und für die jeweiligen zweiten Spiele berechnet. Da die Yannakakis-Hallam Formel über eine Menge von Spielen berechnet wird, haben wir folgende Einteilung der Spiele gewählt: Alle Spiele wurden nach ihrer Bewertung durch den Spieler in fünf Gruppen eingeteilt. Diese Aufteilung bietet sich an, da wir eine Aussage darüber machen wollen, ob die Yannakakis-Hallam Formel tendenziell die gleichen Werte liefert, wie von unseren Probanden empfunden. Das bedeutet, dass wir die Formel dann als verifiziert betrachten, wenn sie in einem Spielset, das zum Beispiel mit einer 4 bewertet wurde (bedeutet: 'Spiel 1 hat viel mehr Spaß gemacht als Spiel 2'), für das erste Spiel einen höheren Wert liefert als für das zweite. Da diese Aussage möglichst über alle fünf möglichen Bewertungen gelten (oder nicht gelten) sollte, um eine Entscheidung über Akzeptieren (oder Ablehnen) der Formel als gültiges Spielspaßmaß zu ermöglichen, haben wir die Werte für jede der fünf Bewertungen berechnet.

Da die Spielerbewertung einen einzigen vergleichenden Spaßwert für beide Spiele liefert, haben wir die Differenz der beiden aus der Formel errechneten Spaßwerte gebildet. Korreliert wurde dann die Differenz der Yannakakis-Hallam Werte mit unserem Spaßwert.

Die Daten wurden einem Signifikanztest mit Konfidenzintervallen von $0,95$ unterzogen. Die Prüfgrößen und die Werte des Konfidenzbereichs sind mit der R-Funktion cor.test() errechnet. Die Daten wurden unter der alternativen Hypothese, dass eine Korrelation von 0 besteht, getestet.

Resultate Die Werte, die sich aus der YH Formel nach dem obigen Berechnungsprinzip ergaben, sind in der Tabelle 7 dargestellt. Korreliert mit unserer Codierung der Umfrageergebnisse ergibt sich ein Korrelationskoeffizient von $-0,721475$.

Da die YH Werte und die Spaßbeurteilung durch die Probanden den gleichen Umstand beschreiben, kann man von einem erwarteten linearen Zusammenhang ausgehen. Damit ist die Verwendung des Korrelationskoeffizieten als Vergleichsmittel gerechtfertig. Der von uns errechnete Korrelationswert weist auf einen (potenziell) hohen negativen linearen Zusammenhang hin. Bei einem hohen YH Wert würden wir eine deutlich geringen Spaßwert vorhersagen.

Der zweiseitige Signifikanztest (Ergebnisse in Abbildung 19) liefert uns jedoch die Wahrscheinlichkeit von $16,89\%$ (entspricht dem p-Wert), dass diese Vorhersage tatsächlich zutrifft. Daher wird die alternative Hypothese akzeptiert und wir nehmen an, dass kein (linearer) Zusammenhang zwischen YH Spielspaßwerten und den auf dem Campusfest abgegebenen Bewertungen besteht.

Tabelle 7: Ergebnisse der Spielspaßmessung nach Yannakakis/Hallam für die von den Spielern bewerteten Spielen.

Bewertung	#VPn	Spiel	YH-Wert	Entropie	Variabilität	Schwierigkt.
viel mehr	22	1	0,6879318	0,7224017	0,6314841	0,7099096
		2	0,8631313	0,7262228	1,1258430	0,7373280
mehr	46	1	0,6846814	0,713283	0,7825843	0,5581770
		2	0,7410708	0,7139312	0,8385552	0,6707259
genauso viel	73	1	0,7895172	0,7169494	0,8502014	0,8014008
		2	0,6002498	0,715003	0,3902309	0,6955154
weniger	39	1	0,6741688	0,7238208	0,596705	0,7019806
		2	0,6978188	0,7166375	0,6899609	0,6868581
viel weniger	3	1	0,4972591	0,702338	0,2173145	0,5721248
		2	0,3407217	0,740357	0,033945	0,2478632

```
R Console

> erg
         diff wert
1 -0.17519947    2
2 -0.05638932    1
3  0.18926745    0
4 -0.02365003   -1
5  0.15653735   -2
> cerg <- cor.test(erg$diff, erg$wert, alternative = "two.sided", conf.level = 0.95)
> cerg

        Pearson's product-moment correlation

data:  erg$diff and erg$wert
t = -1.8047, df = 3, p-value = 0.1689
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.9799627  0.4423820
sample estimates:
       cor
-0.7214754
```

Abbildung 19: Ausgabe der R-Konsole: Ergebnis eines zweiseitigen Signifikanztests mit der alternativen Hypothese: Korrelationskoeffizient ist 0

7.2.2 Methodenkritik

Da die ermittelten Daten wie bereits beschrieben eine gesicherte Aussage nicht zulassen, ist die Frage angebracht, *warum* es zu diesem Ergebnis kam. Einer der Gründe für das Ergebnis könnte sein, dass Spielspaß ein höchst subjektives Empfinden ist, dass sich überhaupt nicht allgemein bestimmen lässt, weil jeder Mensch unterschiedliche Vorlieben und Interessen hat. Dem steht entgegen, dass die Psychologie, zu deren Disziplin diese Fragestellung zu zählen ist, unterscheidet zwischen allgemeiner und differentieller Psychologie und bei sehr vielen Fragestellungen sehr wohl Aussagen über alle Menschen treffen kann, die jedoch beim einzelnen Indivuduum verschieden stark zutreffen können.

Der wohl gewichtigere Grund für die mangelnde Aussagekraft der gewonnenen Daten wird sein, dass die Versuchspersonen bei den gespielten Spielen auf Unterschiede, insbesondere in der Bewegung und Strategie der Geister, gar nicht geachtet haben, weil sie zu sehr

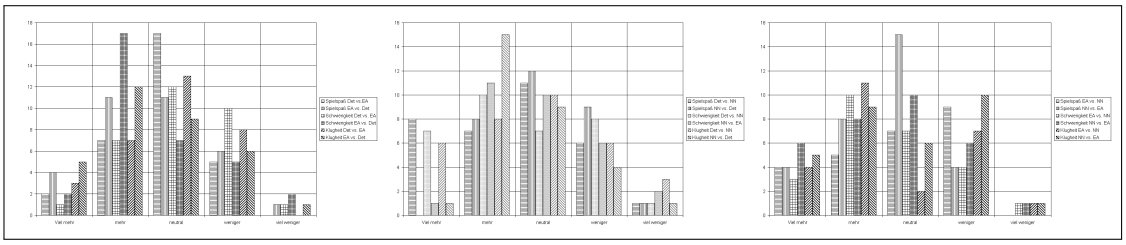


Abbildung 20: Antworten der Versuchspersonen auf die Strategiekombiationen dr-EA (links), dr-NN (Mitte) und EA-NN (rechts)

damit beschäftigt waren, die Steuerung für das Spiel zu erlernen – dieses wurde während des Campusfestes des häufigeren direkt von Versuchspersonen benannt, wenn sie nach dem Spiel noch mit Teilnehmerinnen der Projektgruppe sprachen.

Ein dritter möglicher Grund könnte die Verwendung einer Skala mit ungerader Itemanzahl sein, eine neutrale Antwortmöglichkeit schien unseren Daten zufolge zumindest sehr attraktiv für viele Teilnehmer, 32% aller Daten entfallen auf den neutralen Wert (Abbildung 71). Darüber hinaus gaben 43% positive Antworten ab (12% „viel besser/klüger/...“, 31% „besser/klüger/...“). Ob es sich hierbei um einen generellen Effekt handelt, oder ob den Versuchspersonen das erste Spiel tatsächlich spannender und klüger vorkam, ist im Rahmen dieses Projektes nicht zu klären.

Nicht zuletzt sind auch die Besonderheiten des experimentellen Settings anzumerken. Da die Versuchspersonen Besucher auf dem Campusfest waren, kann man davon ausgehen, dass sie sich wenig auf die Bewegungen der einzelnen Geister konzentriert haben als darauf, selbst an diesem Tag Spaß zu haben. Des weiteren ist es auf dem Campusfest nicht möglich gewesen, die Versuchsbedingungen für jeden Probanden konstant zu halten. Unterschiede ergaben sich bereits dadurch, dass ein Spiel auf der Leinwand übertragen wurde und der zugehörige Spieler möglicherweise unter größerem Leistungsdruck stand.

Insbesondere jedoch wird der von uns gewählte Versuchsaufbau den Anforderungen an die Art des Spielens, die Yannakakis und Hallam in ihrer Spaßformel fordern, nicht gerecht. Die YH Formel wird für eine Sequenz von Spielen berechnet, da die Annahme naheliegt, dass der Spielspaß mit zunehmender Anzahl an absolvierten Spielen abnimmt. In unserer Untersuchung haben wir alle Spiele gleicher Bewertung als ein von einer Person getestetes Spiel betrachtet, denn im Rahmen eines Campusfestes ist es kaum sinnvoll, einen Menschen einer Reihe von Spielen auszusetzen.

8 Erfahrungen

Als abschließende Reflexion des ersten Projekts Pacman lässt sich sagen, dass die Projektgruppe ein gutes Ergebnis mit den Methoden der CI erzielt hat und diese auch auf dem Campusfest präsentieren konnte.

Ein wenn auch kleines Projekt schon im ersten Semester komplett fertig zu stellen, war eine ziemlich anspruchsvolle Aufgabe, die uns als junges Team sehr gefordert hat. Es war nicht nur notwendig, sich erst einmal kennen zu lernen und zu einer möglichst reibungslosen Zusammenarbeit zu finden, sondern auch etwa die Hälfte der Teilnehmerinnen musste die für das ausgewählte Projekt notwendige Programmiersprache C++ erlernen, was je nach Vorwissen unterschiedlich viel Arbeit und Zeit verschlang.

Typische Probleme im Umgang mit C++ waren vor allem der Präprozessor und die Speicherverwaltung. Durch Ersteren entstehen Situationen, bei denen Ursache (Syntaxfehler in Header Dateien) und Wirkung von Compilerfehlern weit auseinander liegen, was bei Neulingen der Sprache häufig für große Verwirrung sorgt. Die bei C++ notwendige Verwaltung von Freispeicher durch den Programmierer führt zum einen zu gängigen Fehlern (Speicherschutzverletzungen durch das Dereferenzieren ungültiger Speicheradressen und Speicherlecks), die mit Hilfe eines Debuggers recht schnell behoben werden können, zum anderen aber auch zu subtilen Fehlern, bei denen auch die erfahrenen C++ Programmierer im Team an ihre Grenzen stießen.

Mit der Wahl von C++ verbunden war auch die Einarbeitung in neue Bibliotheken, namentlich des Simple DirectMedia Layer (SDL) und der Boost Library, sowie in die damit verbundenen Konzepte von Templates und Iteratoren. Wie Joel Spolsky [Spo04] feststellt, wird die Lernkurve für Entwickler immer steiler und entsprechend zeitaufwändig die Lernphase.

Diesem Umstand ist es wohl auch zu verdanken, dass wir den ursprünglichen Zeitplan verpassten und unser ehrgeiziges Ziel, die einfachen Strategien noch zu verbessern, nicht erreichen konnten. Hybride Strategien mussten vollständig in das zweite Semester verschoben werden und die geplanten Unit Tests wurden nur für den Zufallszahlengenerator durchgeführt, da die für die Tests benötigten Spielsituationen schwer herzustellen sind. Eine Probeauswertung musste wegen Problemen mit NJam ebenfalls ausfallen, damit zum Stichtag eine stabile, lauffähige Version zur Verfügung stand. Im nächsten Semester muss die Zeitplanung detaillierter erfolgen und deren Einhaltung besser kontrolliert werden.

Ungeachtet dieser Rückschläge konnten wir wie geplant unser Projekt auf dem Campusfest der Universität Dortmund vorstellen. Diese Vorstellung wurde von allen Teilnehmerinnen als voller Erfolg gewertet. Über 200 Personen spielten an vier Computern unser Spiel probe, ohne dass es zu irgendwelchen Problemen im Ablauf oder im Programm kam. Zu Auswertungszwecken konnten wir 184 Datensätze verwenden. Die Studie kam leider zu keinem aussagekräftigen Ergebnis, wie in Abschnitt 7.2.2 beschrieben wird.

Fazit Obwohl wir unseren Zeitplan nicht einhalten konnten, haben wir im ersten Semester der Projektgruppe einiges erreicht. Auf die Erfahrungen mit den CI-Methoden werden wir beim zweiten Projekt ebenso aufbauen können wie auf Teile der bestehenden Programmierung, die als Framework auch in anderen Spielen eingesetzt werden können. Auch gewonnene Erfahrung im Umgang mit der Programmiersprache C++ sollte uns im

zweiten Projekt eine große Hilfe sein. Außerdem wissen wir nun eher, was man bei der Durchführung einer Spielstudie beachten muss.

9 Zusammenfassung

Im ersten Projekt wurde ein minimales Template-basiertes Framework für die Verwendung und den Umgang mit neuronalen Netzen entwickelt. Mit diesem Framework wurden verschiedene Netze entworfen, trainiert und in Pacman getestet. Der letzte Netzentwurf konnte erfolgreich trainiert werden, um alle 92 signifikanten Patterns zu erkennen und erfolgreich zu generalisieren. Für das Lernen wurde ein Backpropagation-Algorithmus entworfen, der sowohl mit randomisierter Patternauswahl als auch mit geordneten Patterns arbeitet. Zur Entwicklung der neuronalen Netze wurde eine Testumgebung entwickelt, die es ermöglicht, die Generalisierung und die Hit-Funktion zu überprüfen. Das abstrakte Template-basierte Framework wurde mit der Schnittstelle „MrNeuro“ an das Spiel Pacman angebunden, so dass das Framework ohne Anpassungen in andere Spiele übernommen werden kann.

Die Basis für die evolutionären Algorithmen bildet ebenfalls ein allgemeines Framework, dass die Implementierung verschiedener EA-Strategien erlaubt. Für das Spiel Pacman wurden dann drei verschiedene Strategien entwickelt. Der OnlineEA verfolgt einen steuernden Ansatz in dem jede Richtungsentscheidung mittels eines (1+1) EAs evaluiert wird. Es wird versucht, die Position von Pacman im Spiel zu approximieren und zu dieser Position wird auf dem Spielgraphen mittels des Algorithmus von Dijkstra der kürzeste Weg ermittelt und eingeschlagen. Der ProbEAGhost musste die Bewegung sowie Jagd- und Fluchtverhalten im Labyrinth erst lernen. Die gewählte Aktion beruht auf einer Wahrscheinlichkeitsverteilung, die im Lernprozess angepasst wird. Die Auswertung der Zielfunktion geschieht immer am Ende des Spiels. Dabei wird das schlechteste Individuum mit dem besten rekombiniert, mutiert und kommt dann zurück ins Spiel. Der EABBGhost basiert auf einem $(\mu + \mu)$-EA zur Optimierung pseudoboolscher Funktionen. Die Geister starten mit initialem Grundwissen, sind jedoch in der Lage, im Laufe des Spiels neue Strategien zu erlernen, um sich auf den aktuellen Gegner einzustellen.

Die Entwicklung der dr-Strategien stellte die Entwickler weniger vor ein algorithmisches als vor ein psychologisches Problem: Wie sollte die Strategie eines Gegners aussehen, so dass sie als schwierig aber nicht unschlagbar empfunden wird? Wie sieht eine Strategie aus, mit der sich ein Mensch in einem Labyrinth auf die Jagd machen würde, wenn er keine globale Information sondern nur seine eingeschränkte, normale Sicht hätte? Die Verwendung von Gestalten aus Sage, Literatur und Film sowie zweier extrem simpler Strategien erschien uns erfolgversprechend und konnte zum gewünschten Ergebnis gebracht werden.

Im Hinblick auf die Ergebnisse bezüglich des Spielspaßes, die wir bei der Umfrage zu den subjektiven Eindrücken der Testspieler gewinnen wollten, lässt sich abschließend nur feststellen, dass hier wohl viele bereits erwähnte Probleme ein verwertbares Ergebnis zunichte gemacht haben.

10 Ausblick

Wie in den vorherigen Kapiteln detailliert beschrieben, ist das erste Projekt im Großen und Ganzen sehr gelungen. Es wurden mit den CI-Methoden der neuronalen Netze und der evolutionären Algorithmen erfolgreich Spielstrategien für Geister im Spiel Pacman entwickelt. Basierend auf diesen Erfahrungen ergaben sich viele neue Ideen und Herausforderungen für das zweite Projekt. Unabhängig von den angewendeten CI-Methoden wird die Zielsetzung im Weiteren die Entwicklung von Spielstrategien unterschiedlicher Einheitentypen sein. Im Spiel Pacman existiert nur ein Geistertyp, der von jeder Gruppe (beschrieben in den Kapiteln 4, 5 und 6) unterschiedlich umgesetzt worden ist. Es wäre zu erörtern, wie problemlos verschiedene Strategien abhängig von den Einheitentypen entwickelt werden können. Diese Fragestellung steht im engen Zusammenhang mit einer Bibliothek von CI-Methoden, welche – vielleicht sogar spielunabhängig – zur einfachen Erzeugung von künstlicher Intelligenz in Spielen eingesetzt werden kann. Als Spielgenre wird im zweiten Projekt wahrscheinlich das Genre der Strategiespiele als Ausgangspunkt unserer Arbeit stehen. Strategiespiele zeichnen sich einerseits durch ihre Vielfalt an Einheiten aus und andererseits durch die Interaktion der Einheiten eines bestimmten Spielers untereinander. Einheiten sollen sich im Hinblick auf eine Gesamtstrategie des Spielers geeignet verhalten und ein übergeordnetes Spielziel verfolgen. Dies ist nur zu Erreichen, wenn Einheiten untereinander in Interaktion stehen oder von einer übergeordneten Instanz koordiniert werden. Des Weiteren wäre es wünschenswert, wenn Einheiten in bestimmten Situationen ein Schwarmverhalten aufweisen würden, um in einer Gruppe ein Teilziel zu verfolgen.

Die Koordination und das weitere Vorgehen stehen noch für das nächste Projekt zur Diskussion. Es wäre möglich, das Projekt in kleinere Stufen zu gliedern, um schrittweise erst einzelne Einheitentypen unabhängig voneinander zu behandeln und dann als nächsten Schritt die Interaktion zwischen Einheiten beziehungsweise die Entwicklung einer Gesamtstrategie zu implementieren. Allerdings könnte es einen Konflikt zwischen den Entwicklungen von Einzelstrategien und einer Gesamtstrategie geben, so dass im Vorfeld geklärt werden muss, in wie weit die Entwicklungen unabhängig voneinander realisiert werden können.

Nachdem im ersten Projekt viele nützliche Erfahrungen bei der Anwendung von CI-Methoden in Spielen entstanden sind, wird im zweiten Projekt konsequent daran weiterentwickelt.

Die Gruppe, die sich mit neuronalen Netzen auseinander gesetzt hat, konnte erfolgreich im ersten Schritt den Ansatz des supervised-learning (überwachtes Lernen) einsetzen. Allerdings wird mit zunehmender Komplexität der Spielinformationen das überwachte Lernen immer schwieriger umzusetzen, weil eine große Menge von Trainingsdaten erzeugt werden muss. Alternativ wird im Weiteren der Ansatz des unsupervised-learning (unüberwachtes Lernen) verfolgt. Hierbei wird keine Menge von Trainingsdaten vorgegeben, sondern das neuronale Netz lernt selbständig im Spiel hinzu.

Evolutionäre Algorithmen wurden eingesetzt, um sowohl einen selbstlernenden Ansatz zu verfolgen, als auch lokal das Verhalten eines Geistes zu steuern. Für das zweite Projekt werden diese Ansätze weiter vertieft. Ergänzend zu den neuronalen Netzen wäre auch der Einsatz von genetischer Programmierung möglich. Hierbei handelt es sich in der klassischen Variante um eine spezielle Ausprägung eines evolutionären Algorithmus, der in der Lage ist eine Klassifikation durchzuführen, in dem auf einer Population von Bäumen ge-

arbeitet wird. In der Variante der linearen genetischen Programmierung wird hingegen auf einer Population von Computerprogrammen gearbeitet, die durch eine Sequenz von Anweisungen einer imperativen Programmiersprache beschrieben werden.

Eine weitere interessante Möglichkeit, um Spielstrategien zu entwickeln, wäre die Kombination von neuronalen Netzen und evolutionären Algorithmen. Es wäre vorstellbar, dass ein evolutionärer Algorithmus eine Population von neuronalen Netzen beherbergt und diese so mutiert, dass passende neuronale Netze für eine Problemstellung erzeugt werden.

Im ersten Projekt hat sich die Projektgruppe nur auf neuronale Netze und evolutionäre Algorithmen konzentriert. Eine weitere interessante Methode im Bereich der Computational Intelligence ist die Fuzzy-Logik. Diese unscharfe Logik macht es möglich, aus einer verbalen Problembeschreibung Regelsysteme zu schaffen. Diese Regelsysteme können sehr gut zur Entscheidungsfindung oder zur Steuerung genutzt werden. Übertragen auf die Strategieentwicklung in Computerspielen wäre es denkbar, dass wir zum Beispiel unscharfe Verhaltensentscheidungen treffen können oder eine Bewertung der Spielsituation basierend auf einem Regelsystem ermitteln können.

Die deterministische Gruppe hat im ersten Projekt den Auftrag gehabt, gute Computergegner zu entwickeln, die als Vergleich für die von den CI-Methoden gesteurten Geistern dienen sollten. Grund dafür war primär die Zielsetzung, dass wir Geister in Pacman einer Spielspaßbewertung nach Yannakakis und Hallam unterziehen wollten. Für das zweite Projekt steht im wesentlichen eine Machbarkeitsanalyse der CI-Methoden in komplexeren Spielen im Vordergrund, wodurch hier eine deterministische Gruppe – zumindest in dem Ausmaße wie im ersten Projekt – nicht erforderlich ist. Es bestehen aber Überlegungen, im zweiten Projekt den CI-Methoden klassische Lernverfahren (zum Beispiel Entscheidungsbäume) entgegen zu stellen, die in Kombination mit deterministischen Strategien eingesetzt werden könnten.

Abschließend bleibt zu berichten, dass viele Ideen und Herausforderungen für das nächste Projekt im Raum stehen. Die Projektgruppe fühlt sich weiterhin hoch motiviert, die Anwendung von CI-Methoden zur Erzeugung von spielspaßsteigernden NPCs zu untersuchen.

A Evolutionäre Algorithmen

A.1 Einleitung

Evolutionäre Algorithmen (EA) sind randomisierte Suchheuristiken. Ihr Einsatz bietet sich besonders bei komplizierten, z.B. mehrdimensionalen, Optimierungs- oder Suchproblemen und anderen Aufgaben an, bei denen klassische nummerische oder analytische Lösungsverfahren scheitern.
Sie züchten Problemlösungen, indem sie vorhandene Lösungen zu neuen, hoffentlich besseren, Lösungen kombinieren. Dazu greifen sie Ideen auf, die von der Evolutionstheorie der Natur inspiriert sind.
Evolutionäre Algorithmen haben den großen Vorteil, vorurteilsfrei nach Lösungen zu suchen. Ein EA braucht keine Kenntnisse über die Problemstruktur, es reicht wenn Lösungen repräsentiert, bewertet und neue Lösungen erzeugt werden können [Kel00]. Diese Eigenschaft wird *Black–Box–Optimierung* genannt.

A.2 Funktionsweise evolutionärer Algorithmen

A.2.1 Allgemeiner Aufbau eines evolutionären Algorithmus

Aufgrund der Inspiration durch den Evolutionsprozess der Natur wurden viele der im Zusammenhang mit evolutionären Algorithmen auftauchenden Begriffe direkt aus der Biologie übernommen. Evolutionäre Algorithmen operieren auf einer Multimenge von *Individuen*, die *Population* genannt wird. Ein Individuum ist Element des Suchraums des zu lösenden Problems, stellt also eine mögliche Lösung dar. Seine Kodierung, *Repräsentation* genannt, wird von der Struktur des Suchraums bestimmt. Beispiele für mögliche Suchräume sind $\mathbb{B}^n$, $\mathbb{R}^n$, S_n und Bäume.
Der Suchraum S auf dem ein evolutionärer Algorithmus arbeitet muß nicht zwingenderweise mit dem Suchraum A des zu lösenden Problems übereinstimmen. Die Individuen sind also Elemente von S, der *Genotypraum* genannt wird. A bezeichnet man als *Phänotypraum* und die Abbildung $S \rightarrow A$ als *Genotyp-Phänotyp-Mapping*.
Um die gefundenen Lösungen zu verbessern, werden iterativ aus den alten Individuen neue Individuen (*Nachkommen*) erzeugt, bis eine vorgegebene Güte oder eine maximale Anzahl von Iterationen erreicht ist. Ein solcher Iterationsschritt heißt *Generation*. Für die Erzeugung der Nachkommen greifen evolutionäre Algorithmen auf Methoden zur Reproduktion zurück, wie sie die Natur vormacht: Neue Individuen werden aus mehreren alten Individuen erzeugt (*Rekombination* oder auch *Crossover*) oder zufällig verändert (*Mutation*).
Die *Fitnessfunktion* bewertet Individuen hinsichtlich der Qualität der Lösung, die sie repräsentieren. Der Wert der Fitnessfunktion kann in die *Selektion* eingehen, z.B. bei der *fitnessproportionalen Selektion*. Die *Selektion* wählt an zwei Stellen eines evolutionären Algorithmus Individuen aus: Zum einen die Individuen aus der alten Population, aus denen Nachkommen erzeugt werden. Zum Anderen aus den Nachkommen und ggf. der alten Population die Individuen der Folgepopulation.
Fügt man all dies zusammen, ergibt sich der in Abbildung 21 dargestellte Ablauf eines evolutionären Algorithmus. Im Folgenden werden die einzelnen Module detailierter vorgestellt.

```
t:=0
Initialisierung der Anfangspopulation P(0)
Bewertung der Population P(0)
WHILE Terminierungskriterium nicht erfüllt
  Selektiere die Eltern
  Erzeuge aus den Eltern Nachkommen
          mittels Mutation und ggf. Rekombination
  Bewerte die Nachkommen
  Selektiere die Nachfolgepopulation P(t+1)
  t:=t+1
END WHILE
```

Abbildung 21: Pseudocode eines evolutionären Algorithmus

A.2.2 Initialisierung

Die Initialisierung erzeugt die Individuen der Ausgangspopulation. Es bietet sich an, diese zufällig gleichverteilt zu wählen, schließlich wollen wir randomisiert suchen, denn wir haben ja keine Ahnung, wie gute Lösungen aussehen.
Wenn durch Vorkenntnisse doch vielversprechende Individuen bekannt sind, können auch diese als Anfangspopulation genutzt werden. Es sei jedoch darauf hingewiesen, dass ein EA dann schnell gegen ein lokales Optimum konvergieren kann [Eng02].

A.2.3 Fitnessfunktion

Die minimale Anforderung an eine Fitnessfunktion ist, dass sie den Vergleich zweier Individuen erlaubt, um eine Entscheidung darüber zu treffen, welches der beiden Individuen das bessere ist.
Beschränkt man sich allein auf diese Minimalanforderung, spricht man von *relative* oder *competitive fitness evaluation*. Gerade in Spielen kommt es häufig zu einer *Stein–Schere–Papier* genannten Situation, in der bei drei Strategien A, B und C Strategie A von B, B von C und C von A geschlagen wird.

Ansonsten ist eine Fitnessfunktion jedoch eine Abbildung $f(s) : S \to \mathbb{R}, s \in S$, häufig eingeschränkt auf $\mathbb{R}_+$[8], bei der jedes Individuum einzeln bewertet wird. Die Art der Bewertung ist dabei stark von dem zu lösenden Problem abhängig, ein praktisches Beispiel wird in Abschnitt A.4 vorgestellt.
Betrachtet man ein Optimierungsproblem, enthält dieses ja eine Zielfunktion, die es zu minimieren oder maximieren gilt, woraus sich bei einfachen Problemen auf eine natürliche Art und Weise eine Fitnessfunktion ergibt, wie man sich am Problem des Handelsreisenden (TSP) leicht klar macht: Aufgabe ist es, eine kostengünstigste Rundreise durch gegebene Städte zu finden, z.B. die Route mit den geringsten Benzinkosten. Zielfunktion ist dann die Minimierung der Benzinkosten, die Fitness eines Individuums also die Benzinkosten, die es verursacht.

[8] Voraussetzung für die fitnessproportionale Selektion

Für theoretisch motivierte Fitnessfunktionen sei an dieser Stelle auf das Skript von Thomas Jansen verwiesen [Jan04].

A.2.4 Selektion

Die Selektion wählt wie Eingangs erwähnt an zwei Stellen Individuen aus, entweder als Eltern für die Reproduktion oder als die Individuen der Folgepopulation. Die dabei jeweils eingesetzten Selektionsoperatoren können durchaus unterschiedlich sein.

Die einfachste Variante, Individuen einfach zufällig gleichverteilt zu wählen, wird *uniforme Selektion* genannt.
In aller Regel will man jedoch gute Individuen bevorzugen, dazu liegt es nahe, die Selektion direkt an der Fitness zu orientieren.
Bei der *fitnessproportionalen Selektion* wird ein Individuum $s \in S$ mit der Wahrscheinlichkeit $f(s)/\sum_{x \in S} f(x)$ gewählt. Doch gerade diese starke Abhängigkeit vom Fitnesswert erweist sich als Nachteil: Bei großen Unterschieden in den Fitnesswerten verhält sich die fitnessproportionale Selektion fast deterministisch, es wird mit sehr großer Wahrscheinlichkeit das Individuum mit dem größten Fitnesswert ausgewählt und das u.U. immer wieder. Sind die Unterschiede in den Fitnesswerten dagegen sehr klein, unterscheidet sich die fitnessproportionale Selektion kaum von der uniformen Selektion.
Die *Boltzmann–Selektion* versucht diesen Nachteil zu umgehen, indem es die Idee des simulated annealing aufgreift und statt der Funktionswerte $f(x)$ den Ausdruck $e^{\mathrm{f(x)/T}}$ verwendet. T ist dabei die aus dem simulated annealing bekannte $Temperatur$. Man beginnt mit einem großen Wert für T und senkt diesen von Generation zu Generation. Erst bei niedrigen Werten für T, also bei späteren Generationen, gewinnt der Wert der Fitnessfunktion an Einfluß auf den Selektionsprozeß.
Bei der *Schnitt-Selektion* werden im Gegensatz zu den bisher vorgestellten Verfahren nicht eines sondern mehrere Individuen ausgewählt, nämlich eine vorher festgelegte Anzahl von Individuen mit der größten Fitness.

A.2.5 Selektion der Folgepopulation

Die Auswahl der Individuen für die Folgepopulation kann natürlich ebenfalls mit den zuvor vorgestellten Operatoren erfolgen. Es ergibt sich jedoch die Frage aus welcher (Multi-)Menge die Individuen gewählt werden.
Bei der *Plus-Selektion* werden aus der Vereinigung der alten Individuen und der Nachkommen deterministisch die μ Individuen als neue Population ausgewählt, die den größten Fitnesswert haben. Dies wird als $(\mu + \lambda)$–Selektion notiert, wobei μ für die Größe der Population und λ für die Anzahl der Nachkommen steht.
Bei der *Komma-Selektion* werden die Individuen der neuen Population ausschließlich aus den Nachkommen ausgewählt, die alte Population wird komplett verworfen. Das impliziert natürlich, dass die Anzahl der Nachkommen mindestens so groß sein muß wie die Größe der Population. Die Auswahl geschieht wieder deterministisch, es werden die μ Nachkommen mit der größten Fitness gewählt. Analog zur Plus–Selektion wird die Komma-Selektion auch (μ, λ)–Selektion notiert.

A.2.6 Variation

Die Variation erzeugt mit Hilfe von Zufallselementen aus den vorhandenen Individuen einer Generation neue Individuen für die nachfolgende Generation. Je nachdem, ob ein neues Individuum aus einem oder aus mehreren Individuen der alten Population erzeugt wird, spricht man von *Mutation* oder *Rekombination*. Nicht alle evolutionären Algorithmen verwenden die Rekombination, die Mutation wird dagegen immer eingesetzt, ggf. nach der Rekombination.

A.2.7 Rekombination

Bei der Rekombination, auch *Crossover* genannt, wird aus mehreren, meist jedoch zwei, Individuen ein Nachkomme erzeugt. Der Nachkomme wird dabei aus Teilen seiner Eltern zusammengesetzt.
Bei Repräsentationen die sich als Vektoren auffassen lassen (z.B. im $\mathbb{B}^n$ oder $\mathbb{R}^n$) teilt man dazu einfach den Vektor: Beim *k–Punkt–Crossover* werden k verschiedene Position zufällig gleichverteilt ausgewählt und die Vektoren der Eltern an diesen Stellen zerlegt. Der Nachkomme wird dann alternierend aus den $k+1$ Teilen zusammengesetzt. Es ist üblich, k sehr klein zu wählen, häufig ist $k = 1$ oder $k = 2$.
Beim *uniformen Crossover* wird aus zwei Eltern für jede Komponente des Vektors des Nachkommen mit Wahrscheinlichkeit jeweils $1/2$ die Komponente des einen oder des anderen Elter kopiert. Sowohl uniformes als auch k-Punkt-Crossover, bei denen die Teile der Eltern unverändert kopiert werden, werden auch *diskrete* Rekombination genannt.
Dagegen steht im $\mathbb{R}^n$ das *arithmetische* Crossover, bei dem der Nachkomme y als gewichtete Summe $y = \alpha_1 x_1 + ... + \alpha_n x_n$ aus den n Eltern $x_1, ..., x_n$ berechnet wird. Sind alle Gewichte $\alpha_i = 1/n$, spricht man von *intermediärer* Rekombination, der Nachkomme ist dann der Schwerpunkt der Eltern.

A.2.8 Mutation

Die Mutation verändert ein Individuum zufällig, wobei der Grad der Veränderung wie in der Natur im Allgemeinen recht klein gehalten wird. Die Art der Veränderung hängt stark von der Darstellung ab.
Typische Mutationsoperatoren im $\mathbb{B}^n$ sind die *Standardbitmutation* und die *1–Bit–Mutation*. Bei der *Standardbitmutation* wird für jedes Bit mit einer Wahrscheinlichkeit p_m ein Fehler gemacht, wobei p_m in der Praxis relativ klein gehalten wird. Oft verwendet man bei einem Bitstring der Länge n $p_m = 1/n$, so dass im Durchschnitt nur ein Bit verändert wird.
Bei der *1–Bit–Mutation* wird immer nur genau ein zufällig ausgewähltes Bit verändert.

Im $\mathbb{R}^n$ bieten sich ähnliche Mutationsoperatoren wie im $\mathbb{B}^n$ an, wobei eine Komponente des Vektors $x \in \mathbb{R}^n$, der ein Individuum repräsentiert, die Rolle eines Bits im $\mathbb{B}^n$ einnimmt. Das Verfälschen der einzelnen Komponenten geschieht dann durch Addition eines Mutationsvektors $m \in \mathbb{R}^n$, dessen Komponenten bei einfachen evolutionären Algorithmen in der Regel alle gleich sind.

Die vorgestellten Operatoren sind nur wenige Beispiele, der Fantasie sind hier eigentlich keine Grenzen gesetzt.

A.2.9 Parameter

Evolutionäre Algorithmen lassen sich über verschiedenste Parameter beeinflussen, einige wurden bereits in den vorangegangenen Abschnitten erwähnt. Legt man diese Parameter vor dem Start des evolutionären Algorithmus fest und ändert sie danach nicht mehr, spricht man von *statischen Parametereinstellungen*. Diese sehr einfache Methode wird in der Praxis häufig eingesetzt [Jan04], muß aber nicht optimal sein: Parameterwerte die zu Beginn der Ausführung angemessen waren, können im weiteren Verlauf unpassend werden, wenn sich der EA der optimalen Lösung nähert. Es kann also sinnvoll sein, Parameter abhängig von der Anzahl durchlaufener Generationen zu verändern. Dieses Prinzip, Parameter des evolutionären Algorithmus als Funktion der Zeit zu definieren, wird *dynamische Parametereinstellung* genannt.

Der Einsatz von *dynamischer Parametereinstellung* ist dennoch bei evolutionären Algorithmen eher unüblich, da es wesentlich lohnenswerter ist, nicht nur die Zeit sondern auch andere Informationen in die Veränderung der Parameter einfließen zu lassen, wie zum Beispiel die Fitnesswerte und die Verteilung der Population im Suchraum. Man spricht dann von *adaptiver Parametereinstellung*. Dabei kann ein Feedback Mechanismus benutzt werden, der den evolutionären Prozeß überwacht und Parametersets danach beurteilt, wie gut sie in jedem Schritt den Fitnesswert der Population verbessern [BFM00b].

Alle bisherigen Verfahren zur Anpassung der Parameter arbeiten deterministisch und der Benutzer hat die Kontrolle über die Veränderung der Parameter[9].

Bei der *Selbstadaption* unterliegt schließlich auch die Menge der Parameter, die den evolutionären Algorithmus steuern, dem evolutionären Prozess. Jedem Individuum repräsentiert dann wie gehabt einen Punkt des Suchraumes, enthält darüber hinaus aber auch ein Set von Parametern. Bezeichnet man den Raum der möglichen Parameter mit $\mathbb{P}$, so arbeitet der EA auf dem Kreuzprodukt $\mathbb{S} \times \mathbb{P}$ aus dem Suchraum $\mathbb{S}$ und dem Parameterraum $\mathbb{P}$, wobei die Fitnessfunktion weiterhin eine Abbildung $f(s) : S \rightarrow \mathbb{R}, s \in S$ ist.
Bei der Erzeugung der neuen Population werden dann zuerst mit speziellen Operatoren neue Parameter erzeugt und dann mit diesen Parametern die neuen Individuen.

A.3 Varianten evolutionärer Algorithmen

Alle modernen evolutionären Algorithmen folgen im Wesentlichen dem in Abschnitt A.2 vorgestellten Ablauf. Da jedoch ab den 1960er Jahren unterschiedliche Personen unabhängig voneinander verschiedene Varianten evolutionärer Algorithmen entwickelt haben, sind deren Bezeichnungen immer noch weit verbreitet, weshalb auf die Unterschiede an dieser Stelle kurz eingegangen werden soll.

Die *Genetischen Algorithmen* gehen auf John Holland zurück und arbeiten auf Bitstrings, verwenden also den Suchraum $\mathbb{B}^n$. Nachkommen werden mit Hilfe der Rekombination erzeugt, Mutation spielt, wenn überhaupt eingesetzt, nur eine untergeordnete Rolle.
Die *evolutionäre Programmierung* wurde in den 60er Jahren von Larry Fogel entwickelt und arbeitet auf dem Raum der endlichen Automaten. Evolutionäre Programmierung verwendet ausschließlich Mutation als Operator für die Variation.

[9] Bei statischen Parametern ist dies offensichtlich, bei dynamischer und adaptiver Parametereinstellung erfolgt der Einfluß auf die Veränderung über die Vorgabe der gewünschten Funktionen

Evolutionsstrategien gehen auf Peter Bienert, Ingo Rechenberg und Hans–Paul Schwefel zurück, die diese 1964 als Studenten in Berlin entwickelten. Evolutionsstrategien operieren häufig auf dem Suchraum $\mathbb{R}^n$ und haben schon früh nicht–statische Parametereinstellungen verwendet. Die Selbstadaption wurde im wesentlichen an Evolutionsstrategien entwickelt [Jan04].

Genetische Programmierung unterscheidet sich deutlich von anderen evolutionären Algorithmen dadurch, dass der EA auf einem Computerprogramm arbeitet. D.h. jedes Individuum der Population entspricht einem ausführbaren Programm, das im Laufe der Evolution immer weiter verändert wird. Als Konsequenz davon haben Individuen bei der genetischen Programmierung in der Regel keine feste sondern eine variable Länge.
Als Programmiersprache dient oft Lisp, generell ist es jedoch schwierig für normale Programmiersprachen Mutations- und Rekombinationsoperatoren zu finden, die zu syntaktisch korrekten Programmen führen. Einen Ausweg bietet die Kodierung der Programme als Bäume, bei denen die Variationsoperatoren als zufällige Veränderung oder dem Austauschen von Teilbäumen implementiert werden können. Das in Kapitel A.4 vorgestellte Beispiel greift diese Idee auf.

A.4 Praktisches Beispiel: Genetische Programmierung zur Erzeugung eines Wegfindungsalgorithmus

Rick Strom beschreibt in seinem Artikel *Evolving Pathfinding Algorithms using Genetic Programming* [Str06] die Generierung eines Wegfindungsalgorithmus für Strategiespiele mit Hilfe der genetischen Programmierung. Seine Arbeit soll hier als Beispiel für einen evolutionären Algorithmus dienen, schließlich lautet das Thema unserer Projektgruppe Computational Intelligence *in Games*. Das Augenmerk liegt dabei nicht auf einem optimalen Algortihmus, sondern darauf, dass sich der erzeugte Algorithmus aus Sicht eines menschlichen Spielers überzeugend und natürlich verhält.

A.4.1 Szenario

Rick Strom benutzt *Hampton*[10], eine von ihm entwickelte Laufzeitumgebung, die er unter der GPL veröffentlicht hat. Diese simuliert eine zweidimensionale Karte aus 40×40 rechteckigen Feldern, auf der die erzeugten Algorithmen getestet werden können. Ein Zug ist die Bewegung von einem Feld zu einem anderen und je nach Art des Zielfeldes mit Kosten von 0–9 Strafpunkten verbunden, deren Summe in die Fitnessfunktion eingehen. Es gibt Felder, die nicht passierbar sind und Züge außerhalb der Karte sind nicht erlaubt.
Abbildung 22 zeigt einen Screenshot von Hampton: Oben links (*Map*) ist die Visualisierung der Karte zu sehen. Die dünne rote Linie entspricht dem Pfad des aktuell fittesten der erzeugten Algorithmen, die blauen Linien zeigen Pfade anderer Lösungen. Die dicke blaue Linie repräsentiert einen Fluß und die breite rote Linie einen Bereich, der von Feinden besetzt ist.
Unten links befindet sich die Konsole (*console*), in der Daten zum aktuellen Fortschritt der Ausführung ausgegeben werden. Am rechten Bildrand *Best solution* ist der expandierte Funktionsbaum des aktuell fittesten Algorithmus zu sehen.

[10] http://sourceforge.net/projects/hampton

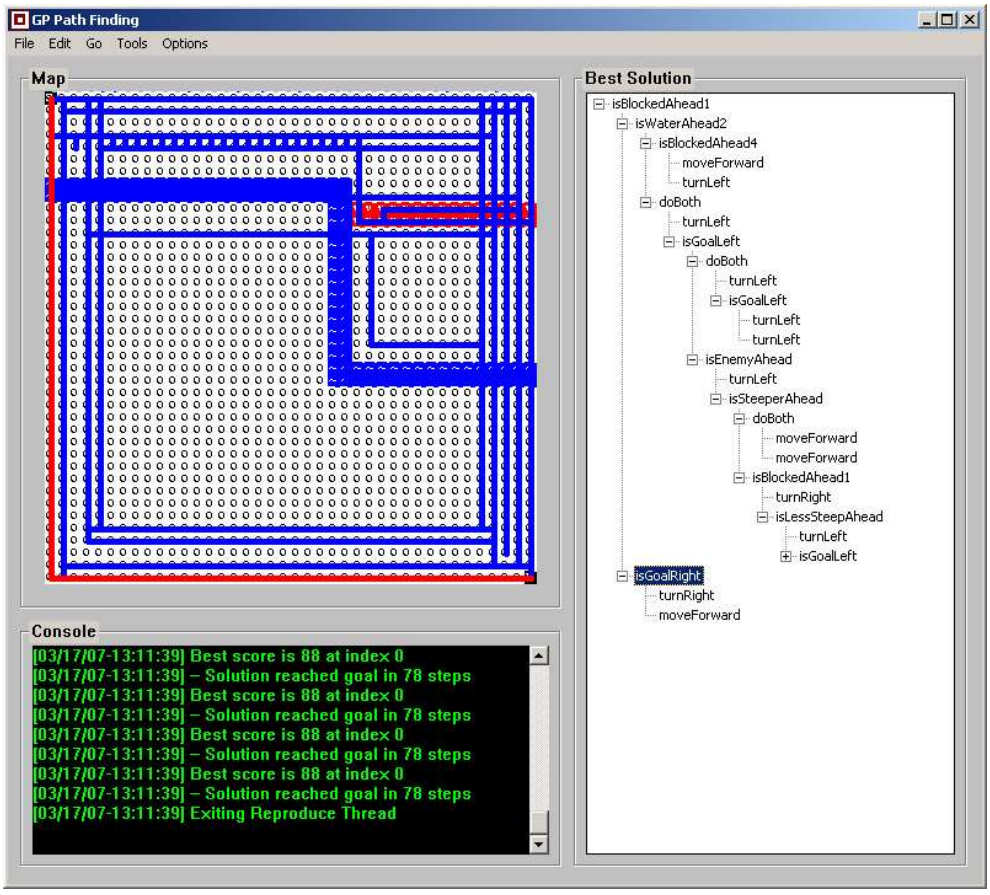


Abbildung 22: Laufzeitumgebung Hampton. Oben links die Karte, darunter die Konsole und am rechten Rand der Funktionsbaum des aktuell fittesten Algorithmus.

A.4.2 Fitnessfunktion

Für Programme die das Ziel erreicht haben, summiert die Fitnessfunktion $f(s)$ die Anzahl n der Züge z_i bis zum Erreichen des Ziels und die dabei gesammelten Strafpunkte auf. Das fitteste Programm ist dann das Programm mit der niedrigsten Bewertung.
Den Programmen, die daran scheitern das Ziel zu erreichen[11], wird ein so hoher Wert zugeordnet, dass sie auf jeden Fall schlechter bewertet werden, als die schlechtesten Programme, die das Ziel erreicht haben. Zu diesem Wert wird noch die Entfernung d_x, d_y der letzten Position zum Ziel addiert.
Damit ergibt sich für $f(s)$ folgende Formel:

$$f(s) = \begin{cases} \sum_{i=1}^{n} 1 + \text{kosten}(z_i) & \text{Programm } s \text{ hat Ziel erreicht} \\ 20000 + d_x + d_y & \text{Programm } s \text{ hat Ziel nicht erreicht} \end{cases}$$

[11] Ein Programm gilt als gescheitert, wenn es das Ziel nach 1000 Schritten nicht erreicht hat

A.4.3 Programmiersprache

Rick Strom benutzt eine von ihm selbst definierte Sprache aus Funktionen und Terminalen zur Beschreibung der Programme. Tabelle 8 listet die drei Terminale der Sprache auf, das sind die Bewegungsmöglichkeiten auf der Karte: Gehe einen Schritt vorwärts, Drehung nach links oder Drehung nach rechts.
Die neun Funktionen in Tabelle 9 stellen den erzeugten Algorithmen Möglichkeiten zur Verfügung, Fragen an die Laufzeitumgebung zu stellen, um sich über die aktuelle Situation zu informieren. Alle Funktionen geben einen boolschen Wert zurück, an einer Funktion kann der Programmablauf also immer in zwei Richtungen verzweigen. Die gezüchteten Algorithmen lassen sich also als binäre Bäume darstellen, die von einem Interpreter in der Laufzeitumgebung ausgeführt werden. Dazu traversiert der Interpreter den Baum, beginnend mit der Wurzel, und verzweigt an Funktionen entsprechend des Ergebnisses, bis ein Terminal erreicht ist. Danach beginnt die Traversierung erneut bei der Wurzel. Das wird solange wiederholt, bis das Ziel erreicht ist.

Terminal	Beschreibung
moveForward	Bewegung um einen Schritt nach vorn
turnLeft	Drehung um 90 Grad nach links
turnRight	Drehung um 90 Grad nach rechts

Tabelle 8: Terminale der Programmiersprache von Hampton

Funktion	Beschreibung
isWaterAhead$X$	Testet, ob sich X Rechtecke vor der aktuellen Position Wasser befindet
isEnemyAhead	Testet, ob sich vor der aktuellen Position ein Feind befindet
isBlockedAhead$X$	Testet, ob sich X Rechtecke vor der aktuellen Position ein Feld befindet, das nicht betreten werden kann
isSteeperAhead	Testet, ob das nächste Rechteck höhere Kosten verursacht, als die aktuelle Position
isLessSteepAhead	Testet, ob das nächste Rechteck niedrigere Kosten verursacht, als die aktuelle Position
isGoal{Left, Right, Ahead, Behind}	Testet, ob sich das Ziel in der angegebenen Richtung befindet
doBoth	Führt beide Kinder aus

Tabelle 9: Funktionen der Programmiersprache von Hampton

Eine besondere Rolle spielt die Funktion *doBoth*. Trifft der Interpreter beim Traversieren des Baumes auf einen *doBoth* Knoten, werden beide Unterbäume ausgeführt. Die Konvention dabei ist, dass eines der beiden Kinder ein Terminal ist. Dies gibt dem Algorithmus quasi ein Gedächnis, er kann aus einer bestimmten Situation heraus einen Zug machen und danach weitere Funktionen ausführen.
Da der Zustand nicht in einer Variablen gespeichert wird, sondern sich allein aus der aktuellen Position im Programmcode ergibt, geht die gespeicherte Information verloren,

sobald die Traversierung des Baumes erneut bei der Wurzel beginnt. Es wäre mit Sicherheit eine interessante Erweiterung, den Programmen persistenten Speicher in Form eines Registersatzes oder eines Stacks zur Verfügung zu stellen.

A.4.4 Evolution

Die Ausgangspopulation besteht aus zufällig erzeugten Individuen, ihre Größe kann zu Beginn beliebig gewählt werden und bleibt dann konstant. Bei der Erzeugung der nachfolgenden Population kommen sowohl Mutation als auch Rekombination zum Einsatz, wobei nur aus den zwei besten Programmen Nachkommen erzeugt werden. Alle anderen Individuen werden aus der Population entfernt.
Für die Rekombination wird in beiden Eltern je ein Knoten zufällig ausgewählt. Die beiden Nachkommen sind dann Kopien der Eltern, wobei die Unterbäume deren Wurzeln die ausgwählten Knoten sind, zwischen den beiden Nachkommen vertauscht werden.
Für die Mutation eines Nachkommen wird ein Knoten zufällig gewählt und aus dem Programm entfernt. An seine Stelle tritt ein neuer, zufällig erzeugter Teilbaum, dessen Tiefe auf 5 begrenzt ist.

A.4.5 Ergebnis

Obwohl die Ausgangspopulation aus zufällig erstellten Programmen besteht, entstehen bereits nach recht kurzer Zeit brauchbare Algorithmen. Interessanterweise werden die Programmbäume nie besonders tief, obwohl es mit Ausnahme der Begrenzung bei durch Mutation entstandenen Teilbäumen keinen Mechanismus gibt, der die Tiefe der Bäume begrenzt.

Leider ist die Laufzeitumgebung Hampton in der aktuell verfügbaren Version sehr instabil, was eigene Experimente auf der Basis von Rick Stroms Arbeit sehr erschwert.

B Fuzzy-Systeme

B.1 Einleitung

Fuzzy ist der englische Begriff für „unscharf“, dem gegeüber steht der englische Begriff „crisp“ [DD97]. Die Idee der Fuzzy-Logik ist die Abbildung von nicht zu präzisierenden umgangssprachlichen Begriffen in eine mathematische Logik.

Betrachten wir folgendes Beispiel: Ein Mensch mit einer Körpergröße von 190 cm ist groß. Ist ein Mensch, der 189 cm groß ist nun klein?

Bei klassischer zweiwertiger Logik wäre die Antwort auf diese Frage „ja“, während die meisten Menschen „nein“ antworten würden. Jan Lukasiewicz, geb. am 21. Dezember 1878 in Lemberg, gestorben am 13. Februar 1956 in Dublin, war ein polnischer Philosoph, Mathematiker und Logiker. Er entwarf 1920 die erste mehrwertige Logik um diesen Problem entgegenzuwirken[12]. Später erfand Lotfi Asker Zadeh, geb. am 4. Februar 1921 in Baku (Aserbaidschan), die Fuzzy-Logik[13].

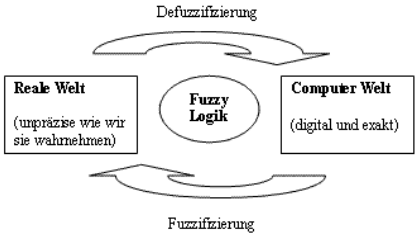


Abbildung 23: Defuzzifizierung bei der Übertragung von Daten in die präzise Computerwelt gegenüber Fuzzifizierung bei der Übertragung aus der Computerwelt in vage umgangssprachliche Begriffe.

Bei der Übertragung von Daten aus der präzisen zweiwertigen Computerwelt in die von Menschen wahrgenommene Welt werden diese fuzzifiziert. Dem entgegengesetzt steht der Prozess der Defuzzifizierung, wenn für Konzepte, die in unserer Welt nicht präzise sind, präzise Werte definiert werden.[Lee05]

Die Fuzzy-Logik stellt Konzepte für die Repräsentation von unscharfen Aussagen und den Umgang mit diesen bereit.

B.2 Fuzzy-Logik

B.2.1 Fuzzy-Menge

Bevor eine Definition für Fuzzy-Mengen angegeben wird, betrachten wir eine Definition einer herkömmlichen Menge: $M = \{x|x > 1\}$. Diese kann auch in der Form $M = \{x|\chi(x) = 1\}$ definiert werden. Dabei ist χ_M mit

[12] siehe `http://de.wikipedia.org/wiki/Jan_%C5%81ukasiewicz`
[13] siehe `http://de.wikipedia.org/wiki/Lotfi_Zadeh`

$\chi_M(x) = \begin{cases} 1 & \text{für } x > 1 \\ 0 & \text{sonst} \end{cases}$
die charakteristische Funktion der Menge M und beschreibt M eindeutig.

Analog wird eine Fuzzy-Menge definiert: Die Abbildung $F : X \to [0, 1]$ heißt Fuzzy-Menge oder Zugehörigskeitsfunktion (engl. membership funktion), wobei X das Universum ist. $F(x)$ ist der Zugehörigkeitsgrad zu F für ein $x \in X$. [Thi99] Je nach Zugehörigkeitsgrad kann eine Fuzzy-Menge eine beliebige Form haben (siehe Abbildung 24), wie z.B. ungefähr 4 oder ungefähr 6, die Fuzzy-Zahl 4 oder auch das Fuzzy-Intervall von 4 bis 6.

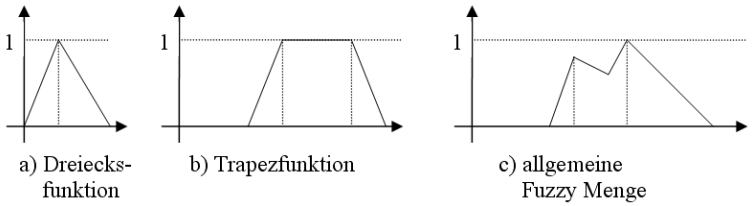


Abbildung 24: Beispiele für Fuzzy-Mengen.

Insbesondere existieren die leere Fuzzy-Menge $\forall x \in X : F(x) = 0$, die kein Element enthält, sowie die universelle Fuzzy-Menge $\forall x \in X : F(x) = 1$, die alle Elemente enthält. [Rud07]

Im Umgang mit Fuzzy-Mengen werden folgende Begrifflichkeiten verwendet.

- Träger $supp(F) := \{x \in X | F(x) > 0\}$ (engl. support)
- Kern $ker(F) := \{x \in X | F(x) = 1\}$
- Co-Kern $coker(F) := \{x \in X | F(x) = 0\}$
- Höhe $hgt(F) := sup\{F(x) | x \in X\}$
- Tiefe $dpth(F) := inf\{F(x) | x \in X\}$
- Kardinalität ist die Fläche der Fuzzy-Menge und wird entsprechend definiert: $card(F) := \begin{cases} \sum_{x \in X} F(x) & \text{falls X endlich} \\ \int_X F(x)dx & \text{sonst} \end{cases}$

Die bisherige Darstellung in Kurvenform wird auch vertikale Darstellung genannt. Diese ist sehr anschaulich, allerdings für einen Rechner schwer zu handhaben. Dazu gibt es eine horizontale Darstellung in der Form von α-Schnitten, in der das Rechnen oft einfacher ist. [GKI06] Wird eine gerade parallel zur x-Achse durch die Fuzzy-Menge F gelegt, so heißen alle Elemente, die einen Zugehörigkeitsgrad von mindestens α haben der α-Schnitt (engl. c-Cut): $cut_c(F) := \{x \in X | F(x) \geq c\}$. Alle Elemente mit deren Zugehörigkeitsgrad auf der Geraden liegen heißen α-Niveau (auch α-Ebene, engl. c-Level): $level_c(F) := \{x \in X | F(x) = c\}$. [Rud07] Entsprechend läßt sich eine Fuzzy-Menge auch vollständig durch den α-Schnitt von 0 bzw. durch die Vereinigung aller α-Ebenen beschreiben. [GKI06]

Zusätzlich können Fuzzy-Mengen weitere Eigenschaften besitzen.

- F ist normal, falls $hgt(F) = 1$
- F ist co-normal, falls $dpth(F) = 0$
- F ist stark normal, falls $\exists x \in X : F(x) = 1$
- F ist stark co-normal, falls $\exists x \in X : F(x) = 0$

Anhand der Auswahl der Eigenschaften ist ersichtlich, dass die Zugehörigkeitswerte bei Fuzzy-Mengen nicht das gesamte Intervall [0,1] abdecken müssen. [Thi99]

B.2.2 Standard-Mengenoperationen

Seien nachfolgend A,B zwei beliebige Fuzzy-Mengen über einem Universum X.

A und B sind gleich, wenn allen Elementen des Universums X der gleiche Zugehörigkeitswert zugewiesen wird: $A = B$ falls $\forall x \in X : A(x) = B(x)$.

Bei der Teilmengenbeziehung können zwei Arten unterschieden werden, die scharfe sowie die unscharfe Teilmenge zweier Fuzzy-Mengen. Bei der scharfen Teilmengenbeziehung gilt $A \subseteq B$ nur, wenn $\forall x \in X : A(x) = 0 \vee A(x) = B(x)$ gilt. Die unscharfe Teilmengenbeziehung $A \subseteq B$ lockert diese auf zu $\forall x \in X : A(x) \leq B(x)$.
Die Standard-Mengenoperationen Vereinigung, Durchschnitt und Komplement der Fuzzy-Logik gehen auf die allgemeinste Fassung der Lukasiewicz Funktionen für „und“, „oder“ und „nicht“ zurück. [Thi99]

- Vereinigung $A \cup B := max\{A(x), B(x)\}$
- Durchschnitt $A \cap B := min\{A(x), B(x)\}$
- Komplement $A^c := 1 - A(x)$

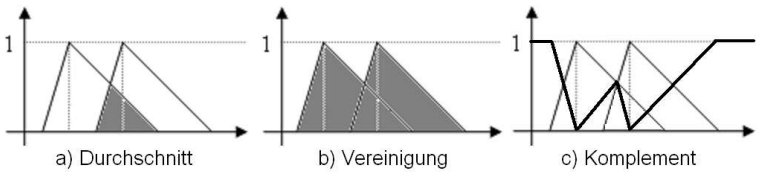


Abbildung 25: Visualisierung der Standard-Mengenoperationen der Fuzzy-Logik basierend auf den allgemeinsten Lukasiewicz Funktionen für „und“, „oder“ und „nicht“.

B.2.3 t-Norm, s-Norm und Komplement

Allgemein können bei Fuzzy-Mengen eigene Operationen für Vereinigung, Durchschnitt und Komplement erstellt werden. Damit diese mit der Fuzzy-Logik kompatibel sind, werden einige Anforderungen an diese in Form von Axiomen definiert.

t-Normen sind Abbildungen $t : [0, 1] \times [0, 1] \rightarrow [0, 1]$ die folgende vier Bedingungen erfüllen und entsprechen damit der Schnittmenge zweier Fuzzy-Mengen. Seien a,b,c Zugehörigkeitswerte.

1. Assoziativität: $t(a, t(b, c)) = t(t(a, b), c)$
2. Kommutativität: $t(a, b) = t(b, a)$
3. Monotonie: Aus $a \leq b$ folgt $t(a, d) \leq t(b, d)$ für $d \in [0, 1]$
4. neutrales Element: $t(a, 1) = a$ für alle $a \in [0, 1]$

Diese Eigenschaften lassen sich aus den Eigenschaften einer Schnittoperation auf zweiwertigen Mengen herleiten. Assoziativität muss gelten, da der Durchschnitt von drei oder mehr Mengen immer derselbe ist, egal in welcher Reihenfolge der Durchschnitt berechnet wird. Ebenso darf es keinen Unterschied machen, ob A mit B oder B mit A geschnitten wird, also muss die Kommutativität gelten. Die Forderung der Monotonie läßt sich gut an der Interpretation des Univserums als Menge von Fakten verdeutlichen. Wenn in A weniger Fakten als in B enthalten sind, und ich vergleiche diese mit den Fakten einer Menge C, so dürfen im Durchschnitt keine neuen Fakten dazukommen. Das neutrale Element wird gefordert, da der Schnitt einer Menge mit dem gesamten Universum genau der Menge selbst entspricht.

Abbildungen $s : [0, 1] \times [0, 1] \rightarrow [0, 1]$ die die Bedingungen 1-3 erfüllen aber deren neutrales Element $s(a, 0) = a$ für alle $a \in [0, 1]$ ist, heißen s-Norm oder t-Conorm. s-Normen entsprechen der Vereinigungsmenge.

Eine Abbildung $c : [0, 1] \rightarrow [0, 1]$ heißt Komplement wenn die drei folgenden Bedingungen erfüllt sind.

1. $c(1) = 0$
2. streng monoton fallend
3. $\forall x \in [0, 1] : c(x) = f^{-1}(f(0) - f(x))$

c heißt involutorische, wenn $c(c(x)) = x$ gilt.

Für jede der drei Operationen können weitere Eigenschaften gefunden werden, die zur feineren Unterscheidung dienen. Im weiteren Umgang mit den Fuzzy-Mengen können diese weiteren Eigenschaften hilfreich sein, da bestimmte Rechenoperationen vereinfacht werden. So sind z.B. die Standardoperatoren für Vereinigung und Durchschnitt die einzigen idempotenten t- bzw. s-Normen.

Soll ein Fuzzy-System mit eigenen Mengenoperationen definiert werden, so wird zu der Kombination der t-Norm, s-Norm und des Komplements zusätzlich gefordert, dass die de Morgan'schen Gesetze erfüllt sind. Seien $a, b \in [0, 1]$.

- $c(t(a, b)) = s(c(a), c(b))$
- $c(s(a, b)) = t(c(a), c(b))$

Eine t-Norm und eine s-Norm heißen dann dual bzgl. einem Fuzzy-Komplement bzw. duales Tripel (t, s, c) [Thi99].

B.2.4 Fuzzy-Relationen

Eine Relation R über $U_1, ..., U_n$ ist im Allgemeinen definiert als eine Teilmenge $R \subseteq U_1 \times ... \times U_n$, die angibt ob ein Element in R enthalten ist. Eine Fuzzy-Relation R über $U_1, ..., U_n$ ist definiert als $R : U_1 \times ... \times U_n \rightarrow [0, 1]$ und ordnet somit jedem Element seinen Zugehörigkeitsgrad zu der Fuzzy-Menge R zu. Damit sind Fuzzy-Relationen Teilmengen des kartesischen Produkts der einzelnen Universa. Entsprechend behalten die bereits definierten t-Normen, s-Normen sowie das Komplement ihre Gültigkeit [Thi99]. So kann z.B. eine Relation auf zwei Fuzzy-Mengen mittels der Standard-Mengenoperationen definiert werden: Seien X,Y zwei beliebige Fuzzy-Mengen. $R(X,Y)(z) := sup\{min\{X(x_1), Y(x_2)\}|(x_1, x_2) \in U^2\}$
Da Relationen über Abbildungen definiert sind, kann für eine Funktion f deren Bild dem Zielbereich der Relation entspricht, eine fuzzyfizierte Funktion $\hat{f}$, Extension genannt, angegeben werden [GKI06]. Seien X,Y zwei beliebige Fuzzy-Mengen.

$\hat{f}(X,Y)(z) := sup\{min\{X(x_1), Y(x_2)\}|(x_1, x_2) \in U^2 \wedge z = f(x_1, x_2)\}$
Die Extension ordnet dabei einem Element $z \in U$ einen Zugehörigkeitswert gemäß dem Minimum der beiden Zugehörigkeitswerte der Fuzzy-Mengen X und Y. Dabei ist insbesondere z definiert als $z = f(x_1, x_2)$, also der Anwendung der zu fuzzyfizierenden Funktion f auf den Elementen $x_1, x_2 \in U$. Da nicht garantiert ist, dass f bijektiv ist, wird z das Supremum aller für z errechneten Werte zugewiesen.

Zur Anwendung von Extensionen betrachten wir die Addition von Fuzzy-Mengen als Beispiel, siehe Abbildung 26. Gegeben seien die Fuzzy-Mengen ungefähr 1 als A und und ungefähr 4 oder ungefähr 6 als B.

$$A = \begin{cases} x & \text{falls } x < 1 \\ \frac{3-x}{2} & \text{sonst} \end{cases}$$

$$B = \begin{cases} x - 3 & \text{falls } x < 4 \\ 5 - x & \text{falls } x \geq 4 \text{ und } x < 5 \\ x - 5 & \text{falls } x \geq 5 \text{ und } x < 6 \\ 7 - x & \text{sonst} \end{cases}$$

Damit ergibt sich für die Addition (f ist also +) z.B. bei z=3 und z=5
$\hat{+}(A,B)(z) := sup\{min\{A(0), B(3)\}|(0,3) \in U^2 \wedge z = +(0,3) = 3\} = 0$ und
$\hat{+}(A,B)(z) := sup\{min\{A(1), B(4)\}|(1,4) \in U^2 \wedge z = +(1,4) = 5\} = 1.$
Da Relationen im Allgemeinen n-dimesional definiert sind, können Extensio-

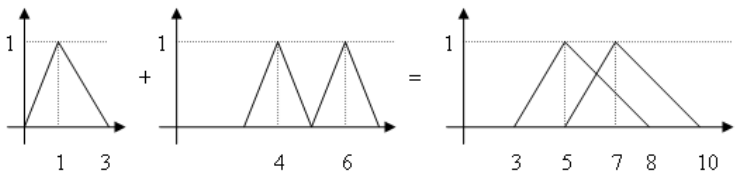


Abbildung 26: Fuzzyfizierte Addition zweier Fuzzy-Mengen.

nen entsprechend verallgemeinert werden. Seien $X_1, ..., X_n$ beliebige Fuzzy-Mengen, t t-Norm sowie f die zu fuzzyfizierende Funktion. Dann ist $\hat{f}(X_1, ..., X_n)(z) := sup\{t\{X_1(x_1), ..., X_n(x_n)\}|(x_1, ..., x_n) \in U^n \wedge z = f(x_1, ..., x_n)\}$ die Extension zu f.

Die hintereinander Ausführung von Fuzzy-Relationen wird als Produkt der Relationen bezeichnet. Voraussetzung ist eine Überlappung der Universa derart, dass das letzte Universum der einen Relation immer mit dem ersten Universum der folgenden Relation übereinstimmt: $R_1 = U_1 \times ... \times U_n \times W$ multipliziert mit $R_2 = W \times V_1 \times ... \times V_n$ ergibt $R_1 \circ R_2 = U_1 \times ... \times U_n \times V_1 \times ... \times V_n$ [Thi99].

B.2.5 Implikation

Vor dem Einstieg in die Inferenz der Fuzzy-Logik und der Regelbasen müssen zwei Begriffe definiert werden, die den Zusammenhang zwischen Fuzzy-Mengen definieren. Dazu betrachten wir nochmal folgenden Satz aus dem Motivationsbeispiel: „Ein Mensch mit einer Körpergröße von 190 cm ist groß." Das umgangssprachliche Konzept „groß" beschreibt dabei eine Ausprägung des Merkmals „Körpergröße". Andere Ausprägungen wären z.B. „klein" oder „durchschnittlich". Diese Ausprägungen heißen linguistische Terme. Das Merkmal, dass durch die linguistischen Terme beschrieben wird, heißt linguistische Variable und faßt alle linguistischen Terme in einer neuen Fuzzy-Menge zusammen. [DD97]

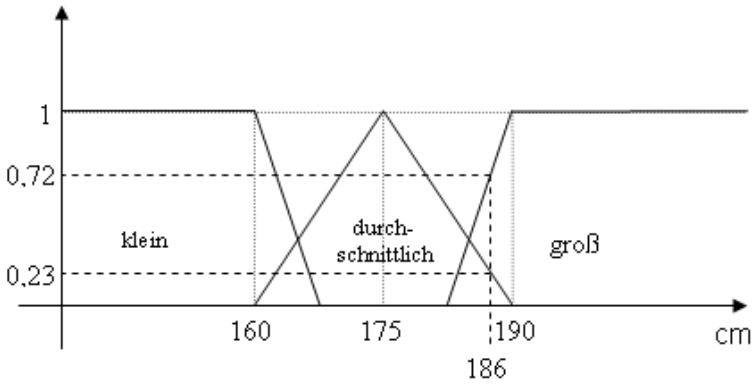


Abbildung 27: Die linguistische Variable „Körpergröße" zu den linguistischen Termen „klein", „durchschnittlich" und „groß".

Das in Abbildung 27 eingezeichnete Beispiel zeigt, dass in einer linguistischen Variable festgelegt wird, wie hoch der Zugehörigkeitswert der Körpergröße von 186 cm zu den einzelnen linguistischen Termen ist. Wenn man diese als Wahrscheinlichkeiten interpretiert ist die Aussage einer linguistischen Variable also, dass ein Mensch der 186 cm Körpergröße hat, mit 23% Wahrscheinlichkeit als durchschnittlich und mit 72% Wahrscheinlichkeit als groß gilt. Das es sich aber nicht tatsächlich um Wahrscheinlichkeiten handelt wird dadurch deutlich, dass die Summe der Zugehörigkeitswerte nicht eins (100%) ergeben müssen.

Ausgehend von den definierten Ausdrücken können Fuzzy-Regeln beschrieben werden, bei denen vom Zustand einer linguistischen Variable auf den Zustand einer anderen linguistischen Variable geschlossen wird: Das Fuzzy-IF-THEN ist definiert als „IF X ist A, THEN Y ist B". Dabei sind X,Y linguistische Variablen und A,B sind linguistische Terme. Zur Berechnung des Zugehörigkeitswertes von B(y) kann eine Relation $R(x, y) = Imp(A(x), B(x))$ definiert werden, die für die unscharfen Mengen A' über X und B' über Y bei Multiplikation den gewünschten Wert liefert: A'$\circ$R=B'. Diese Form der Definition entspricht auch der Schlußfolgerung „Wenn A, dann B" in der Aussagenlogik,

die über die Implikation „A⇒B“ in Form einer Abbildung $\Rightarrow: A \times B \rightarrow \{0,1\}$ mit den bekannten Wahrheitswerten definiert ist.

Analog zur Definition der nicht-Standard-Mengenoperationen werden an die Relation $Imp(\cdot,\cdot)$ Anforderungen in Form von Axiomen gestellt.

1. $a \leq b$ impliziert $Imp(a,x) \geq Imp(b,x)$ Monotonie im ersten Argument
2. $a \leq b$ impliziert $Imp(x,a) \leq Imp(x,b)$ Monotonie im zweiten Argument
3. $Imp(0,a) = 1$ Dominanz der Unrichtigkeit
4. $Imp(1,b) = b$ Neutralität der Korrektheit
5. $Imp(a,a) = 1$ Identität
6. $Imp(a, Imp(b,x)) = Imp(b, Imp(a,x))$
7. $Imp(a,b) = 1$ gdw. $a \leq b$ Randbedingung
8. $Imp(a,b) = Imp(c(b), c(a))$ Kontraposition
9. $Imp(\cdot,\cdot)$ Stetigkeit

Für nähere Erläuterungen verweise ich auf [Thi99] und zeige eine Auswahl möglicher Definitionen von $Imp(\cdot,\cdot)$, die prinzipiell über drei verschiedene Ideen hergeleitet werden können und somit drei Typen von Implikationen definieren.

1. S-Implikationen mit $Imp(a,b) = s(c(a),b)$
   Beispiel: $Imp(a,b) = max\{1-a, b\}$ (Kleene-Dienes)
2. R-Implikationen mit $Imp(a,b) = max\{x \in [0,1] : t(a,x) \leq b\}$
   Beispiel: $Imp(a,b) = \begin{cases} 1 & \text{für a} \leq \text{b} \\ b & \text{sonst} \end{cases}$ (Gödel)
3. QL-Implikationen mit $Imp(a,b) = s(c(a), t(a,b))$
   Beispiel: $Imp(a,b) = max\{1-a, min\{a,b\}\}$ (Zadeh)

B.2.6 Fuzzy-Regelbasen

Eine Fuzzy-Regelbasis ist eine Sammlung von Fuzzy-IF-THEN Regeln, in der von den Zuständen verschiedener linguistischer Variablen auf die möglichen linguistischen Terme einer oder mehrerer anderer linguistischer Variablen gefolgert wird.

IF X ist A_1, THEN Y ist B_1
IF X ist A_2, THEN Y ist B_2
...
IF X ist A_n, THEN Y ist B_n
Dabei können nun mehrere B_i einen Zugehörigkeitswert größer als 0 erhalten, also müssen die Ergebnisse der Regelauswertung zu einer Fuzzy-Menge zusammengefaßt werden. Die intuitive Vorgehensweise entspricht dem FITA (First inference, then aggregate) Prinzip, bei dem zuerst alle Regeln für das gesamte betroffene Universum ausgewertet werden. Erst

wenn die Ergebnisse aller Regeln vorliegen werden diese zu einem Ergebnis zusammengefasst: $B'(y) = \beta(R_1(x,y) \circ A'(x), ..., R_n(x,y) \circ A'(x))$.
Dem gegenüber werden bei dem FATI (First aggregate, then inference) Prinzip zuerst alle Regeln zu einer Superrelation zusammengefaßt, die dann im Anschluss ausgewertet wird: $B'(y) = \alpha(R_1(x,y), ..., R_n(x,y)) \circ A'(x)$.
FITA und FATI liefern für eine beliebige t-Norm gleiche Ergebnisse wenn $\alpha(\cdot) = \beta(\cdot)$ gilt. Beide Prinzipien haben ihre Vorteile. Bei FITA sind z.B. erste Ergebnisse schnell zu haben, dafür muss jedoch mehrfach das gesamte Universum ausgewertet werden, was in Abhängigkeit von dessen Größe eventuell lange dauert. Bei FATI hingegen wird das Universum nur einmal ausgewertet, dafür gibt es ein Ergebnis erst nach der vollständigen Auswertung der Regelbasis.

Durch das auswerten der Regelbasis wird neues Wissen abgeleitet. Dies kann in beide Richtungen der Fuzzy-IF-THEN Regel erfolgen. Dazu stehen in der Fuzzy-Logik generalisierte Formen von Inferenzen zur Verfügung.

- Generalisierten Modus Ponens:
  IF X ist A, THEN Y ist B
  X ist A'
  Y ist B'
- Generalisierter Modus Tollens:
  IF X ist A, THEN Y ist B
  Y ist B'
  X ist A'
- Generalisierter Hypothetischer Syllogismus:
  IF X ist A, THEN Y ist B
  IF Y ist B, THEN Z ist C
  IF X ist A, THEN Z ist C

Betrachten wir ein Beispiel eines „klassichen“ Mamdani-Controllers aus der Regelungstechnik, dem ursprünglichen Anwendungsgebiet der Fuzzy-Logik.

Der Regler soll die Erdgaszufuhr zu einer Heizung in Abhängigkeit von der Temperatur regulieren. Aus der Beschreibung ergeben sich die linguistischen Variablen

- T (Temperatur) mit den Termen „hoch“ (F2) und „niedrig“ (F1) und
- NG (Erdgaszufuhr) mit den Termen „viel“ (G1) und „wenig“ (G2).

Dazu sei folgende Regelbasis gegeben.

- IF T ist niedrig THEN NG ist viel
- IF T ist hoch THEN NG ist wenig

In Abbildung 28 ist für ein konkretes x_0 ein Zugehörigkeitswert zu den linuistischen Termen „hoch“ und „niedrig“ der Temperatur zu sehen. Da nicht die gesamte Fuzzy-Menge

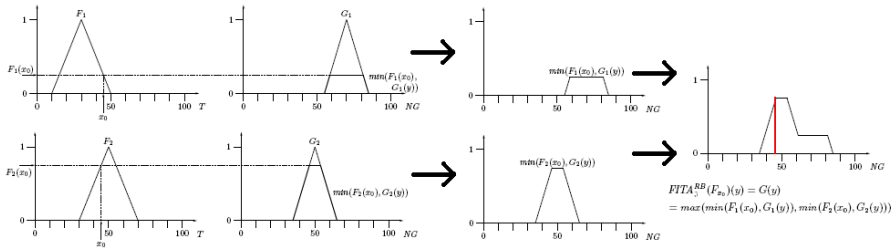


Abbildung 28: Beispiel einer Heizungssteuerung mittels eines Fuzzy-Systems [Thiele].

des jeweiligen linguistischen Terms weiterverwendet wird, sondern nur ein Schnitt, liegt folgende Situation vor: Die Regel „IF X ist A, THEN Y ist B“ ist bekannt sowie das X den Wert A' (die Teilmenge von A) hat. Die Inferenzart ist also der generalisierte Modus Ponens und es kann auf B' geschlossen werden. Hier wird zur Auswertung das FITA Prinzip verwendet, da zunächst beide Regeln ausgewertet werden. Das Ergebnis beider Regeln wird dann zusammengefaßt. Ein sparsamer Heizungsregler kann dann z.B. den höchsten Zugehörigkeitswert mit der niedrigsten Erdgaszufuhr wählen (rote Linie).

B.3 Struktur eines Fuzzy-Systems

Es gibt verschiedene Modelle von Fuzzy-Systemen, die dessen Komponenten sowie eventuell den Ablauf beschreiben (siehe Abbildung 29). Zu beiden Varianten soll nun ein Modell vorgestellt werden.

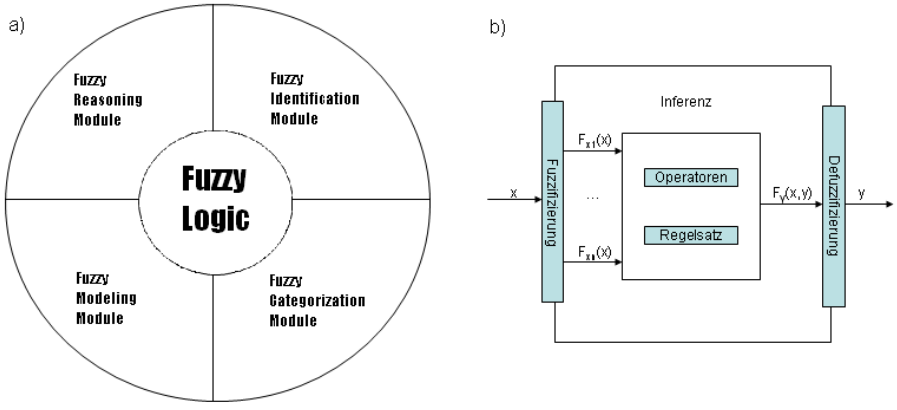


Abbildung 29: Das ICMR structural Framework [Lee2005] (a) und die Struktur eines Mamdani-Fuzzy-Systems [Ana2001] (b).

Bei dem abstrakteren ICMR structural Framework steht die Fuzzy-Logik im Mittelpunkt und die vier wesentlichen Modellierungskomponenten kapseln diese. Wenn ein Fuzzy-System nach dem ICMR structural Framework aufgebaut werden soll, so werden im ersten Schritt in dem Fuzzy Identification Module alle für das System relevanten linguistischen Variablen identifiziert. Im zweiten Schritt werden dann im Fuzzy Categorization Module die für das System relevanten linguistischen Terme festgelegt. Das Fuzzy Modeling Module faßt schließlich die linguistischen Variablen und deren Terme zusammen (für ein Beispiel

siehe Abbildung 27). Als letzter Schritt bei der Erstellung des Fuzzy-Systems wird die Regelbasis definiert, die die Reaktionen des Fuzzy-Systems in Abhängigkeit von der Eingabe festlegt und somit eine Verwertung des bisher modellierten Wissens erlaubt.

Die Struktur des Mamdani-Fuzzy-Systems stellt die Inferenz hingegen direkt in den Vordergrund. Es zeigt wie als Eingabe ein konkreter Wert x vorliegt, dessen Zugehörigkeitsgrad zu verschiedenen linguistischen Termen $F_{xi}(x)$ ermittelt wird. Das Ergebnis dieser Fuzzifizierung wird durch das Regelwerk verarbeitet und es entsteht eine neue Fuzzy-Menge $F_y(x, y)$. Bei der Defuzzifizierung wird sich dann nach einer definierten Auswahlregel für einen konkreten Wert aus der Fuzzy-Menge $F_y(x, y)$ entschieden und dieser als Ergebnis ausgegeben.

B.4 Praxisbeispiel: Kartenentdeckung

Dieser Abschnitt basiert auf dem Artikel [YI05]. Hierbei wurde ein Roboter mit einem Sonarsensor[14] durch eine Büroumgebung geschickt und aus den ermittelten Daten eine vollständige Karte gezeichnet. Die Idee dahinter ist folgende: Es wird nicht nur ein Fuzzy-System definiert, sondern direkt drei Fuzzy-Systeme, die jeweils eine spezielle Aufgabe haben und hierarchisch derart gegliedert sind, dass die Ausgaben von zwei Fuzzy-Systemen dem dritten System als Eingabe dienen. Damit dies auch in Echtzeit funktioniert, wird zunächst auf Teilen (lokale Zellen) der Gesamtkarte (globale Karte) gearbeitet, die dann nur in bestimmten Intervallen zu einer Karte zusammengefaßt werden. Wenn eine Sichtlinie in Nero existiert, dann ist diese das Gegenstück zu den Sonarauswertungen.

Im ersten Schritt werden die empfangenen Sensorinformationen den einzelnen lokalen Zellen der gesamten Karte zugeordnet. In einer lokalen Zelle werden mit *adaptive fuzzy Clustering* [IR04] Messpunkte zusammengefaßt um Fehlinformationen zu Filtern und Linienabschnitte zu erkennen. Die beiden Fuzzy-Systeme auf der unteren Hierarchie-Ebene arbeiten mit dem Eigenvektor bzw. den Eigenwerten der Cluster um die Liniensegmente gut aus den Meßdaten zu approximieren. Nachfolgend wird davon abstrahiert: Es wird davon ausgegangen, dass exakte Liniensegmente ermittelt wurden, da das Zustandekommen der Segmente in den Fuzzy-Systemen keine Rolle spielt. Im nächsten Schritt geht es darum, die gefundenen Linienabschnitte in die Gesamtkarte zu integrieren, in der bereits Linienabschnitte existieren. Bei der Integration sowohl in den lokalen Zellen als auch in der Gesamtkarte, werden Linien zusammengefaßt, so dass jeweils der Effekt auftritt, dass einzelne Abschnitte sich „gerade ziehen“. Siehe dazu Abbildung 30.

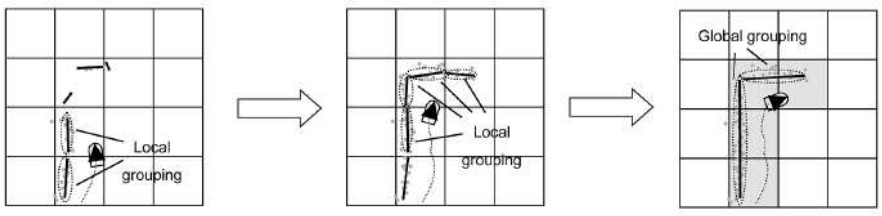


Abbildung 30: Beispiel zur Kartenentwicklung bei Bewegung des Roboters. Einzelbilder aus [IRW2005]

[14] siehe http://de.wikipedia.org/wiki/Sonar

Das erste Fuzzy-System bestimmt dabei, ob es sich bei den beiden betrachteten Linienabschnitten um eine Kreuzung handelt. Dazu werden die Linienabschnitte zu Geraden verlängert und ein Schnittpunkt ermittelt. Wenn die Geraden sich im drei-dimensionalen Raum nicht schneiden, aber der Abstand der beiden Geraden zum Schnittpunkt bei der Projektion in eine Ebene gering ist, dann erkennt das Fuzzy-System dies als ungefähren Schnittpunkt. Je nach Orientierung (z.B. 100° Winkel am Schnittpunkt) der Geraden gibt das erste Fuzzy-System nun als Antwort einen Zugehörigkeitswert zu einem linguistischen Term „gering“, „mittel“ oder „hoch“ der linguistischen Variable „ist Ecke“ zurück.

Das zweite Fuzzy-System verwendet ebenfalls wieder die zu Geraden verlängerten Linienabschnitte. Dieses versucht jedoch herauszufinden, ob die Geraden in etwa parallel verlaufen und ungefähr in der gleichen Ebene liegen. Auch dieses Fuzzy-System hat als Ausgabe den Zugehörigkeitswert zu einem linguistischen Term „gering“, „mittel“ oder „hoch“ der linguistischen Variable „ist Parallel“.

Das dritte Fuzzy-System entscheidet sich nun in Abhängigkeit von der Antwort der ersten beiden Fuzzy-Systeme für eine der beiden Möglichkeiten. Wichtig ist, dass in der Modellierung darauf geachtet wird, dass nicht beide Fuzzy-Systeme als Ausgabe „hoch“ liefern, da sonst keine sinnvolle Entscheidung für eine der beiden Alternativen getroffen werden kann.

Nachdem der Roboter sich entschieden hat, um was es sich bei den beiden Linien handelt, faßt er diese bei Parallelität zu einer Linie zusammen oder er setzt die Endpunkt beider Linien auf den ermittelten Schnittpunkt, so dass sich eine saubere Ecke ergibt.

C Neuronale Netze

C.1 Biologische Grundlagen „Vorbild menschliches Gehirn“

Künstliche neuronale Netze sind dem Nervensystem und speziell dem Gehirn von Tieren und Menschen nachempfunden. Das Nervensystem von Lebewesen besteht aus dem Gehirn (Zentrales Nervensystem), den verschiedenen sensorischen Systemen, die Informationen sammeln, und dem motorischen System, das für die Bewegung zuständig ist. Bei der Informationsverarbeitung sind die Neuronen von zentraler Bedeutung. Ein prototypischer Aufbau biologischer Neuronen und deren Kommunikation wird in Abbildung 31 dargestellt.

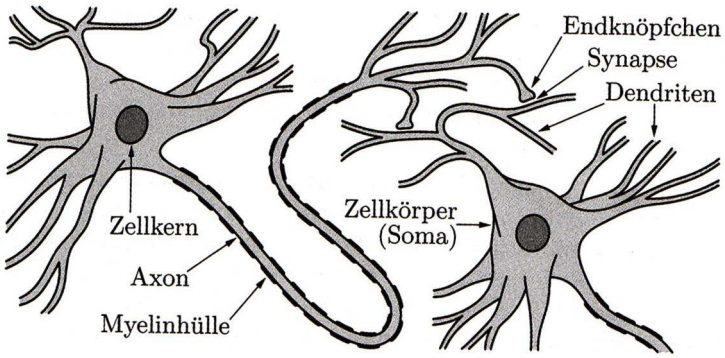


Abbildung 31: Darstellung der Kommunikation zwischen zwei Neuronen (entnommen aus [BKKN03] Seite 6)

Ein Neuron besteht aus einem Zellkörper (Soma), der den Zellkern enthält, dem Axon, der an den Enden so genannte „Endknöpfchen“ besitzt, und den Dendriten, die das Gegenstück zu den „Endknöpfchen“ des Axons darstellen. Das Axon ist von einer Myelinhülle umgeben, was jedoch für die Kommunikation nicht relevant ist. Eine „Beinaheberührung“ eines Axons und eines Dendriten wird als Synapse bezeichnet. Eine Kommunikation zwischen Neuronen läuft wie folgt ab:

1. Signalweitergabe: Das Axon eines Neurons erhält einen Nervenimpuls, wodurch es an den „Endknöpfchen“ zu einer Ausschüttung von Neurotransmittern (bestimmte Chemikalien) kommt.

2. Signalaufnahme: Die ausgeschütteten Neurotransmitter wirken sich auf alle anliegenden Dendriten (verbundene Neuronen) aus und ändern dort die Polarisation (elektrisches Potential). Dabei können die Synapsen exzitatorisch (erregend) oder inhibitorisch (hemmend) wirken.

3. Signalverarbeitung: Die Änderungen des elektrischen Potentials werden am Zellkörper akkumuliert und falls ein gewisser Schwellenwert erreicht ist, wird dieses geänderte elektrische Potential (Aktionspotential) entlang des Axons weitergereicht.

Die Information im Nervensystem des Menschen besteht primär aus zwei sich ständig ändernden Größen:

- dem elektrischen Potential der Neuronenmembran,
- der Anzahl der Nervenimpulse, die ein Neuron pro Sekunde weiterleitet (Feuerrate).

Da das menschliche Gehirn aus circa 100 Milliarden Neuronen besteht, ist die Informationsverarbeitung beim Menschen ein sehr komplexer Prozess. In diesem Seminar werden die Grundlagen der künstlichen neuronalen Netze vorgestellt.

C.2 Das künstliche Neuron

Ein künstliches Neuron kann als Tupel $(\vec{x}, \vec{w}, \mathrm{f}_{net}, net, \mathrm{f}_a, \mathrm{f}_o, o)$ definiert werden, bestehend aus einem Eingabevektor $\vec{x}=(x_1, ..., x_n)$ mit $x_i \in \Re$, einem Gewichtsvektor $\vec{w}=(w_1, ..., w_n)$ mit $w_i \in \Re$, einer Funkion f_{net}, die die Netzeingabe $net \in \Re$ berechnet, einer Aktivierungsfunktion f_a und einer Ausgabefunktion f_o, die die Ausgabe $o \in \Re$ berechnet. Die Abbildung 2 zeigt ein künstliches Neuron mit seinen Elementen. Im Folgenden werden die Charakteristika des künstlichen Neurons genauer beschrieben.

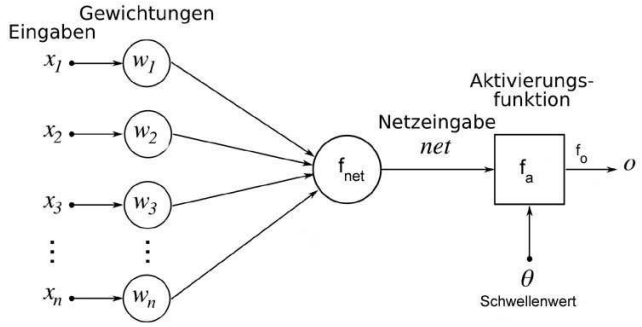


Abbildung 32: Grafische Darstellung eines künstlichen Neurons mit Eingaben, Gewichtungen, Netzberechnungsfunktion, Netzeingabe, Aktivierungsfunktion, Schwellenwert, Ausgabefunktion und Ausgabe.

- Eingabe, Gewichtungen: Die Gewichtungen w_i bestimmen den Einfluss der jeweiligen Eingabe bei der späteren Berechnung der Aktivierungsfunktion. Je nach Vorzeichen der Gewichtung wirkt die Eingabe inhibitorisch oder exzitatorisch. Der Wert 0 wird benutzt, um nicht existente Verbindungen zwischen zwei Neuronen darzustellen.
- *net*, f_{net}: Die Netzeingabe *net* wird der Aktivierungsfunktion übergeben. Sie wird mit Hilfe der Funktion f_{net} berechnet. Dabei unterscheiden wir zwei Arten der Berechnung (siehe [AP02] Seite 18):
  1. „summation units“ (SU): Hier wird die Netzeingabe als gewichtete Summe über alle Eingabesignale berechnet:
     $$f_{net}(\vec{x}, \vec{w}) = \sum_{i=1}^{N} x_i w_i = net$$
  2. „product units“ (PU): Hier werden zunächst die Eingaben potenziert mit den Gewichtungen. Die Netzeingabe ist dann das Produkt der potenzierten Eingaben:

$$f_{net}(\vec{x}, \vec{w}) = \prod_{i=1}^{N} x_i^{w_i} = net$$

- f_a: Durch die Aktivierungsfunktion $f_a : \Re \rightarrow \Re$ des Neurons wird entschieden, ob das Neuron „feuert“ (aktiviert ist). Die Aktivierung wird beeinflusst durch die Netzeingabe net und einen Schwellenwert (threshold) θ. Auch hier gibt es verschiedene Arten der Berechnung. Eine einfache Funktion ist beispielsweise die Schwellenwertfunktion:

$$f_a = \begin{cases} 1, & \text{wenn } net \geq \theta \\ 0, & \text{sonst} \end{cases}$$

  Wenn also die gewichteten Eingaben des Neurons einen Wert (threshold) überschreiten, dann feuert das Neuron. Einige weitere Aktivierungsfunktionen sind im Buch von Andries P. Engelbrecht „Computational Intelligence An Introduction“ ([AP02]) auf Seite 19 dargestellt.

- o, f_o: Die Ausgabefunktion $f_o : \Re \rightarrow \Re$ berechnet bezüglich der Aktivierungsfunktion den Ausgabewert des Neurons. Meist wird dies als binäre Zuweisung realisiert: $f_o : \Re \rightarrow \{0, 1\}$. D.h. wenn das Neuron feuert, wird eine 1 ausgegeben, ansonsten eine 0.

In Abbildung 33 werden die Funktionen Konjunktion, Disjunktion und Negation in Form von künstlichen Neuronen dargestellt. Bei den drei Neuronen werden summation units für die Berechnung der Netzeingabe benutzt und die Aktivierungsfunktionen sind Schwellenwertfunktionen. Bei der Negation wird das Gewicht -1 gewählt und der Schwellenwert -0,5. Bei einer binären Eingabe ergibt sich am Ausgang eine 1, wenn am Eingang eine -1 anliegt. Wenn aber am Eingang eine 1 anliegt, dann ist die Netzeingabe -1, was kleiner als der Schwellenwert ist, und somit liegt am Ausgang eine 0 an. Bei der Konjunktion sind beide Gewichte 1 und der Schwellenwert beträgt 1,5. Bei einer binären Eingabe ergibt sich am Ausgang nur eine 1, wenn an beiden Eingängen eine 1 anliegt. Die Disjunktion funktioniert analog zur Konjunktion mit dem Schwellenwert 0,5.

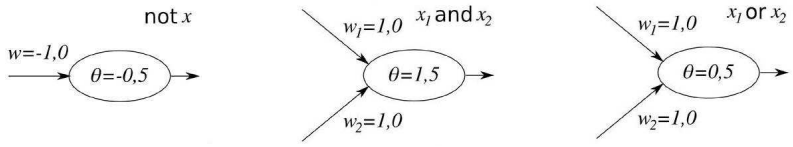


Abbildung 33: Negation, Konjunktion und Disjunktion dargestellt durch künstliche Neuronen.
f_{net} : “summation units“, f_a : Schwellenwertfunktionen. (entnommen [wiki:5])

C.3 Neuronale Netze

Definition: Ein künstliches neuronales Netz ist ein (gerichteter) Graph G=(N,E), dessen Knoten n_i, $i \in \{1, ..., |N|\}$ Neuronen und dessen Kanten e_i, $i \in \{1, ..., |E|\}$ Verbindungen heißen. Die Menge N der Knoten ist unterteilt in die Menge N_{in} der Eingabeneuronen (input neurons), die Menge N_{out} der Ausgabeneuronen (output neurons) und die Menge N_{hidden} der versteckten Neuronen (hidden neurons)[NBKK03]. Es gilt:

$$N = N_{in} \cup N_{out} \cup N_{hidden}$$
$$N_{in} \neq \emptyset, N_{out} \neq \emptyset$$
$$N_{hidden} \cap (N_{in} \cup N_{out}) = \emptyset$$

Jeder Verbindung zwischen zwei Neuronen ist ein Gewicht w zugeordnet und jedes Neuron n_i besitzt die Zustandsgrößen Netzeingabe net_{n_i}, Aktivierung act_{n_i} und Ausgabe out_{n_i}. Eingabeneuronen besitzen noch zusätzlich die externe Eingabe ext_{n_i} als Zustandsgröße.

Die Abbildung 34 zeigt ein allgemeines Schema eines neuronalen Netzes.

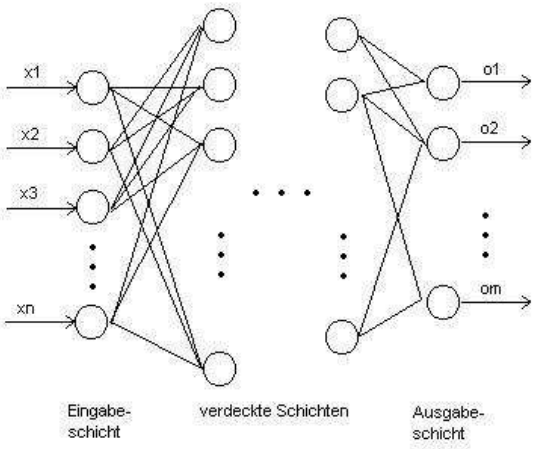


Abbildung 34: Allgemeines Neuronales Netz (entnommen aus [Lip07])

Anhand der Grafik kann man erkennen, dass nur die Eingabe- und Ausgabeneuronen mit der Umgebung verbunden sind, während die „hidden neurons“ nur interne Verbindungen besitzen. Auch müssen die Mengen N_{in} und N_{out} nicht disjunkt sein. D.h. ein Neuron kann gleichzeitig Eingabe- als auch Ausgabeneuron sein.

Es gibt folgende zwei Netztypen:

- vorwärtsbetriebene Netze (Feedforward-Netze), falls der Graph azyklisch ist.
- rückgekoppelte Netze (rekurrente Netze), falls der Graph Schleifen oder gerichtete Kreise besitzt.

Beide Netzwerktypen werden in den folgenden Unterkapiteln genauer erläutern und spezielle Arten dieser Typen vorgestellt. Im Folgenden soll die Arbeitsweise neuronaler Netze

vorgestellt werden. In Kapitel C.2 wurde gezeigt, wie ein einzelnes Neuron Eingaben verarbeitet und Ausgaben produziert. Die Arbeitsweise eines neuronalen Netzes kann man in zwei Phasen unterteilen:

1. Eingabephase: Diese Phase dient der Initialisierung des neuronalen Netzes. Es werden die externen Eingaben eingelesen und den Aktivierungen der Eingabeneuronen zugewiesen. Die Aktivierungen der übrigen Neuronen werden willkürlich gesetzt, gewöhnlich 0. Zuletzt werden noch die Ausgabefunktionen, je nach Aktivierung des Neurons, berechnet, so dass alle Neuronen Ausgaben produzieren ([NBKK03]).

2. Arbeitsphase: In dieser Phase werden die externen Eingaben deaktiviert (nicht beachtet) und alle Aktivierungen und Ausgaben mehrfach neu berechnet. Dieser Schritt ist notwendig, da es rekurrente Kanten geben kann, die sich noch auf das Netz auswirken können. Die Phase wird beendet, wenn das Netz einen stabilen Zustand erreicht, wenn sich also die Ausgaben der Neuronen nach Neuberechnungen nicht mehr ändern oder wenn eine bestimmte Anzahl von Neuberechnungen durchgeführt wurde ([NBKK03]).

Hier sei noch darauf hingewiesen, dass sich bei rekurrenten Netzen je nach Reihenfolge der Berechnungen unterschiedliche Ergebnisse einstellen können, da die zeitliche Abfolge der Neuberechnungen im Allgemeinen nicht festgelegt ist. Bei Feedforward-Netzen wird meist eine Berechnung gemäß topologischer Ordnung (siehe [Bri05]) durchgeführt, die sicherstellt, dass alle Eingaben eines Neurons bereits verfügbar sind (schon berechnet wurden), bevor es seine Ausgabe berechnet. Dadurch werden keine unnötigen Berechnungen durchgeführt.

C.3.1 Feedforward-Netze

Wie in Kapitel C.3 beschrieben, besitzen Feedforward-Netze keine Kreise und der Graph ist somit azyklisch. Des Weiteren unterscheiden wir zwischen zwei Arten von Netzen:

- Ebenenweise verbundene Netze: Diese Netze besitzen nur Verbindungen zwischen benachbarten Schichten. Falls jedes Neuron der Schicht N_i, $i \in \{1, ..., |N-1|\}$ mit jedem Neuron der darauf folgenden Schicht N_{i+1} verbunden ist, spricht man von einem vollständig verbundenen Netz ([Lip07]).

- Allgemeine feedforward-Netzen: Hier sind so genannte „shortcut connections“ erlaubt, also Verbindungen zwischen Neuronen, die Ebenen überspringen ([Lip07]).

Mehrschichtige Perzeptren Eine der bekanntesten Formen eines Feedforward-Netzes ist das so genannte „multilayer perceptron“ (MLP). *Definition (siehe [NBKK03]):* Ein r-schichtiges Perzeptron ist ein neuronales Netz mit einem Graphen G=(N,E), der den folgenden Einschränkungen genügt:

1. Ein Neuron darf nicht Eingabeneuron und Ausgabeneuron sein: $N_{in} \cap N_{out} = \emptyset$

2. Es gibt genau r-2 versteckte Neuronen-Schichten und ein „hidden-Neuron“ ist genau einer versteckten Schicht zugeordnet:

$$N_{hidden} = N_{hidden}^{(1)} \dot\cup \ldots \dot\cup N_{hidden}^{(r-2)}, \forall 1 \leq i < j \leq r-2 : N_{hidden}^{(i)} \cap N_{hidden}^{(j)} = \emptyset,$$
$$r, i, j \in \mathbb{N}$$

3. Es liegt ein ebenenweise verbundenes Netz vor, also gilt für die Kanten E:

$$E \subseteq (N_{in} \times N_{hidden}^{(1)}) \cup (\bigcup_{i=1}^{r-3}(N_{hidden}^{(i)} \times N_{hidden}^{(i+1)})) \cup (N_{hidden}^{(r-2)} \times N_{out})$$

Falls r=2, also $N_{hidden} = \emptyset$: $E \subseteq N_{in} \times N_{out}$

4. Die Netzeingabefunktion jedes versteckten und jedes Ausgabeneurons ist die gewichtete Summe der Eingänge:

$$\forall n \in N_{hidden} \cup N_{out} : f_{net}^{(n)}(\vec{w_n}, \vec{in_n}) = \sum_{v \in pred(n)} w_{vn} out_v$$

Hierbei bezeichnet $pred(n)$ alle verbundenen Neuronen der Vorgängerschicht, w_{vn} ist das Gewicht der jeweiligen Kante und out_v ist die Ausgabe des jeweilig verbundenen Neurons.

5. Die Aktivierungsfunktion jedes versteckten Neurons ist eine sigmoide Funktion. (siehe [AP02] Seite 19-20)

6. Die Aktivierungsfunktion jedes Ausgabeneurons ist entweder eine sigmoide Funktion oder eine lineare Funktion.

Ein mehrschichtiges Perzeptron besteht also aus einer Eingabe- und einer Ausgabeschicht und aus keiner, einer oder mehreren versteckten Schichten. Das Netz ist ebenenweise verbunden. Als Abschluss der MLPs wird in Abbildung 35 die Biimplikation als mehrschichtiges Perzeptron dargestellt. Die Aktivierungsfunktionen der Neuronen sind Schwellen-

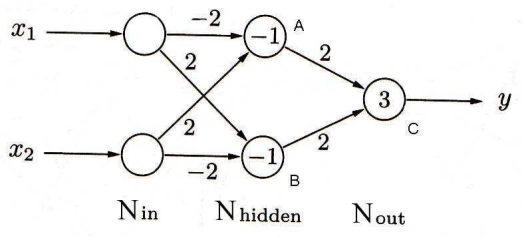


Abbildung 35: Biimplikation (entnommen aus [BKKN03])

wertfunktionen, wie in Kapitel C.2 vorgestellt. Nun lässt sich das MLP folgendermaßen erklären:

- Das Neuron A berechnet die Funktion $x_1 \Rightarrow x_2$, da falls $x_1 = 0$ ist, das Neuron immer „feuert“ und falls $x_1 = 1$ ist, muss zusätzlich $x_2 = 1$ sein, damit der Schwellenwert überschritten wird.
- Das Neuron B berechnet die Funktion $x_2 \Rightarrow x_1$. Die Erklärung erfolgt hier analog zu Neuron A.
- Das Neuron C berechnet nun die Funktion $x_1 \Leftrightarrow x_2$, da der Schwellenwert 3 nur überschritten wird, falls Neuron A und Neuron B „feuern“ und somit beide Richtungen der Implikation korrekt sind.

C.3.2 Selbstorganisierende Karten

Diese sogenannten SOMs (self-organizing maps) sind eine spezielle Form der Radiale-Basisfunktionen-Netze (siehe im Buch [NBKK03] Seite 76-80), bei denen die versteckte Schicht bereits die Ausgabeschicht ist. Des Weiteren sind die Ausgabeneuronen in der Ausgabeschicht in einem Gitter strukturiert, wodurch die Nachbarschaftsbeziehungen beim Training ausgenutzt werden können (siehe dazu Kapitel C.4.2, wo ein SOM-spezifischer Ablauf des unüberwachten Lernens beschrieben wird). Dieses zweischichtige neuronale Netz wird auch Kohonenkarte genannt, benannt nach seinem Erfinder Teuvo Kohonen (siehe [Wik07d]). Für weiterführende Informationen sei auf das Buch „Self-Organizing Maps“ von Kohonen ([Koh01]) verwiesen. Abschließend folgt noch die formale Definition selbstorganisierender Karten, sowie zwei Beispiele für die Anordnung der Ausgabeneuronen in einem Gitter.

Definition ([NBKK03]): Eine selbstorganisierende Karte oder Kohonenkarte ist ein neuronales Netz mit einem Graphen G=(N,E), der folgende Eigenschaften hat:

1. $N_{hidden} = \emptyset$, $N_{in} \cap N_{out} = \emptyset$
2. $E = N_{in} \times N_{out}$
3. Die Netzeingabefunktion und die Aktivierungsfunktion jedes Ausgabeneurons werden definiert wie bei den Radiale-Basisfunktionen-Netzen (siehe [NBKK03] Seite 76).
4. Die Ausgabefunktion jedes Ausgabeneurons ist die Identität. Außerdem gilt das „winner takes all“-Prinzip: Das Neuron mit der höchsten Aktivierung erhält die Ausgabe 1, alle anderen die Ausgabe 0.
5. Auf den Neuronen der Ausgabeschicht wird eine Nachbarschaftsbeziehung definiert, die durch eine Abstandsfunktion beschrieben wird und jedem Paar von Ausgabeneuronen eine nicht negative reelle Zahl zuordnet: $d_{neurons} : N_{out} \times N_{out} \rightarrow \Re_0^+$

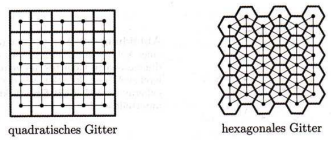


Abbildung 36: SOM: Anordnung der Ausgabeneuronen

C.3.3 Rekurrente Netze

Rekurrente Netze sind Netze, die auch rückwirkende Kanten erlauben, und somit Graphen ermöglichen, die Kreise enthalten. Man kann die sogenannten rückgekoppelten Netze in folgende vier Kategorien unterteilen:

1. Netze mit direkten Rückkopplungen: Die Neuronen haben eine direkte Verbindung von ihrer Ausgabe zu ihrer Eingabe und können dadurch ihre eigene Aktivierung

verstärken oder hemmen. Dies führt meist dazu, dass die Neuronen die Grenzzustände ihrer Aktivierungen annehmen ([Lip07]).

2. Netze mit indirekten Rückkopplungen: Diese Netze besitzen Rückkopplungen von Neuronen in
höheren Schichten zu Neuronen in niedrigeren Schichten. Durch diese Architektur kann man bestimmte Gebiete verstärken und somit bestimmte Eingabeneuronen bevorzugen ([Lip07]).

3. Netze mit Rückkopplungen innerhalb einer Schicht: Diese Netze besitzen Rückkopplungen innerhalb der selben Schicht. Meist werden solche Netze benutzt, um ein Neuron in einer Schicht zu bevorzugen. Dies erreicht man dadurch, dass jedes Neuron zu sich selbst eine verstärkende Rückkopplung hat und zu allen anderen Neuronen in der selben Schicht eine hemmende Rückkopplung. Diese Netze sind auch unter dem Namen „winner-takes-all"-Netzwerk bekannt ([Lip07]).

4. Vollständig verbundene Netze: Diese Netze besitzen Verbindungen zwischen allen Neuronen. Diese sogenannten Hopfield-Netze werden in Unterkapitel C.3.4 detailliert vorgestellt.

Rekurrente Netze können auch erfolgreich eingesetzt werden, um zeitliche Abhängigkeiten in der Datenmenge darzustellen. Des Weiteren sei darauf hingewiesen, dass es sehr viele erfolgreiche Lernverfahren für rekurrente Netze gibt, wobei eines der wichtigsten Lernverfahren das „Backpropagation" ist, das in Kapitel C.4.1 vorgestellt wird.

C.3.4 Hopfield-Netze

Definition [NBKK03]: Ein Hopfield-Netz ist ein neuronales Netz mit einem Graphen G=(N,E) mit folgenden Bedingungen:

1. $N_{hidden} = \emptyset$, $N_{in} = N_{out} = N$
2. $E = N \times N \backslash \{(n,n) | n \in N\}$
3. Die Verbindungsgewichte sind symmetrisch.
4. Die Netzeingabefunktion jedes Neurons n ist die gewichtete Summe der Ausgaben aller anderen Neuronen:

$$\forall n \in N : f_{net}^{(n)}(\vec{w_n}, \vec{in_n}) = \sum_{v \in N \backslash \{n\}} w_{vn} out_v$$

5. Die Aktivierungsfunktion jedes Neurons n ist eine Schwellenwertfunktion:

$$\forall n \in N : f_{act}^{n}(net_n, \theta_n) = \begin{cases} 1, & \text{falls } net_n \geq \theta_n \\ -1, & \text{sonst} \end{cases}$$

6. Die Ausgabefunktion jedes Neurons ist die Identität.

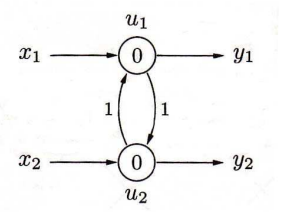


Abbildung 37: Hopfield-Netz, was je nach Eingabe zwischen zwei Zuständen schwankt oder in einen stabilen Zustand konvergiert.

Nach der Definition erhält kein Neuron seine eigene Ausgabe als Eingabe, was Schleifen verhindert. In Abbildung 9 soll die Arbeitsweise eines Hopfield-Netzes dargestellt werden. Das neuronale Netz kann bei paralleler Aktualisierung eine Oszillation darstellen und erreicht bei asynchroner Aktualisierung einen stabilen Zustand. Dies kann man sich gut verdeutlichen, wenn man das Netz mit $x_1 = -1$ und $x_2 = 1$ initialisiert, was zu den Ausgabefunktionen $y_1 = -1$ und $y_2 = 1$ führt. Wenn man nun die externen Eingaben deaktiviert und die Ausgaben neu berechnet, erhält man bei synchroner Aktualisierung eine Oszillation der Ausgaben y_1 und y_2, während das Netz bei asynchroner Aktualisierung in den stabilen Zustand (-1,-1) bzw. (1,1) konvergiert, je nachdem welches Neuron man als erstes aktualisiert.

C.4 Modellierung des Lernens

Die Lernfähigkeit ist die wichtigste Eigenschaft neuronaler Netze. Ein neuronales Netz „lernt“, indem es gemäß einer fest vorgebenen Vorschrift, bestimmte Parameter modifiziert. Prinzipiell kann der Lernprozess aus folgenden Schritten bestehen ([Lip07]):

- Entwicklung neuer Verbindungen
- Löschen existierender Verbindungen
- Modifikation der Gewichte (diese Möglichkeit wird am häufigsten in Lernverfahren umgesetzt)
- Modifikation des Schwellenwertes
- Modifikation der Aktivierungs- bzw. Ausgabefunktion
- Entwicklung neuer Zellen
- Löschen bestehender Zellen

Neuronale Netze lassen sich in vielen Bereichen verändern und können somit einen Lernvorgang nachahmen. Ferner unterscheiden wir zwischen dem Lernen, das extern gesteuert ist (Überwachtes Lernen, siehe Kapitel C.4.1) und dem Lernen, das nicht extern gesteuert ist, sondern sich selbst organisiert (Unüberwachtes Lernen, siehe Kapitel C.4.2). Eine dritte Form des Lernens ist das „Bestärkende Lernen“ (siehe Kapitel C.4.3). Bevor wir uns in den Unterkapiteln den verschiedenen Arten von Lernverfahren widmen, wollen wir uns zwei bekannte Lernregeln anschauen.

Hebb'sche Lernregel: Die Hebb'sche Lernregel ist die Grundlage für viele weitere Lernregeln. Wenn zwei verbundene Neuronen stark aktiviert sind, dann wird das Gewicht der Verbindung erhöht. Formal ([Lip07]):

$$\Delta w_{ij} = \eta \cdot o_i \cdot a_j \text{ und } w_{ij}(t+1) = w_{ij} + \Delta w_{ij}$$

η: Lernrate(Konstante)
Δw_{ij}: Änderung des Gewichtes w_{ij}
o_i: Ausgabe des Vorgängerneurons i
a_j: Aktivität des Nachfolgerneurons j

Das Gewicht zum Zeitpunkt $(t+1)$ ist so definiert, dass zum aktuellen Gewicht die Veränderung Δw_{ij} addiert wird. Es ist zu erkennen, dass das Gewicht mit einer Lernrate von η schneller ansteigt, je stärker die Ausgabe und Aktivität benachbarter Neuronen ist. Ein Problem bei dieser Lernregel tritt auf, falls die Ausgabe und die Aktivität benachbarter Neuronen konstant hoch bleibt. In diesem Fall steigt das Gewicht der Kante ins Unendliche. Wie oben aufgeführt, ist die Hebb'sche Lernregel die Basis vieler Lernregeln, die in den vergangenen Jahren entwickelt wurden.

Delta Regel: Bei dieser Lernregel werden die Gewichte proportional zur Differenz der erzeugten Ausgabe und der gewünschten Ausgabe verändert.

$$\Delta w_{ij} = \eta \cdot o_i \cdot \delta_j$$
$$\delta_j = (v_j - o_j) \text{ und } w_{ij}(t+1) = w_{ij} + \Delta w_{ij}$$

Das Gewicht zum Zeitpunkt $(t+1)$ ist wieder so definiert, dass zum aktuellen Gewicht die Veränderung Δw_{ij} addiert wird. Ferner bezeichnet δ_j den Fehler des Ausgabeneurons j, wobei v_j die gewünschte Ausgabe und o_j die propagierte Ausgabe ist. η ist die Lernrate und o_i bezeichnet die Ausgabe des Neurons i. Ein größerer Fehler sorgt also für eine stärkere Veränderung des Gewichtes der Kante zwischen i und j.

Abschließend unterteilen wir Lernverfahren in zwei Typen:

- batch-Lernverfahren: Dieses sogenannte Offlineverfahren aktualisiert die Gewichte erst nachdem $n \in \mathbb{N}$ Trainingsmuster durchlaufen wurden. Die Anzahl der Trainingsmuster wird auch „batch-size“ genannt([Lip07]).
- online-Lernverfahren: Hier werden die Gewichte nach jedem Trainingsmuster aktualisiert. Meist wird das nächste Trainingspattern randomisiert gewählt, um sich mögliche Reihenfolgen, die in der Trainingsmenge bestehen, nicht mit zu merken. Als Wahrscheinlichkeitsverteilung wird die Gleichverteilung gewählt.

C.4.1 Überwachtes Lernen (Supervised Learning)

Beim „überwachten Lernen“ wird dem Netz sowohl eine Reihe von Eingaben, sowie die zugehörigen korrekten Eingaben präsentiert. Die Aufgabe des Lernalgorithmus ist es nun die Gewichte und Schwellenwerte so anzupassen, dass nach erneuter Präsentation der Muster das neuronale Netz die passende Ausgabe zur Eingabe erzeugt. Ferner ist es wünschenswert, dass das Netz neue Muster korrekt einordnen kann. Die Generalisierungsfähigkeit

ist eine der wichtigsten Eigenschaften neuronaler Netze, da meist nicht alle Eingabemuster von Anfang an bekannt sind ([Lam01]). Das Konzept des überwachten Lernens spricht jedoch gegen das biologische Prinzip, wo es keine Institution gibt, die Eingaben mit gewünschten korrekten Ausgaben präsentiert. Ein überwachter Lernprozess hat folgenden Ablauf ([Lam01]):

1. Aktivierung der Neuronen der Eingabeschicht, entsprechend der Eingabemuster.
2. Die Eingabe wird vom neuronalen Netz verarbeitet und eine Ausgabe erzeugt.
3. Vergleich der Ausgabe mit der gewünschten Ausgabe.
4. Rückwärtspropagation des Fehlers zur versteckten Schicht.
5. Verkleinern des Fehlers durch Modifikation der Gewichte.

Im Unterkapitel C.4.1 wird das Backpropagation stellvertretend für „Supervised Learning“ vorgestellt.

Backpropagation – Gradientenabstiegsverfahren Das Lernverfahren Backpropagation ist eine Verallgemeinerung der Delta-Regel für mehrstufige Netze ([Lam01]). Es kann in drei Phasen unterteilt werden ([Lip07]):

1. Forward Pass: Dem Netz wird ein beliebiger Eingabevektor $\vec{x}$ aus der Trainingsmenge präsentiert. Das Netz berechnet Schicht für Schicht die jeweiligen internen Ausgaben und die internen Eingaben, bis die Ausgabeschicht erreicht ist, in der der Ausgabevektor $\vec{o}$ zurückgegeben wird.
2. Bestimmung des Fehlers: Zu jeder Eingabe $\vec{x}$ aus der Trainingsmenge ist die jeweilige gewünschte Ausgabe bekannt (Supervised Learning). Nun wird mit Hilfe einer Fehlerfunktion der Fehler des Netzes bestimmt. Falls dieser Fehler oberhalb eines „threshold“ (Güteschwelle) liegt, wird das Netz durch den Backward Pass modifiziert, ansonsten wird das Training beendet und eventuell eine Testphase eingeleitet, die die Generalisierungsfähigkeit überprüft. Die Fehlerfunktion kann nun wie folgt aussehen:

   $$\text{mittlerer quadratischer Fehler (MSE): } F(\vec{w}) = \lim_{N\to\infty} \frac{1}{N} \sum_{k=1}^{N} Q(\vec{x}_k, \vec{w})$$

   Der mittlere quadratische Fehler ist der Grenzwert von einer möglichst großen Anzahl von Eingabevektoren $\vec{x}_k \in N$, gemittelt über die Anzahl der betrachteten Trainingsmuster. $F(\vec{w})$ wird als Fehler des Netzes bezüglich des Gewichtsvektors $\vec{w}$ bezeichnet. Die Funktion Q kann wie folgt definiert werden:

   $$\text{quadratischer Fehler: } Q(\vec{x}, \vec{w}) = \left| f(\vec{x}) - o_{net}(\vec{x}, \vec{w}) \right|^2$$

   Dies ist der quadratische Fehler, der bei genau einem speziellen Eingabe- und Gewichtsvektor entsteht. $f(\vec{x})$ ist hier die tatsächlich gewünschte Ausgabe des Netzes bei Eingabevektor $\vec{x}$.
   $o_{net}(\vec{x}, \vec{w})$ ist die tatsächliche Ausgabe des neuronalen Netzes. Es sei noch angemerkt, dass man auch andere Fehlerfunktionen verwenden kann, jedoch ist der MSE ein häufig verwendetes Fehlermaß.

3. Backward Pass: In diesem Schritt werden rückwärts-gerichtet die Verbindungen zwischen den Neuronen modifiziert. D.h. als erstes werden die Verbindungen zwischen der letzten versteckten Schicht und der Ausgabeschicht aktualisiert und als letztes werden die Verbindungen zwischen der Eingabeschicht und der ersten versteckten Schicht modifiziert. Dabei erfolgt die Modifikation mit Hilfe einer Lernregel:

$$\nabla_{\vec{w}} F(\vec{w}) = \lim_{N\to\infty} \frac{1}{N} \sum_{k=1}^{N} \nabla_{\vec{w}} Q(\vec{x}_k, \vec{w})$$
$$\vec{w}\,(t+1) = \vec{w} - \eta \cdot \nabla_{\vec{w}} F(\vec{w})$$

   Backpropagation ist ein Gradientenabstiegsverfahren (siehe [AP02]). Es wird versucht durch Verschieben des Gewichtsvektors $\vec{w}$ in Richtung $-\nabla_{\vec{w}} F(\vec{w})$ das Minimum der Fehlerfunktion zu bestimmen. Dabei bezeichnet $\nabla_{\vec{w}}$ den Gradienten (siehe [AP02]) der Fehlerfunktion bezüglich des Gewichtsvektors $\vec{w}$ und Q ist der quadratische Fehler bezüglich des Eingabevektors x_k und dem Gewichtsvektor $\vec{w}$. Nun wird versucht, ähnlich wie bei der MSE, gemittelt über die Anzahl der Eingabemuster eine Minimierung der Fehlerfunktion bezüglich möglichst vieler Eingabevektoren $\vec{x}_k$ zu erreichen. Der Gewichtsvektor $\vec{w}$ zum Zeitpunkt (t+1) ist so definiert, dass man vom aktuellen Gewicht den Gradienten $\nabla_{\vec{w}} F(\vec{w})$ multipliziert mit einem konstanten Faktor η (Schrittweite) subtrahiert. Danach erfolgt wieder Schritt 2 (Fehlerbestimmung), bis ein lokales Minimum erreicht ist.

Backpropagation ist ein sehr erfolgreiches Lernverfahren. Für weiterführende Informationen sei auf [Lip07] verwiesen.

C.4.2 Unüberwachtes Lernen (Unsupervised Learning)

Beim „unüberwachten Lernen“ erhält das Netz lediglich eine Trainingsmenge von Eingaben und muss diese dann ohne Vorgaben in Cluster (Gruppen) einteilen. Die bekanntesten Netze, die nach diesem Prinzip arbeiten sind die „self-organizing maps“ (siehe Abschnitt C.3.2). Bei SOM beeinflussen aktive Neuronen ihre Nachbarneuronen und es kommt dadurch zu einer Clusterbildung. Beim Lernverfahren wird nach dem stärksten aktivierten Neuron gesucht, und die Gewichte der anderen Neuronen entsprechend der Entfernung zum „Gewinnerneuron“ angepasst. Ein SOM-spezifischer Ablauf eines „unüberwachten Lernens“ ist beispielsweise [Lam01]:

1. Aktiviere Eingabeneuronen, aufgrund des Eingabemusters.
2. Berechne das Netz (Ausgaben).
3. Suche nach dem stärksten aktivierten Neuron (winner-takes-all-Prinzip).
4. Ändere die Gewichte der Neuronen anhand der Entfernung vom zu aktualisiernden Neuron zum Gewinnerneuron.

Für weiterführende Informationen bezüglich Lernverfahren in SOM verweise ich auf das Buch [AP02] (Seite 55-72).

C.4.3 Reinforcement Learning

Als letztes Lernverfahren soll hier das „bestärkende Lernen“ vorgestellt werden. Die Idee hierbei ist, den Lernenden für korrekte Aktionen zu belohnen (reward) und ihn für fehlerhafte Aktionen zu bestrafen (punish) ([AP02]). Das Netz erhält Eingaben und die zugehörigen Ausgaben, die hier lediglich binäre Ausgaben sind, wobei eine „1“ eine korrekte Ausgabe und eine „0“ eine falsche Ausgabe bezeichnet. Auch in der Biologie gibt es solche Vorgänge, wie „Erfolg“ und „Misserfolg“, so dass dieses Verfahren näher am biologischen Vorbild ist, als das „überwachte Lernen“. Des Weiteren hat der Lernende keine Ahnung, welche Aktionen ihn zu einem korrekten Ergebnis führen, so dass er nach dem Prinzip „try-and-error“ versucht herauszufinden, bei welchen Gewichtswerten er die die höchste Auszeichnung erhält. Der Ablauf eines „bestärkenden Lernverfahrens“ kann wie folgt beschrieben werden:

1. Der Lernende erhält ein Eingabemuster.
2. Der Lernende führt, nach der Interpretation der Eingabe, eine Aktion aus und produziert eine Ausgabe.
3. Ein außen stehender Bewerter gibt dem Lernenden Rückmeldung über die Korrektheit seiner Aktion (Reward/Punish).
4. Der Lernende aktualisiert seine Gewichte mit folgender Gewichtskorrektur:

   $$\delta_{w_{ij}} = \eta(r_p - \theta)e_{ij}$$

   Hier bezeichnet η die Lernrate, r_p gibt an, ob das Ergebnis bezüglich der Eingabe p korrekt oder falsch ist, θ ist der Schwellenwert und e_{ij} gibt an, wie relevant die Eingabe für das zu ändernde Gewicht ist.

Eine Art des bestärkenden Lernens wird auch bei dem Spiel NERO verwendet.

C.5 Verwendung in Spielen

Eines der Hauptziele beim Einsatz neuronaler Netze in Computerspielen ist die Erzeugung von intelligenten Agenten, die lernfähig sind, ihr Verhalten aufgrund von Interaktionen mit der Umwelt verändern können, die effizienter werden, je mehr Zeit vergeht und die auf neue Situationen intelligent reagieren können ([MBC$^+$06]). Man versucht vor allem non-player-characters (NPCs) so zu implementieren, dass sie sich menschlich verhalten. Folgende Aufzählung zeigt mögliche Einsatzgebiete von neuronalen Netzen in Spielen:

- Navigation: Die Bewegung eines Agenten im Spiel ist eine der wichtigsten Tätigkeiten, die implementiert werden muss. Ein Mensch bewegt sich normalerweise sehr intuitiv und reagiert auf seine wahrgenommenen Objekte. D.h. man könnte neuronale Netze benutzen, um die Umgebung wahrzunehmen, um beispielsweise Kollisionen zu vermeiden.

- Kampf: Auch hier ist der Einsatz neuronaler Netze sehr interessant. Während bei deterministischen Algorithmen der Gegner beispielsweise durch komplettes Wissen über eine Situation entscheidet, ob er jemanden angreifen soll oder nicht, soll der Agent aufgrund von beobachteten Faktoren entscheiden, ob er eine Aktion durchführt oder nicht. Dies ähnelt dem menschlichen Verhalten.

- Strategie: Auch der Bereich des strategischen Denkens kann gut mit neuronalen Netzen dargestellt werden. Ein Mensch agiert strategisch, indem er aktuelle Beobachtungen auswertet, mit seinem Wissen verknüpft und dann eine Strategie wählt. Dies läßt sich auch durch neuronale Netze realisieren, wobei ein „Feuern“ eines Neurons (Strategie wählen), aufgrund von Beobachtungen (Eingaben) geschieht.

Die einfachste Weise neuronale Netze in Spielen einzusetzen, ist ein bestehendes Spiel zu erweitert („mod“ schreiben), in dem man dann beispielsweise einen neuen NPC erstellt, der mittels neuronaler Netze implementiert ist. Dies ändert nichts an der grundlegenden Struktur und es ist sehr einfach implementieren. In den folgenden Unterkapiteln wird der Einsatz von neuronalen Netzen an drei konkreten Spiele-Beispielen vorgestellt.

C.5.1 JoeBot - Ein Counterstrike-Bot

In diesem Abschnitt soll erläutert werden, wie neuronale Netze verwendet werden, um im Spiel Counterstrike (siehe [Wik07b]) „Bots“ (Computergegenspieler) zu erstellen, die menschlicher agieren als beispielsweise deterministisch programmierte Bots. Dieser sogenannte JoeBot, der von Johannes Lampel ([Lam01]) implementiert wurde, verwendet ein Offline-Lernverfahren, muss also außerhalb des Spiels trainiert werden. Neuronale Netze werden für folgende Bereiche verwendet.

1. Kampfphase: Das Kampf-NN hat 6 Eingaben und produziert 5 Ausgaben. Es wird mit Hilfe des Backpropagation-Lernverfahren trainiert. Die Eingaben des Netzes sind „Eigene Gesundheit“, „Gegnerentfernung“ (falls Gegner sichtbar), „Waffenbeschreibung des Gegners“ (falls Gegner sichtbar), „Eigene Waffe“, „Munition“ und die „Momentane Situation“ (siehe Text). Das Netz produziert folgende Ausgaben: „Springen oder Ducken“, „Verstecken“, „Links/Rechts gehen“, „Angreifen“. Alle Eingaben befinden sich im Bereich [-1,1]. Die Neuronen haben Schwellenwertfunktionen als Aktivierungsfunktionen, so dass sich der Bot beispielsweise bei einem Schwellenwert von 0,5 duckt. Die Eingaben bei der Waffenbeschreibung haben fest definierte Werte. So besitzt das Messer (als schwächste Waffe) den Wert -1 und das Scharfschützengewehr (stärkste Waffe) den Wert 1, so dass ein Messer als eigene Waffe kaum einen Einfluss auf die Aktion des Bots hat, während stärkere Waffen einen stärkeren Einfluss haben. Unter der „momentanen Situation“ versteht man die Anzahl der lebenden Gegner und eigenen lebenden Mitspieler, so wie die bisherige Spielweise des Bots (aggressiv/defensiv). Auch ist menschliches Verhalten implementiert, wie man am Beispiel der Langeweile sehen kann. Wenn JoeBot längere Zeit nichts getan hat, dann zerschiesst er wahllos Objekte, was auf der menschlichen Empfindung „Langeweile“ beruht.

2. Kollisionsvermeidung: Das Netz zur Vermeidung von Kollisionen ist ein Feedforward-Netz und hat drei Eingaben und eine Ausgabe. Die Eingaben sind „Entfernung

links“, „Entfernung Mitte“ und „Entfernung rechts“. Als Ausgabe wird eine Bewegung „links, rechts oder geradeaus“ produziert. Das Netz wird immer dann aktiv, wenn ein Objekt in Reichweite ist. Im offenen Gelände ist es meist inaktiv, jedoch in engeren Gassen fast durchgehend aktiv. Der Bot dreht sich, je nachdem welche Entfernung gemessen wird, mehr oder weniger. In Abbildung 38 ist die Funktionsweise des Kollisionsnetzes visuell dargestellt.

Abbildung 38: Kollisions-NN JoeBot visuelle Darstellung (entnommen aus [Lam01])

Training der neuronalen Netze
Da hier ein offline-Lernverfahren verwendet wurde, wurden den neuronalen Netzen verschiedene Muster bereitgestellt. Um zu überprüfen, ob die neuronalen Netze gut arbeiten, sammelte der Autor Johannes Lampel alle Eingabemuster des Kampf-NN während des Spiels und erstellte daraus eine selbtorganisierende Karte (siehe Kapitel C.3.2). Nun wurden alle vorhandenen definierten Trainingsmuster auf die SOM projiziert. Aufgrund der Gewichtsverteilung, sowie den Distanzen zwischen der Trainingseingabe und der Eingabe im Spiel, konnten Muster, die kaum erkannt wurden, bei denen also der Abstand sehr groß war, gefunden werden und so der Bot optimiert werden. An JoeBot sieht man, dass neuronale Netze erfolgreich eingesetzt werden können, um menschliche Reaktionen (Kampfphase), sowie Wahrnehmung (Kollisionserkennung) zu implementieren.

C.5.2 Pacman

Pacman ist ein Spiel, bei dem eine Gruppe von Geistern versucht den „Pacman“zu fangen, während er versucht im Labyrinth alle Punkte auf zu sammeln. Traditionell wurden in diesem Spiel deterministische Methoden benutzt, um für die Geister die Suchalgorithmen zu implementieren. Auch hier kann man neuronale Netze dazu verwenden, optimale Suchwege für die Geister zu erstellen. Kalyanpur und Simon entwickelten ein rekurrentes neuronales Netz für Pacman. Es bestand aus einer hidden-Schicht, besaß zwei Eingabeneuronen und hatte ein Ausgabeneuron. Als Lernverfahren wurde das Backpropagation-Lernverfahren (siehe C.4.1) benutzt, um die Gewichte anzupassen. Für ausführliche Informationen bezüglich Pacman und künstlicher neuronaler Netze siehe den Projektbericht [KS01b] von Kalyanpur und Simon.

C.5.3 NERO

In NERO (Neuro Evolving Robotic Operatives, siehe [Aus07]) besitzen die Roboter, die trainiert werden, künstliche neuronale Netze zum Denken und Handeln (Gehirn). Sie lernen durch den Algorithmus NEAT (Neuro-Evolution of Augmenting Topologies), der eine Art „reinforcement learning Algorithmus“ (siehe Kapitel C.4.3) ist. Die Agenten (Roboter) lernen, indem man sie für gute Handlungen belohnt und für schlechte bestraft. In NERO wird das Bestrafen und Belohnen dem Spieler überlassen, der über Schieberegler in der GUI die Handlungen definieren kann, für die der Roboter entweder belohnt oder bestraft wird. Dabei gibt es die Handlungen „Approach Enemy“ (nähere dich dem Gegner), „Hit Target“ (Feuere auf Gegner), „Avoid Fire“ (Ausweichen), „Approach Flag“ (nähere dich der Flagge), „Stick Together“ (zusammen bleiben) und „Stand Ground“ (Stellung halten). Der Algorithmus entscheidet nun, welche neuronalen Netze die Bedingungen am besten erfüllen. Dabei wird der beste Roboter belohnt und der schlechteste Roboter bestraft.

Der eingesetzte Algorithmus NEAT (Neuro-Evolution of Augmenting Topologies) startet im Gegensatz zu bisherigen Ansätzen mit einem künstlichen neuronalen Netz, was minimal verbunden ist und fügt dort Komplexität hinzu (neue Kanten, andere Gewichte etc), wo sie hilft das Problem zu lösen. Dadurch soll verhindert werden, dass zu Beginn viel zu komplexe Netze erzeugt werden, die für die Problemlösung nicht optimal sind. Des Weiteren wird in NERO eine Echtzeit-Variante des Algorithmus NEAT, genannt rtNEAT, eingesetzt, in der der Spieler einer kleinen Population von Robotern beim Lernen zu sehen kann.

Als Eingaben besitzen die neuronalen Netze der Roboter „EnemyRadar“, „OnTarget“, „Object Rangefinders“, „EnemyDetails“ und als Ausgaben werden „Left/Right“, „Forward/Backward“ und „Fire“ produziert. Das „EnemyRadar“ zeigt dem Roboter die Positionen aller Gegner an, die in Sichtweite sind. Unter der Eingabe „OnRadar“ versteht man die Eigenschaft, ob ein Gegner in Schusslinie (auf 12 Uhr) ist. Des Weiteren können die Roboter Objekte und Wände finden, die in Sichtweise sind und beurteilen in welche Richtung ein Gegner beispielsweise schießt (Enemy Details). Die Ausgaben bei NERO bestehen aus der Bewegung der Roboter und der Schusskontrolle. NERO ist ein sehr interessantes Spiel, um den Einsatz künstlicher neuronaler Netze in Spielen in Echtzeit zu beobachten, da man in Echtzeit als Lehrer seinen Robotern etwas beibringen kann und das Gelernte danach in Spielszenarien beobachten kann.

C.6 Fazit

Neuronale Netze sind mächtige Konstrukte im Bereich des „Computational Intelligence“. Sie eignen sich hervorragend diverse Funktionen zu realisieren, die kombiniert fast jedes gewünschte Verhalten nachahmen können. Der Lernprozess eines Gehirns beispielsweise, kann mit den Lernregeln, die in Kapitel C.4 vorgestellt wurden, verglichen werden. Dies macht neuronale Netze zu einer interessanten Anwendung in Spielen, denn vor allem das Spiel gegen „menschliche“ Gegner macht Spaß. Wie in Kapitel C.5 beschrieben wurde, können neuronale Netze erfolgreich „menschliches“ Verhalten erzeugen. Als letztes sei auf Kapitel C.1 verwiesen, in dem die Beziehung zum menschlichen Gehirn (Nervensystem) beschrieben wurde.

D Machine Learning, Regelbasierte Steuerung

D.1 Einleitung

D.1.1 Logisches Schließen

Nach Charles S. Peirce [Pei65, Kapitel 5] gibt es die drei Inferenzarten Deduktion, Abduktion und Induktion. Dabei sind Abduktion und Induktion unsichere Schlussweisen, wie durch die Beispiele deutlich wird. In den Beispielen wird eine Schreibweise verwendet, in der die Prämissen über dem Folgerungsstrich stehen und die Konklusion unterhalb.

Deduktion: Aus gegebenem Wissen W wird B sicher gefolgert. Beispiel:

$$\begin{array}{lcl} W & : & \text{Pinguin} \Rightarrow \text{Vogel} \\ W & : & \text{Pinguin} \\ \hline B & : & \text{Vogel} \end{array}$$

Abduktion: Suche nach Erklärung E für Beobachtung B bei gegebenem Wissen W. Beispiel:

$$\begin{array}{lcl} W & : & \text{Hasen haben lange Ohren} \\ B & : & \text{Max hat lange Ohren} \\ \hline E & : & \text{Max ist ein Hase} \end{array}$$

Induktion: Aus vielen Beispielen B_i wird allgemeines Wissen W abgeleitet. Beispiel:

$$\begin{array}{lcl} B_1 & : & \text{Vogel1 fliegt} \\ B_2 & : & \text{Vogel2 fliegt} \\ \vdots & & \\ \hline W & : & \text{Vögel fliegen} \end{array}$$

D.1.2 Klassische und nicht-klassische Inferenzsysteme

In Spielen und in vielen alltäglichen Situationen besteht die Notwendigkeit, unsichere Folgerungen und unsicheres Wissen auszudrücken. Dazu sind klassische Logiken wegen ihrer Monotonieeigenschaft nicht in der Lage. Die Eigenschaft der Monotonie besagt, dass, wenn eine bestimmte Aussage aus einer Menge von Prämissen folgt, sie immer noch folgt, wenn weitere Prämissen hinzugenommen werden.

Zum Verständnis regelbasierter Systeme sind einige Grundlagen zu logischem Schließen hilfreich. Eine gute Einführung in die meisten hier angesprochenen Themen bieten Beierle und Kern-Isberner [BKI03]. Auf Seite 71 werden Regeln folgendermaßen definiert:

> Regeln sind formalisierte Konditionalsätze der Form
>
> Wenn (**if**) A dann (**then**) B
>
> mit der Bedeutung
>
> Wenn A wahr (erfüllt, bewiesen) ist, dann schließe, dass auch B wahr ist.

Eine Regel, die immer gilt, heißt deterministisch. Als regelbasierte Systeme, die in Abschnitt D.3 behandelt werden, werden heutzutage nur solche bezeichnet, die aus deterministischen Regeln bestehen. Auftretende Widersprüche werden in regelbasierten Systemen daher nur heuristisch aufgelöst. Jüngere, nicht-klassische Ansätze gehen von vornherein von unsicheren Schlussweisen aus.

Defaultlogik Eine Regel δ der Reiter'schen Defaultlogik, eingeführt von Reiter [Rei80], wird syntaktisch folgendermaßen formuliert:

$$\delta = \frac{\varphi : \psi_1, \ldots, \psi_n}{\chi}$$

Hierbei sind φ, ψ_i und χ geschlossene aussagenlogische oder prädikatenlogische Formeln. Die semantische Bedeutung dieser Regel entspricht: „Wenn φ bekannt ist und $\psi_1, \ldots, \psi_n$ konsistent angenommen werden können, dann folgere χ". Diese Logik versucht also Sachverhalte auszudrücken, die „meistens gelten" oder die „solange gelten, bis das Gegenteil bekannt wird".

Probabilistisches Schließen Beim Probabilistischem Schließen werden Regeln durch bedingte Wahrscheinlichkeiten repräsentiert. Aus diesen Regeln kann man dann ein Bayes'sches Netzwerk aufbauen, das die bedingten Unabhängigkeiten repräsentiert. Die bedingte Wahrscheinlichkeit für B gegeben A ist

$$P(B|A) = \frac{P(A \wedge B)}{P(A)} = \frac{P(A|B) \cdot P(B)}{P(A)}.$$

Es gilt $P(B|A) \leq P(A \Rightarrow B) = P(\neg A \vee B)$. Beispielsweise sei $P(A) = 0,01$ und $P(B) = 0,95$ und A und B haben die gemeinsame Verteilung

A	B	$P(\cdot)$
0	0	0,04
0	1	0,95
1	0	0,01
1	1	0

Dann ist $P(B|A) = 0$, aber $P(A \Rightarrow B) = 0,99$.

D.2 Maschinelles Lernen

Ein Lernvorgang wird von Russell und Norvig [RN04, Seite 793] folgendermaßen definiert:

> Das Konzept hinter dem Lernen ist, Wahrnehmungen nicht nur für das aktuelle Handeln zu verwenden, sondern auch, um zukünftige Handlungen des Agenten zu verbessern. Lernen findet statt, wenn der Agent seine Interaktion mit der Welt und seine eigenen Entscheidungsprozesse beobachtet.

Dabei ist ein Agent „alles, was seine Umgebung über Sensoren wahrnehmen kann und in dieser Umgebung durch Aktuatoren handelt“ [RN04, Seite 793]. Lernformen unterscheiden sich vor allem im Umfang des Feedbacks, der zur Verfügung steht, um das Verhalten zu verbessern. In dieser Hinsicht gibt es drei Kategorien von Lernverfahren: nicht überwachtes, verstärkendes und überwachtes Lernen.

Der Fall unüberwachten Lernens liegt vor, wenn gar kein Feedback über das bisherige Verhalten verfügbar ist. Daher ist in einem rein unüberwachtem Lernprozess keine Beurteilung des Lernerfolgs möglich.

Bei überwachtem Lernen steht eine Trainigsmenge von bereits vorklassifizierten Beispielen zur Verfügung. Anhand dieser kann man eine Funktion ermitteln, die den Fehler bezüglich der Beispiele minimiert. Dass Beispiele zur Verfügung stehen, bedeutet, dass die Umgebung des Agenten vollständig beobachtbar ist und er somit die Wirkung seiner Aktionen beobachten und vorhersagen kann.

Verstärkendes Lernen bedeutet Lernen durch Belohnung oder Bestrafung von bisherigem Verhalten. Der Lernende erhält also Informationen darüber, wie gut oder schlecht seine Aktionen sind, allerdings nicht, was genau daran falsch ist. Außerdem besteht neben der eigentlichen Lernaufgabe meist das Unterproblem, zu lernen, wie die Umgebung funktioniert, da man erwartet, durch das Verständnis der Umgebung einen größeren Lernerfolg zu erzielen. Aus Platzgründen wird hier auf verstärkendes Lernen nicht weiter eingegangen.

In der Realität ist so eine scharfe Trennung von Lernverfahren nicht möglich. Bereits am Beispiel Clustering werden wir sehen, dass bereits durch die Wahl eines Algorithmus gewisse Informationen in den Lernprozess eingehen.

D.2.1 Unüberwachtes Lernen: Clustering

Unter Clustering versteht man die unüberwachte Einteilung von Beobachtungen in Klassen. Von der Klassenaufteilung werden zwei Eigenschaften gefordert:

- Ähnlichkeit innerhalb von Klassen (intra-class-similarity)
- Unähnlichkeit zwischen verschiedenen Klassen (inter-class-dissimilarity)

Dies bedeutet also, es sollen keine ähnlichen Beobachtungen in verschiedene Klassen eingeteilt werden und keine unähnlichen Beobachtungen in einer Klasse versammelt werden. Zur Beurteilung der Ähnlichkeit von Beobachtungen muss ein Distanzmaß gewählt werden. Häufig wird z.B. die Euklidische Distanz gewählt.

Es gibt viele verschiedene Clusteringverfahren, aber keines, dass allgemein besser als andere ist. Da es sich um unüberwachte Lernverfahren handelt, müssen Unterschiede in den Fähigkeiten zur Erkennung von Clusterformen in impliziten Annahmen der Verfahren bezüglich der Form der Cluster liegen. Zum Beispiel würde ein graphentheoretisches Clusteringverfahren, das einen minimalen Spannbaum auf der Beobachtungsmenge berechnet und dann die längste Kante entfernt, mit hoher Wahrscheinlichkeit Cluster in Form von konzentrischen Kreisen wie in Abb. 39 erkennen. Das Verfahren k-means, das im nächsten Abschnitt beschrieben wird, ist dazu nicht in der Lage. Auch die Datenrepräsentation hat Einfluß. Viele Verfahren können z.B. eine kreisförmige Punktwolke um den Koordinatenursprung nicht als eine solche einzelne erkennen, wenn die Beobachtungen in einem

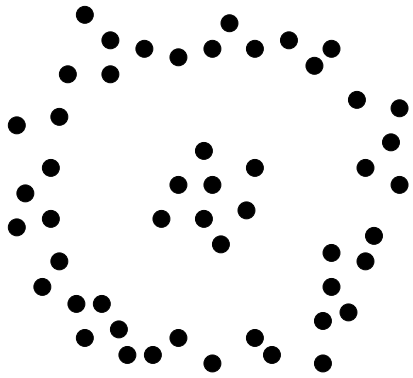


Abbildung 39: Konzentrische Kreise

karthesischen Koordinatensystem vorliegen, wohl aber wenn die Punkte in Polarkoordinaten dargestellt werden.

k-means Clustering k-means ist ein Clusteringverfahren, bei dem zur Initialisierung die Anzahl der Cluster fest vorgegeben werden muss, indem man für jeden Cluster einen Startzentroid definiert. Ein Zentroid stellt den Mittelpunkt des Clusters dar. Da das Verfahren abhängig von den initialen Zentroiden in lokale Optima konvergieren kann, führt man in der Praxis mehrere Durchläufe mit zufälliger Initialisierung durch. Es ist auch möglich, dass im Verlauf des Algorithmus Cluster leer bleiben und somit degenerieren, da es nicht mehr möglich ist, Zentroide für solche Cluster zu berechnen. Der Algorithmus läuft folgendermaßen ab:

1. Wähle k beliebige Beobachtungen als Zentroide von k Clustern.
2. Ordne jede Beobachtung dem nächsten Zentroid zu.
3. Berechne neue Zentroide der so entstehenden Cluster (z.B. als Mittelwerte).
4. Falls sich nun die Partitionierung der Menge geändert hat, weiter mit Schritt 2, ansonsten Abbruch.

Zur Bewertung der Partitionierung wird häufig der Quadratfehler

$$V = \sum_{i=1}^{k} \sum_{x_j \in C_i} |x_j - \mu_i|^2$$

(mit k Anzahl der Cluster C_i, $i \in \{1, 2, \ldots, k\}$, μ_i Zentroid des Clusters C_i und x_j Beobachtungen in C_i) gewählt. Das Verfahren ist einfach zu implementieren und hat sich in der Praxis als sehr schnell erwiesen, wodurch die oben erwähnte Mehrfachausführung möglich wird. Trotz der guten praktischen Ergebnisse hat k-Means allerdings eine worst-case Laufzeitschranke von $2^{\Omega(\sqrt{n})}$, wie von Arthur und Vassilvitskii [AV06] gezeigt wurde.

D.2.2 Überwachtes Lernen: Entscheidungsbäume

Abbildung 40 stellt einen Entscheidungsbaum dar, wie ihn jeder Mensch unbewusst benutzen könnte, um sich die Frage zu beantworten, ob es notwendig ist, Lebensmittel einkaufen zu gehen. Das wichtigste Kriterium steht an der Wurzel. Weitere Kriterien werden nur benutzt, wenn die bisher betrachteten Kriterien keine eindeutige Entscheidung hervorgebracht haben.

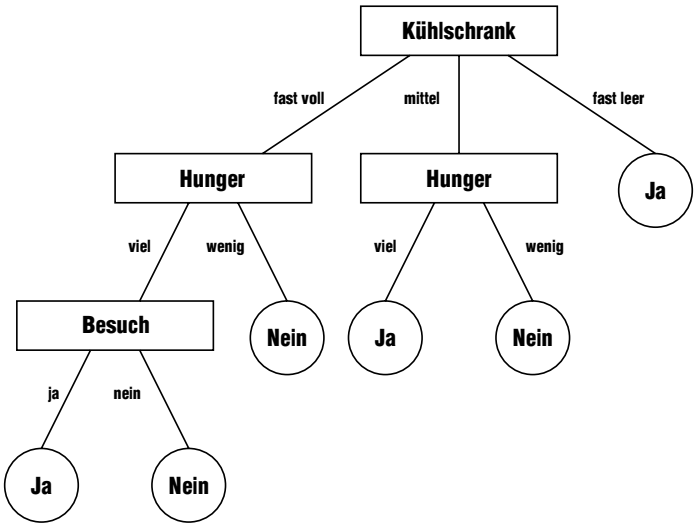


Abbildung 40: Ein Entscheidungsbaum zur Beantwortung der Frage „Einkaufen gehen?“

Entscheidungsbäume können jede beliebige boolesche Funktion darstellen. Dies ist auf triviale Art möglich, indem jede Zeile der Wahrheitstabelle der Funktion durch einen Pfad von der Wurzel zu einem Blatt im Entscheidungsbaum repräsentiert wird. Da die Wahrheitstabelle für eine Funktion mit n Variablen 2^n Zeilen enthält, wäre auch ein so entstandener Entscheidungsbaum exponentiell groß.

Es lassen sich zwar viele Funktionen durch kompakte Entscheidungsbäume darstellen, es gibt aber Funktionen, für die ein Entscheidungsbaum tatsächlich exponentielle Größe annimmt. Zum Beispiel lässt sich eine Paritätsfunktion, die 1 zurückgibt, wenn die Anzahl der Einsen in der Eingabe gerade ist, nur durch einen Entscheidungsbaum exponentieller Größe darstellen.

Dieses Phänomen ist allerdings kein spezielles Problem von Entscheidungsbäumen, denn es gibt keine Repräsentation, die für alle Arten von Funktionen effizient ist, wie von Russell und Norvig [RN04, Seite 800] gezeigt wird.

Entscheidungsbäume werden dazu benutzt, aus einer Trainingsmenge eine möglichst einfache Funktion zu extrahieren, die die Beispiele der Trainingsmenge korrekt klassifiziert und so allgemein ist, dass sie auch für zukünftige Fälle korrekt arbeitet. Der Lernvorgang findet also durch das Bilden einer induktiven Folgerung statt.

Top-down Induction of Decision Trees Bei diesem Verfahren wird der Entscheidungsbaum induktiv von der Wurzel her aufgebaut. Jeder rekursive Aufruf erhält als

Eingabe die Menge der noch nicht klassifizierten Beispiele und die Menge der noch zur Verfügung stehenden Attribute. An den aktuellen Blattknoten können dann vier Fälle auftreten. Im ersten Fall ist die Beispielmenge am aktuellen Blatt leer. Das Attribut, an dem dieses Blatt hängt, bekommt dann die Klassifikation, die die Mehrheit der Beispiele der nächsthöheren Ebene besitzt. Die Rekursion ist dann in diesem Zweig beendet.

Im zweiten Fall haben alle Beispiele am aktuellen Blatt die gleiche Klassifikation. Auch in diesem Fall ist die Rekursion beendet, da alle Beispiele in diesem Zweig korrekt klassifiziert sind. Im dritten Fall ist der Baum nicht erzeugbar, da in der Beispielmenge nicht genug Informationen vorhanden sind, um alle Beispiele zu klassifizieren. Es sind also noch positive und negative Beispiele, aber keine Attribute zur weiteren Unterteilung mehr vorhanden. Der vierte Fall führt den rekursiven Aufruf durch. Dabei wird die Beispielmenge ihrer Attribute entsprechend auf die Teilbäume aufgeteilt. Für jeden Zweig muss nach einem Kriterium das nächste Attribut ausgewählt werden. Dabei kommen nur Attribute in Frage, die noch nicht oberhalb des neuen Teilbaums verwendet wurden.

Für die Wahl des nächsten Attributs sind verschiedene Auswahlkriterien denkbar. Das einfachste Verfahren ist die Auswahl nach dem Kardinalitätskriterium, nach dem dasjenige Attribut gewählt wird, welches die meisten Beispiele korrekt klassifiziert. Großer Nachteil des Kardinalitätskriteriums ist dessen große Abhängigkeit von der aktuell betrachteten Teilmenge der Beispiele. Alternativ kann man die Attribute nach der in ihnen enthaltenen Information beurteilen.

Im vierten Fall des Algorithmus würde also das Attribut gewählt, welches den maximalen Informationsgewinn bietet. Die Auswahl nach diesem Kriterium bevorzugt allerdings Attribute mit vielen Ausprägungen. Daher ist es ratsamer, stattdessen den über die Anzahl der Ausprägungen normierten Informationsgewinn zu benutzen. All diese Verfahren werden von Beierle und Kern-Isberner in [BKI03, Kapitel 5] ausführlich beschrieben.

Aus einem Entscheidungsbaum lässt sich außerdem direkt eine widerspruchsfreie Regelbasis ableiten. Jeder Pfad von der Wurzel zu einem Blattknoten entspricht einer **if-then** Regel. Dies kann man sich zum Beispiel zunutze machen, um ein neuronales Netz aus einem Entscheidungsbaum zu generieren.

Initialisierung von neuronalen Netzwerken durch Entscheidungsbäume Ivanova und Kubat stellen in [IK95] ein Verfahren vor, um aus einem Entscheidungsbaum ein neuronales Netz zu erzeugen. Neuronale Netze, beschrieben z.B. von Nauck et al. [NKK94], können ebenso wie Entscheidungsbäume zur Klassifikation eingesetzt werden. Das Verfahren wird so angegeben:

1. Generiere aus einer Teilmenge der Beispielmenge einen Entscheidungsbaum.
2. Forme den Entscheidungsbaum in eine Regelbasis um und übersetze diese Regelbasis in ein neuronales Netzwerk, in dem nur die diesen Regeln entsprechenden Kanten existieren.
3. Füge weitere Kanten ein, so dass jeweils zwei aufeinanderfolgende Schichten vollständig vernetzt sind. Diese Kanten erhalten kleine initiale Gewichte ϵ.
4. Verändere alle Gewichte leicht per Zufall.

5. Setze in allen Neuronen sigmoide Aktivierungsfunktionen ein. Überführe die Attributwerte in unscharfe Intervallzugehörigkeitsfunktionen.

6. Trainiere das Netzwerk mittels des Backpropagation-Algorithmus unter Verwendung aller Trainingsbeispiele.

Die zugrundeliegende Idee ist, dass ein Entscheidungsbaum gut dazu geeignet ist, einfache Regeln abzubilden. Nachdem die Funktion des Entscheidungsbaums in ein neuronales Netz überführt wurde, soll sie durch die ausdrucksstärkere Beschreibungssprache des Netzes verfeinert werden. Dies geschieht in den Schritten 3–6. Wie die Autoren experimentell zeigen, trägt jede der beschriebenen Maßnahmen einen Teil zur Verbesserung der Fehlerrate bei.

D.3 Regelbasierte Steuerung

Von regelbasierter Steuerung ist die Rede, wenn ein Agent durch ein regelbasiertes System gesteuert wird. Ein regelbasiertes System kann zum Beispiel automatisch aus einem Entscheidungsbaum generiert oder von einem Experten entworfen worden sein. Der Zweck eines regelbasierten Systems ist also, dieses Expertenwissen abzubilden beziehungsweise Verhalten nachzuahmen.

Im Gegensatz zum herkömmlichen, prozeduralen Ansatz sind in einem regelbasierten System Regelbasis *(rule base, rule memory)* und Faktenwissen *(working memory)* getrennt. Das Verzichten auf eine Strukturierung der Regelbasis führt zu einer langsamen Laufzeit bei der Auswertung, bietet aber andererseits Vorteile. Vor allem kann Wissen durch den modularen Aufbau leicht hinzugefügt und entfernt werden und man kann durch Zurückverfolgung der Regeln leicht herausfinden, auf welchem Weg das Programm zu einer bestimmten Schlussfolgerung gelangt ist. Regelbasen können auf zwei Arten inkonsistent sein:

1. Die Regelbasis ist klassisch-logisch inkonsistent, d.h. es gibt keine Belegung der Objekte mit Werten, so dass alle Regeln erfüllt sind.

2. Die Regelbasis führt zu widersprüchlichen Ableitungen.

Der erste Fall tritt zum Beispiel beim Regelsystem

$$\begin{array}{l}\textbf{if } A \textbf{ then } B\\ \textbf{if } \neg A \textbf{ then } B\\ \textbf{if } B \textbf{ then } A\\ \textbf{if } B \textbf{ then } \neg A\end{array}$$

auf, wie folgende Wahrheitstabelle zeigt:

A	B	$A \Rightarrow B$ $\equiv \neg A \vee B$	$\neg A \Rightarrow B$ $\equiv A \vee B$	$B \Rightarrow A$ $\equiv \neg B \vee A$	$B \Rightarrow \neg A$ $\equiv \neg B \vee \neg A$
0	0	1	0	1	1
0	1	1	1	0	1
1	0	0	1	1	1
1	1	1	1	1	0

Es gibt keine Zeile ($\widehat{=}$ Variablenbelegung) in der Tabelle, in der alle Regeln erfüllt sind. Ein Beispiel für den zweiten Fall wäre

$$\textbf{if } V \textbf{ then } F$$
$$\textbf{if } V \wedge P \textbf{ then } \neg F$$

wo gleichzeitig F und $\neg F$ gefolgert werden, wenn V und P wahr sind. Die beiden Regeln alleine sind noch nicht widersprüchlich, sondern werden dies erst durch das Hinzufügen der Fakten V und P. Es müsste also sichergestellt sein, dass V und P nicht gleichzeitig gelten können, wenn man Konsistenz erreichen will.

Üblicherweise wird bei der Herstellung von Spielen nicht nur Faktenwissen und Regelbasis getrennt, sondern auch der gesamte KI-Programmteil aus der Programmlogik ausgelagert. Stattdessen wird die KI separat in einer Scriptsprache formuliert und dann während des Programmablaufes geparst. Dieser Ansatz ist daher als „Scripted AI“ oder „Scripting“ bekannt. Ein Grund für dieses Vorgehen ist der, dass so häufige Kompiliervorgänge zur Justierung der KI vermieden werden und die Arbeit nicht unbedingt von Programmierern übernommen werden muss. Ausführlicher wird Scripting von Bourg und Seeman [BS04, Kapitel 8] behandelt.

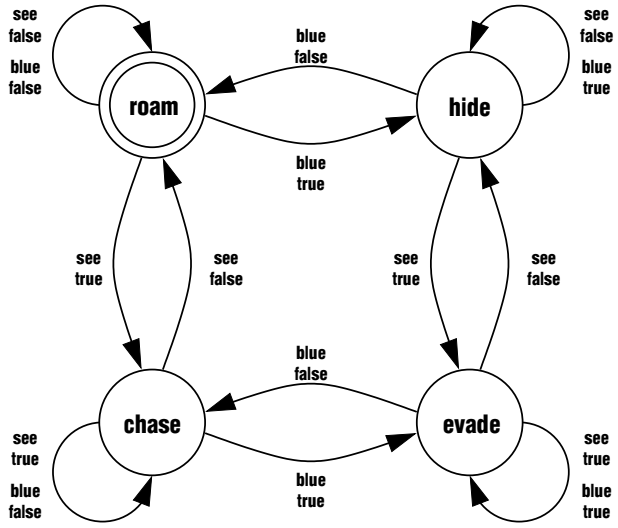


Abbildung 41: Ein Geist im Spiel Pacman.

Zusammenhang mit endlichen Automaten Besonders einfache Agenten lassen sich als endliche Automaten darstellen, welche wiederum eine andere Darstellungsweise eines regelbasierten Systems sind. Der Automat aus Abb. D.3 soll das Verhalten eines Geistes im Spiel Pacman modellieren. Der Geist kann entweder ziellos umherstreifen *(roam)*, Pacman jagen *(chase)*, vor Pacman fliehen *(evade)* oder sich vor Pacman verstecken *(hide)*. Wenn Pacman einen Zauberpunkt isst, wird der Geist blau und vom Jäger zum Gejagten. Weitere Zustandstransitionen finden statt, wenn Pacman gesichtet oder verloren wird und wenn die Wirkung des Zauberpunktes nachlässt. Wie das Regelsystem in Listing 9 aus Abb.

D.3 hervorgeht, ist offensichtlich: Für jeden Zustandsübergang wird eine Regel benötigt. Transitionen, die keinen Zustandswechsel bewirken, können weggelassen werden.

```
if state = roam and blue = true then state = evade
if state = roam and see = true then state = chase

if state = chase and blue = true then state = evade
if state = chase and see = false then state = roam

if state = evade and blue = false then state = chase
if state = evade and see = false then state = hide

if state = hide and blue = false then state = roam
if state = hide and see = true then state = evade
```

Listing 9: Der Geist in Form einer Regelbasis

D.3.1 Vereinfachung von Regeln

In regelbasierten Systemen wird der Einfachheit halber nur der Modus ponens

if A **then** B	(Regel)
A wahr	(Faktum)
B wahr	(Schlussfolgerung)

angewendet. Dass die Benutzung des Modus tollens

if A **then** B	(Regel)
B falsch	(Faktum)
A falsch	(Schlussfolgerung)

nicht erlaubt ist, führt dazu, dass, wenn **if** A **then** B in der Regelbasis enthalten ist und $\neg B$ bekannt ist, $\neg A$ nicht abgeleitet werden kann. Um ein regelbasiertes System mit aussagenlogischem Verhalten zu erreichen, muss man also auch die Regel **if** $\neg B$ **then** $\neg A$ explizit in die Regelbasis aufnehmen. Diese Beschränkung halbiert den Auswertungsaufwand einer Regel.

Um den Auswertungsaufwand weiter zu verringern, ist es erstrebenswert, die syntaktische Form von Regeln möglichst einfach zu halten. Häufig stellt man daher zwei Bedingungen an die Form von Regeln:

- Es sind keine Disjunktionen im Regelrumpf erlaubt.
- Im Folgerungsteil ist nur ein einziges Literal zugelassen.

Unter Verwendung der Distributivgesetze und der de Morgan'schen Regeln ist es immer möglich, die Prämisse einer Regel in disjunktive Normalform und die Konklusion in konjunktive Normalform zu bringen. Solche Regeln wiederum kann man durch wiederholte Anwendung folgender zwei Umformungen in die gewünschte Form bringen.

- Ersetze eine Regel

  if $K_1 \vee \cdots \vee K_n$ **then** $D_1 \wedge \cdots \wedge D_m$

  durch die $n \cdot m$ Regeln

  if K_i **then** D_j

- Ersetze eine Regel

  if K **then** $L_1 \vee \cdots \vee L_p$

  (wobei K eine Konjunktion von Literalen ist) durch die p Regeln

  if $K \wedge (\bigwedge_{k \neq m} \neg L_k)$ **then** L_m , $m \in \{1, \ldots, p\}$

Die zweite Vorschrift ist nicht sofort einsichtig, wird aber durch folgende Äquivalenzumformung klar:

$$\begin{aligned} & K \Rightarrow L_1 \vee \cdots \vee L_i \vee \cdots \vee L_p \\ \equiv\ & \neg K \vee (L_1 \vee \cdots \vee L_i \vee \cdots \vee L_p) \\ \equiv\ & \neg K \vee (L_1 \vee \cdots \vee L_{i-1} \vee L_{i+1} \vee \cdots \vee L_p) \vee L_i \\ \equiv\ & \neg(K \wedge \neg L_1 \wedge \cdots \wedge \neg L_{i-1} \wedge \neg L_{i+1} \wedge \cdots \wedge \neg L_p) \vee L_i \\ \equiv\ & K \wedge \neg L_1 \wedge \cdots \wedge \neg L_p \Rightarrow L_i \end{aligned}$$

D.3.2 Verkettung von Regeln

Rückwärtsverkettung Falls man nur an dem Wert einer bestimmten Variablen interessiert ist, wäre es übertrieben, alle möglichen Schlussfolgerungen zu ziehen. Die Rückwärtsverkettung beginnt daher mit einer Anfrage, ob ein bestimmtes Ziel erreicht wird. Von diesem ausgehend werden Regeln gesucht, die das Ziel in der Konklusion enthalten. Rekursiv wird dann das Verfahren für die Prämissen dieser Regeln angewendet. Kommt der Algorithmus so zu dem Ergebnis, dass durch die Fakten alle Voraussetzungen für die Zwischenziele und das Ziel gegeben sind, dann ist das Ziel erfüllt. Bei der Rückwärtsverkettung stellen Inkonsistenzen kein Problem dar, da nur getestet wird, ob es *irgendeinen* Pfad zum Ziel gibt.

Vorwärtsverkettung Das Prinzip der Vorwärtsverkettung besteht darin, sich das Faktenwissen herzunehmen und mittels der Regelbasis solange zu erweitern, bis ein Fixpunkt erreicht ist. Das Verfahren wird daher als *datengetrieben* bezeichnet. Sei F die Menge der bewiesenen Fakten.

1. Für jede Regel **if** A **then** B der Regelbasis überprüfe, ob A erfüllt ist. Wenn ja $F := F \cup \{B\}$.
2. Wiederhole Schritt 1, bis F durch den gesamten Schritt 1 nicht mehr vergößert wird.

Dieser Algorithmus geht davon aus, dass die Regelbasis konsistent ist. Um Widersprüche aufzulösen, kann man die Vorwärtsverkettung um eine Heuristik erweitern, welche von den in Konflikt stehenden Regeln ausgeführt werden soll. Mögliche Metaregeln zur Konfliktresolution sind:

- Wähle die speziellste Regel.
- Füge den Regeln Gewichte hinzu und wähle anhand dieser eine aus.
- Wähle die am längsten nicht benutzte Regel.
- Wähle eine Regel zufällig.

Der abgeänderte Algorithmus wird dadurch deutlich aufwändiger, da nun jedesmal alle Regeln betrachtet werden müssen um eine Regel auszuführen.

1. Für jede Regel **if** A **then** B der Regelbasis überprüfe, ob A erfüllt ist.
2. Wähle aus den anwendbaren Regeln anhand einer Metaregel eine aus. Falls keine anwendbar ist, stoppe.
3. Wende die ausgewählte Regel an.
4. Gehe zu Schritt 1.

Rete-Algorithmus Die zweite Version des Algorithmus zur Vorwärtsverkettung führt viele redundante Auswertungen der Regeln durch, da für jede ausgeführte Regel alle Regeln betrachtet werden. Da aber seit der jeweils letzten Betrachtung aller Regeln nur eine Regel ausgeführt wurde, kann man davon ausgehen, dass sich nur wenige Fakten geändert haben. Der Rete-Algorithmus, der von Forgy [For82] vorgestellt wurde, vermeidet überflüssige Auswertungen durch Zwischenspeichern der Ergebnisse.

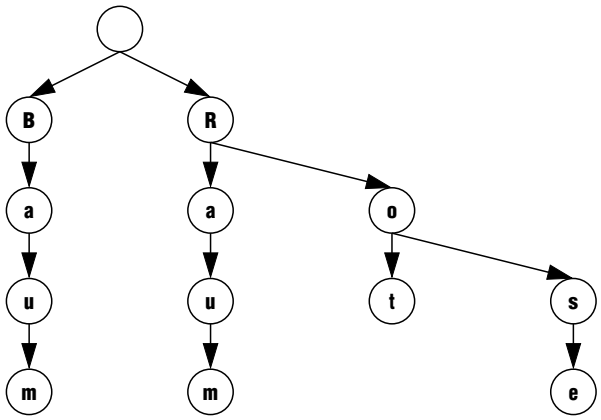


Abbildung 42: Prefix-Tree für die Zeichenketten Baum, Raum, Rot, Rose

Da der Algorithmus für ein Produktionssystem entwickelt wurde, in dem Objekte mit Mustern verglichen wurden, geht er davon aus, dass die Prämissen der Regeln strukturiert sind und die Faktenmenge in Form von Objekten vorliegt.

Eine mögliche Regel wäre zum Beispiel:

```
if (Buch = X and Status = ueberfaellig and Ausleiher = Y) and
   (Ausleiher = Y and Adresse = Z)
then
   (Aktion: Sende Nachricht an Z wegen Rueckgabe von X)
```

Um nicht für jede auszuwertende Regel die gesamte Faktenmenge durchlaufen zu müssen, erzeugt der Rete-Algorithmus einen Prefix-Tree aus den Prämissen. In Abb. 42 sieht man ein Beispiel für einen Prefix-Tree. Erreicht der Interpreter ein Blatt des Prefix-Trees, so wurde ein Objekt der Faktenmenge gefunden, welches dem durch den Pfad im Baum definierten Muster entspricht.

Von den Blättern des Baumes gehen Verbindungen zu sogenannten Join-Knoten ab. Auch die Join-Knoten werden mehrfach benutzt, wo immer dies möglich ist. Ein Join-Knoten stellt eine Und-Verknüpfung von zwei Mustern dar und enthält für jedes Muster eine separate Liste. Das Objekt wird dann, je nachdem ob es der Faktenmenge hinzugefügt oder aus ihr entfernt werden soll, in der entsprechenden Liste eingefügt oder entfernt. Diese Listen bilden den Zwischenspeicher, der dafür sorgt, dass zwischen zwei Auswertungszyklen nur die Veränderungen an der Faktenmenge wirklich neu ausgwertet werden.

Die Join-Knoten werden entsprechend den Regeln verschachtelt vernetzt. Ein an einem Join-Knoten eingehendes Muster kann also auch eine Und-Verknüpfung von einfacheren Mustern sein. Die Join-Knoten, bei denen eine Prämisse vollständig erfüllt ist, führen die Konklusion der Regel aus, was wiederum zu einer Änderung in der Konfliktmenge führen kann. Im Beispiel in Abb. 43 sind zwei Regeln enthalten, von denen die eine drei Objekte und die andere zwei Objekte in ihrem Voraussetzungsteil enthält. Zusätzlich sind die beiden mittleren Objekte teilweise identisch.

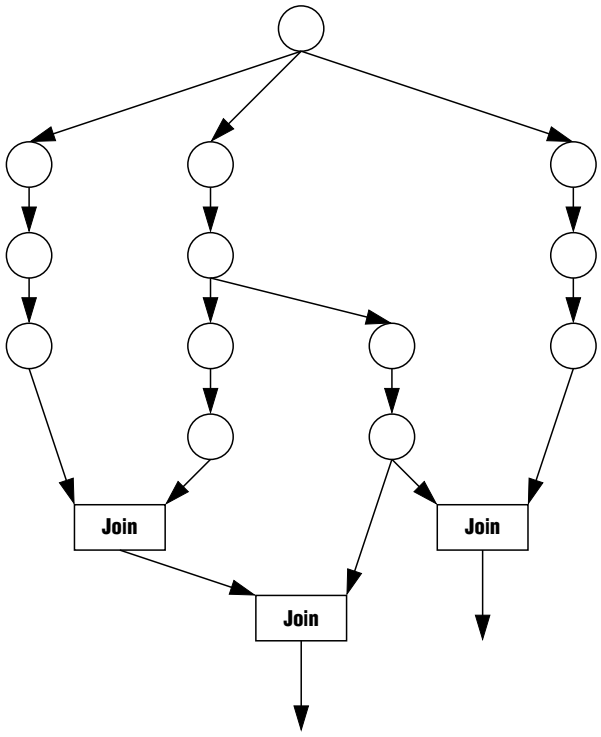


Abbildung 43: Schematisches Beispiel eines Rete-Netzwerkes

Da der naive Algorithmus zur Vorwärtsverkettung für reale Regelbasen nicht praxistauglich ist, kommen meist nur der Rete-Algorithmus oder andere schnelle Algorithmen in Frage. Implementierungen des Rete-Algorithmus sind zum Beispiel in C (CLIPS) oder

Java (Jess) für wissenschaftliche Zwecke kostenlos verfügbar.

D.3.3 Anwendungsbeispiel: Beat-'em-up-Spiel

Abbildung 44: Lycos Prügelpause, ein E-Mail-basiertes Browserspiel (2003 eingestellt). Der Herausforderer legte eine Reihenfolge aus fünf Schlägen aus {hoch, mitte, tief} und ebenso fünf Verteidigungen fest. Nahm der Herausgeforderte an, tat er das selbe. Den Kampf konnten sich die Kontrahenten dann unabhängig voneinander und ohne weitere Interaktion im Browser anschauen.

In [BS04, Kapitel 11] wird eine C++ Implementierung für einen Computergegner in einem einfachen Beat-'em-up-Spiel vorgestellt. In diesem Spiel gibt es die drei Angriffsvarianten Schlag, hoher Tritt und tiefer Tritt. Solch ein einfaches Spielprinzip hat es tatsächlich gegeben, wie man in Abb. 44 sieht. Die KI soll versuchen, auf Basis der letzten beiden Schläge des Spielers den nächsten vorauszusagen.

In die Regelbasis werden alle kombinatorisch möglichen Regeln der Form **if** Schlag1 **and** Schlag2 **then** Schlag3 aufgenommen. Daher ist die Regelbasis offensichtlich inkonsistent. Allerdings ist ohnehin keine Verkettung von Regeln möglich, da keine Regel auf der Konklusion einer anderen aufbaut. Zur Konfliktresolution erhält jede Regel ein Gewicht, nach dem ausgewählt wird. Bei der Initialisierung werden alle 27 möglichen Regeln gebildet und alle Variablen auf initiale Werte gesetzt.

Wenn der Spieler erst weniger als zwei Schläge gemacht hat, ist noch keine Vorhersage möglich. Bevor eine Vorhersage gemacht wird, wird zuerst die letzte Vorhersage ausgewertet. War sie richtig, wird das Gewicht der vorhergesagten Regel erhöht. Sonst wird ihr Gewicht vermindert und das Gewicht der angewendeten Regel erhöht. Dann wird das Faktenwissen mit dem neuesten Schlag aktualisiert.

Zuerst werden alle passenden Regeln markiert. Dann wird aus den passenden Regeln diejenige mit dem höchsten Gewicht ausgewählt und ausgeführt. Nebenbei wird auch noch

eine zufallsbasierte Vorhersage gemacht, um den Erfolg des Verfahrens zu messen.

Der Autor gibt in seinem Buch an, mit dem Regelsystem mindestens 65 % richtig vorhergesagte Schläge zu erreichen, während die zufallsbasierte Vorhersage auf ein Drittel kommt. In dem Buch wird dieselbe Aufgabe außerdem noch mit einem Bayes'schen Netzwerk gelöst, das aber auch nur dieselbe Erfolgsrate erreicht.

E Learning Classifier Systems

E.1 Einleitung

Seitdem der Computer nicht nur als Arbeitsutensil sondern auch für die Freizeitgestaltung genutzt wird, versuchen Spieleentwickler jedes Spiel, insbesondere die Intelligenz der sogenannte NPC's (non player characters) zu verbessern und somit weniger berechnend zu gestalten. Jeder Computerspieler kennt die Situation: Ist die richtige Strategie zum besiegen eines Gegners gefunden, so funktioniert diese immer wieder. Dies führt natürlich zu einem extremen Motivationsverlust, so dass das betroffene Spiel schnell uninteressant wird. Dem Computergegner soll also ermöglicht werden, aus seinen Aktionen zu lernen und sein Verhalten der Umwelt anzupassen.

Ein mögliches Konzept zur Verbesserung dieses Verhaltens bieten die „Learning Classifier Systems“ (LCS) [BK05] welche 1976 durch J.H. Holland das erste mal vorgestellt wurden. In [Wil94] stellt Steward W. Wilson auf der Basis dieses klassischen LCS das Zeroth Level Classifier System (ZCS) vor , welches den klassischen Ansatz von Holland vereinfacht, um so eine Leistungsteigerung zu erreichen. Das XCS (eXtended Classifier System) von Tim Kovacs [BK05] stellt einen anderen Ansatz zur Lösung bekannter Probleme der Classifier Systemen (CS) dar.

Ziel dieser Arbeit ist es, die Grundlagen von LCS darzustellen. Was sind die Grundgedanken des Konzeptes und wie sehen die Weiterentwicklungen des CS aus? Hierzu wird in Kapitel 2 auf die Struktur des „Learning Classifier Systems“, dass heißt auf die einzelnen Komponenten „if-then-rules“, „reinforcement learning“, „genetic algorithm“ und deren Zusammenspiel eingegangen.

Das dritte Kapitel befasst sich mit dem klassischen LCS nach Holland [BK05] und der Struktur des ZCS nach [Wil94].

In Kapitel 4 wird der Bezug zum eigentlichen Projektgruppenthema „Methoden der Computational Intelligence zur Entwicklung von Spielstrategien“ gesucht. Wo bieten sich Einsatzmöglichkeiten der LCS, was wurde bereits realisiert und welche Probleme könnten beim Einsatz von LCS auftreten?

E.2 Grundlagen eines LCS

E.2.1 Was ist ein „Learning Classifier Sytem“

Was ist eine Learning Classifier System? Wie in [HBC+99] ersichtlich wird, sind sich die führenden Forscher im Gebiet der LCS bei der Beantwortung dieser Frage nicht ganz einig. Nach John H. Holland in [HBC+99] ist ein LCS ein System welches evolutionäre Algorithmen (EA) verwendet, um in einem Regelbasierten Bedingung/Aktion System zu lernen. Nach Larry Bull und Tim Kovacs in [BK05] ist ein LCS eine Technik des Maschinellen Lernens, welches evolutionäre Algorithmen, reinforcement learning, überwachtes Lernen, unüberwachtes Lernen und heuristische Mittel kombiniert, um ein adaptives System zu erzeugen. [BK05] Stewart W. Wilson [HBC+99] beschreibt ein CS als ein lernendes System, welches auf evolutionären Prinzipien beruht und aus zwei Teilen besteht:

- einer Sammlung von „if-then-Regeln“, den sogenannten Classifiern
- einem Algorithmus, der die Classifier anwendet, auswertet und verbessert

Wie sich zeigt, stützen sich alle Antworten auf ein Regelbasiertes System, welches mittels evolutionären Algorithmen einen Lernerfolg erzielen soll.

Ein LCS ist ein mit der Umwelt agierendes System, das lernen soll, ohne genaue Informationen zu haben, wie es lernen soll. Die Verbindung zur Umwelt[15] (siehe Abbildung: 45) geschieht im LCS über Effektoren und Detektoren. Die Detektoren[16] liefern dem LCS die Daten der Umwelt in Form von Nachrichten[17] (messages), die in die Nachrichten Liste (Message list) eingetragen werden. Als Detektoren kann man sich zum Beispiel optische Sensoren, Temperatursensoren vorstellen. Welche Aktionen das System durchführen kann, wird durch die Regeln (Classifier) festgelegt. Jeder Classifier besteht besteht aus einem Bedingungs- und einem Aktionsteil. Die Bedingungen legen fest, welche Nachrichten zum auslösen der Aktion führen. Die Aktion kann entweder eine neue Nachricht in die Nachrichten Liste schreiben oder die Effektoren ansprechen. Die Effektoren können zum Beispiel Arme eines Roboters sein.

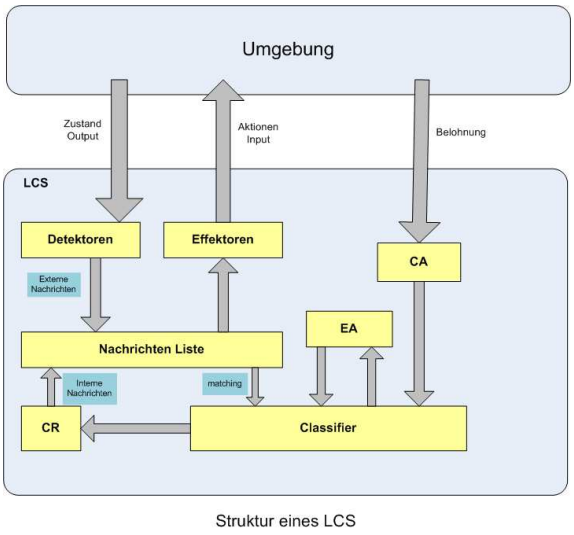


Abbildung 45: Struktur eines LCS

In [BFM00a] erfolgt die Ausführung eines LCS durch Wiederholung der folgenden Schritte:

- Detektoren tragen die Nachrichten der Umgebung die Nachrichten Liste ein.
- Jede Bedingung jedes Classifiers wird mit allen Nachrichten auf der Liste verglichen, um passende Classifier zu finden.
- Die Classifier, bei denen die Bedingung erfüllt ist, nehmen an einem Wettbewerb statt, indem sie sich für die Situation anbieten und ein Gebot abgeben.

[15] Die Umwelt kann durch Zustände beschrieben werden
[16] kodieren die Zustandsinformationen der Umgebung
[17] Binäre Nachrichten

- Eine Methode zur Konfliktlösung (CR) begrenzt die Liste der Regeln (Classifier) um widersprüchliche Aktionen zu entfernen.
- Die Nachrichten Liste wird gelöscht.
- Die Classifier, die den Wettbewerb[18] gewonnen haben, senden ihre Nachricht ab.
- Effektoren lesen die Nachrichten in der Nachrichten Liste und führen entsprechende Aktionen in der Umgebung durch.
- Falls eine Belohnung durch die Umgebung erzeugt wird, werden die Classifier durch eine credit allocation (CA) belohnt.

Die Bewertung und Belohnung erfolgreicher Classifier wird (siehe Abbildung 45) durch einen Bewertungsalgorithmus (bucket brigade) durchgeführt. Eine genauere Beschreibung des „bucket brigade“ Algorithmus folgt in Kapitel 3.1. Durch einen EA werden erfolglose und somit nicht benutze Classifier durch neu generierte Classifier erzeugt.

Formal kann ein LCS nach [Boe94] als Tupel der Form $\langle C, B, F \rangle$ definiert werden, mit

- C einer Menge von Classifiern. Die Größe der Menge $|C|$ wird mit M bezeichnet.
- B der Menge von Nachrichten, der Nachrichten Liste
- F einer Funktion $F(C, B) \rightarrow (B', \Upsilon)$ die eine Nachrichten Liste und eine Menge von Classifiern in eine andere Nachrichten Liste B' und den Output Υ konvertiert.

E.2.2 Classifier

Die Regeln des Systems werden durch Classifier repräsentiert. Diese haben immer eine feste IF-THEN Struktur. Einfache Beispiele von Classifiern sind:

- IF Player far away THEN use Sniper Rifle
- IF Player in small range THEN use Knife
- IF 011*1 THEN 01
- IF 011*1|1001* THEN 11

Bedingungen und Aktionen der Classifier können also als Bit-String kodiert werden, so dass im folgenden immer von Bit-Strings ausgegangen wird. Die Bedingungen können aus $\{0, 1, *(Wildcards)\}$ bestehen, wobei nun zwei verschiedene Fälle unterschieden werden können:

1. unmatched true: Falls keine Nachricht aus der Nachrichten Liste die Bedingung eines Classifiers erfüllt.

[18] Der Wettbewerb basiert auf einer Bewertung (Stärke), welche den bisherigen Nutzen des Classifiers darstellt

2. matched true: Eine Bedingung wird aktiviert, wenn sie durch eine Nachricht aus der Nachrichten Liste erfüllt wird.

Der Classifier kann zwei verschiedene Aktionen durchführen:

1. Ein Teil, der als Nachricht in die Nachrichten Liste geschrieben werden kann[19]. Dieser besteht aus $\{0, 1, \#(paththrough)\}$. Wird eine Nachricht in die Nachrichten Liste geschrieben, so werden die „paththroughs“ mit den korrespondierenden Bits aus einer Nachricht aufgefüllt, welche die Regel aktiviert hat. Zum besseren Verständnis wird ein Classifier mit der Bedingung $10 * 1001 * *1$ und der Aktion 1010#10#1 von der Nachricht 1011001001 aktiviert. Es entsteht durch ersetzten der korrespondierenden Bits die Nachricht 101001001.
2. Ein Teil, der die Umgebung direkt durch einen Output beeinflussen kann.

Mit jedem Classifier wird ein skalarer Fitnesswert verbunden, der die Nützlichkeit des Classifiers beschreibt eine externe Belohnung zu erhalten.[Boe94]

Ein Classifier kann als $\langle \Gamma, A, \Upsilon, S \rangle$ Tupel definiert werden, wobei

- Γ eine Menge von Tupeln der Form $\langle g, C \rangle$ mit $g \in \{false, true\}$ dem boolschen Teil der Bedingung, und C dem String Teil der Bedingung $\{0, 1, *\}^l (l \in \mathbb{N}^+)$. Ist der boolsche Teil der Bedingung *true* so wird die Bedingung als matched true bezeichnet. Im negativem Fall gilt die Bedingung als unmatched true.
- A ist ein String der Form $\langle 0, 1, \# \rangle$ der Aktion.
- Υ ist ein String der Form $\{0, 1\}^l (l \in \mathbb{N}^+)$ oder einem leeren String ϵ, dem Output.
- $S \in \mathbb{R}$ der Stärke.

E.2.3 Nachrichten Liste

Um zu ermöglichen, das mehrere Classifier gleichzeitig aktiviert werden können, besitzen Classifier Systems die Nachrichten Liste. Diese ist, wie der Name schon sagt, eine Liste von Nachrichten, die entweder durch die Umwelt erzeugt wurden oder Ergebnisse von Classifiern darstellen.

Sobald die Bedingung eines Classifiers durch eine Nachricht erfüllt wird, wird dieser aktiviert. Ob nun ein aktivierter Classifier seine Ausgabe in eine neue Nachrichten Liste schreiben darf, hängt davon ab, welche „Stärke“ dieser besitzt. „Stärkere“ Classifier haben eine höhere Chance ihre Ausgabe zu erzeugen als „Schwächere“. Nachdem die Nachrichten Liste vollständig abgearbeitet ist, wird sie verworfen und die neu erzeugte Liste tritt an ihre Stelle. So entsteht ein Kreislauf, der durch Nachrichten, die durch die Umwelt generiert wurden, startet. Dieser Kreislauf endet, sobald ein Classifier eine Ausgabe erzeugt, die die Umwelt beeinflusst.[Boe94]

[19] sogenannte interne Nachrichten

E.2.4 Evolutionäre Algorithmen

In LCS versucht der EA aus der Menge der bereits vorhanden Classifier, die „starken“ beizubehalten, die „schwachen“ zu eleminieren, neue „stärkere“ Classifier zu erzeugen und zu überprüfen, in welchem Maße die neuen Classifier den Anforderungen gerecht werden. Für genauere Informationen zum Thema „Evolutionäre Algorithmen“ siehe [BFM00a], [BFM00b]. Für den Einsatz von EA in LCS haben sich zwei Methoden durchgesetzt.

Pittsburgh-Methode Bei der Pittsburgh-Methode wird jedes Individuum durch eine Menge von Classifiern repräsentiert. In diesem Ansatz wird die Anzahl der einzelnen Regeln jedes Individuums der Population durch Kreuzung oder andere Operatoren verändert. Dies hat den Vorteil, dass eine komplette Lösung mit jedem Individuum des EA erreicht werden kann. Der EA kann zu einer homogenen Population konvergieren, welche durch das beste Individuum, dem Individuum mit dem höchsten Fitnesswert, repräsentiert wird. Das gefundene Individuum, die Menge von Classifiern, bilden dann die Lösung des Problems. Der Nachteil dieser Methode liegt in der Auswertung jedes Individuums. Die Auswertung verursacht einen hohen Aufwand, so dass das Lernverhalten erschwert wird.

Michigan-Ansatz Im Michigan-Ansatz steht jeder Classifier für ein Individuum, so dass nur eine Population ausgewertet werden muss. Somit kann der EA nicht zu einer homogenen Population konvergieren, da eine Regel keine gültige Lösung für das Problem darstellen kann.

E.2.5 Reinforcement Learning

Learning Classifier Systems sind in der Lage mit Hilfe von Detektoren und Effektoren ihre Umwelt zu überwachen und sie zu beeinflussen. Auf der einen Seite wird die Umgebung durch das LCS beeinflusst, auf der anderen Seite hängt die Auswahl der Aktionen des LCS von seiner Umgebung ab. 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[Boe94].

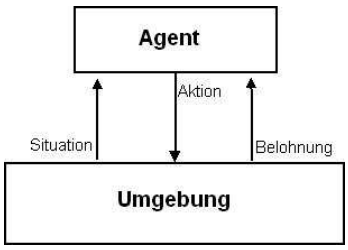


Abbildung 46: Reinforcement learning

Der Agent kann mit seiner Umgebung interagieren. Über Input-Schnittstellen (in LCS Detektoren) wird dem Agenten die vorherrschende Situation der Umwelt mitgeteilt. Aufgrund dieser Informationen wählt er Aktionen aus, die sich auf die Umwelt (in LDS über

die Effektoren) auswirken. Anschließend wird der Agent für seine durchgeführten Aktionen belohnt oder bestraft. Für genauere Informationen zum Thema „Reinforcement Learning“ siehe [SB98].

E.2.6 Classifier System am Beispiel des Animaten

Es gibt mehrere Interpretationen von Hollands Classifier Systems. Die Erklärung eines Classifier Systems mit Hilfe des Animaten (animal + robot = Animat) bietet jedoch eine sehr intuitive und anschauliche Beschreibung, wie ein CS arbeitet, so dass diese für das erste Verstehen sehr gut geeignet ist siehe [HB99].

Es wird von folgender Annahme ausgegangen: Der Animat befindet sich in einer virtuellen Umgebung und kann mittels Sensoren (Detektoren), Greifarmen und Motoren (Effektoren) mit dieser interagieren. Detektoren und Effektoren sollen nun dem Animaten ermöglichen, sich in der Umgebung zurechtzufinden.

Also benötigt der Animat folgende Komponenten:

- Die Umwelt (Environment)
- Die Sensoren (Detektoren) die dem System mitteilen, wie sich die Umgebung verhält.
- Die Effektoren, die dem Animaten die Möglichkeit bieten, die Umgebung zu verändern.
- Das Classifier System (Logik)

Der Animat befindet sich in einer digitalen Welt und muss sich der Aufgabe stellen, in dieser Welt gut ernährt zu überleben. Somit muss er lernen, zwischen „Genießbarem“ und „Ungenießbarem“ zu unterscheiden.

Wir nehmen also an, dass unser Animat, in diesem Fall ein Frosch namens Kermit, in einem kleinen Teich (seiner Umgebung) lebt. Er ist mit Sensoren (Augen) und Effektoren (Armen, Beinen und einem Maul zur Nahrungsaufnahme) ausgestattet. Kermits Logik, kann man sich am einfachsten als Computer vorstellen, welcher Eingaben erhält, diese intern durch Programme auswerten läßt und eine Ausgabe zurückliefert.

Als Classifier bezeichnen wir IF-THEN Regeln, bei denen die Bedingungen und Aktionen als Bit-String codiert sind. Eine Menge aus Classifiern wird Population genannt. Neue Classifier, also Regeln, können durch Manipulation der Classifierpopulation mit dem Einsatz EA erzeugt werden.

Aufgrund der gegeben Informationen ergibt sich folgendes Szenario.

Die Sensoren beobachten die Umwelt und erzeugen Nachrichten, welche in die Nachrichten Liste geschrieben werden. Nun werden diese erzeugten Nachrichten mit dem Bedingungsteil der IF-THEN Regeln verglichen. Anschließend wird die Nachrichten Liste gelöscht und die durch die Classifier ausgelösten Aktionen werden wieder in die Nachrichten Liste geschrieben. Dies tritt genau dann ein, wenn seine Bedingung erfüllt wurde. Nun überprüft das Interface der Effektoren, ob in der Nachrichten Liste Nachrichten enthalten sind, die die Effektoren ansprechen. Somit ist ein kompletter Zyklus abgearbeitet und der Kreislauf

kann von neuem beginnen. Da nach dem Löschen der Nachrichten Liste wieder neue Einträge durch die Classifier vorgenommen werden, ist es möglich, dass die Nachrichten Liste zu Beginn eines Zyklus nicht leer ist.

Die Idee eines Classifier Systems ist es, mit einer zufällig gewählten Classifierpopulation, also ohne jeglicher Kenntniss zu starten und mit der Zeit zu lernen, das Verhalten des Systems anzupassen.

Kommen wir zurück zu Kermit. Er lebt in einem (digitalen) Teich und bewegt sich zufällig in seiner Umgebung. Immer wenn er ein kleines fliegendes Objekt ohne Streifen sieht, soll Kermit es essen. Falls das fliegende Objekt Streifen besitzt, sollte Kermit auf den Verzehr besser verzichten. Entdeckt er ein großes nichtidentifizierbares Objekt, soll er so schnell wie möglich wegspringen. Aus diesen Verhaltensregeln lassen sich folgende Classifier herleiten[HB99]:

1. IF kleines, fliegendes Objekt zur Linken THEN sende @
2. IF kleines, fliegendes Objekt zur Rechten THEN sende %
3. IF kleines, fliegendes Objekt mittig THEN sende $
4. IF großes, nicht identifizierbares Objekt THEN sende !
5. IF kein großes, nicht identifizierbares Objekt THEN sende *
6. IF * und @ THEN bewege den Kopf 15° nach links
7. IF * und % THEN bewege den Kopf 15° nach rechts
8. IF ! THEN springe schnell weg

Die binäre Kodierung der oben dargestellten Regeln hat folgende Form:

IF	Größe	Art	Richtung	THEN
1	0000	00	00	0000
2	0000	00	01	0001
3	0000	00	10	0010
4	1111	01	**	1111
5	¬1111	01	**	1000
6	1000	00	00	0100
7	1000	00	01	0101
8	1111	**	**	0111

Die Kodierung ist folgendermaßen zu verstehen: „0000“ steht für klein, „00“ im ersten Teil der zweiten Spalte bedeutet fliegend, während der zweite Teil die Richtung vorgibt. Die Aktionen werden durch die letzten vier Bits repräsentiert. Die Aktionen @,%, !, können als interne Nachrichten verstanden werden. Die eigentlichen Aktionen mit der Umwelt gehen dann aus den anderen Classifiern hervor.

E.3 LCS nach Holland und ZCS

Nach dem gerade dargestellten intuitiven Ansatz zur Erklärung eines Classifier Systems werden wir nun einen genaueren Blick auf Hollands Entwicklung des LCS [WG89] und anschließend auf eine mögliche Weiterentwicklung zum ZCS nach Wilson, welche in [Wil94] vorgestellt wird, werfen.

Hollands Classifier System One simulierte ein lineares Labyrinth. Wenn der Animat den Ausgang des Labyrinths erreicht hatte, wurde das gesamte System belohnt, so dass zuerst an jeder Kreuzung der Schritt in die richtige Richtung gelernt werden musste, damit der Animat den Ausgang findet.[HB99]

Dieses Prinzip des infrequent payoffs[20] erwies sich später als problematisch. Man stelle sich vor, dass eine große Menge von Classifiern am Ende eines Durchlaufs belohnt werden soll. Lange Laufzeiten bei relativ einfachen Aufgaben lassen sich nicht vermeiden oder die Belohnung ist sehr aufwendig. Durch verschiedene Ansätze wird versucht, die Entlohnung des Systems, also den gesamten Lernprozess, zu verbessern. Ein weiteres Problem stellte die richtigen Auswahl der Regeln für bestimmten Situationen dar. Ein Ansatz zur Bewältigung dieses Problems bietet die Zusammenlegung solcher Regeln, die sich in ähnlichen Situation auch ähnlich verhalten. Jedoch muss das CS schon einige Zyklen durchlaufen haben, um solch einen Zusammenhang zu erkennen. Durch diesen Ansatz konnte eine Leistungssteigerung beobachtet werden. Wie wir in unserem oben dargestellten Beispiel eines Frosches sehen, können durch die Verwendung von Wildcards (#), mehrere Regeln in einer bestimmten Situation Anwendung finden. Normalerweise würde die „stärkere“ Regel gewählt und entsprechend entlohnt. Jedoch ist dies nicht immer wünschenswert, da die zweite Regel eventuell ein besseres Ergebnis erzeugen würde. Es kristallisieren sich prinzipiell zwei grundlegende Herausforderungen an die CS heraus:

1. Wie findet eine sinnvolle Entlohnung der durchgeführten Aktionen statt?
2. Wie wird eine sinnvolle Auswahl eines Classifiers in einer bestimmten Situation durchgeführt?

Für diese beiden zentralen Probleme haben sich zwei Verfahren durchgesetzt, die im folgeneden genauer diskutiert werden sollen. Das Konzept der „bucket brigade“(Kapitel 3.1) behandelt die Frage einer sinnvollen Entlohnung, während sich die „Bidding and Payment“-Strategie (Kapitel 3.2) mit der Auswahl eines Classifiers befasst.

E.3.1 Bucket Brigade

Das Konzept „bucket brigade“ befasst sich mit der Entlohnung einzelner Classifier. Das bedeutet, dass jeder Classifier eine Belohnung erhält, sofern er an einer erfolgreichen Aktion beteiligt war. Hier erkennt man direkt das eigentliche Problem des Prinzips. Es darf nicht nur der Classifier belohnt werden, der letztendlich die Aktion ausgeführt hat, sondern auch dessen Vorgänger, welche ja maßgeblich am Erfolg beteiligt gewesen sein können. Sie haben sozusagen die Vorarbeit geleistet und dürfen daher nicht vernachlässigt werden.

[20] unregelmäßige und seltene Belohnung

Die Belohnung mehrerer Classifier wird durch eine „bucket brigade“, eine Art Eimerkette, realisiert. Nach [WG89] bedeutet dies, dass der Classifier, der zu einer Entlohnung führt, sich die Belohnung mit bis zu vier oder fünf Vorgängern teilen muss, da diese entscheidend am Erfolg beteiligt waren. Somit erhält jeder relevante Classifier einen Stärkezuwachs, womit die Wahrscheinlichkeit steigt, dass einer dieser Classifier beim nächsten Zyklus ausgewählt wird. Es wird vorausgesetzt, dass lange Regelketten verwaltet werden müssen. Dies umfasst sowohl die Erzeugung solcher Regelketten als auch das Aufrecherhalten selbiger.

Das Aufrechterhalten der Regelketten ist von besonderer Bedeutung, da viele Systeme erst mehrere Ausgaben erzeugen können, bevor die Belohnung durch ihre Umgebung erfolgt. Jedoch wird diese Aufrechterhaltung durch den Konkurrenzkampf und die Kooperation der Classifier in einer Kette wesentlich erschwert[WG89].

Classifier in einer Kette sind kooperativ, da die „frühen“ Classifier von der Belohnung der „späteren“ Classifier profitieren, hingegen können die „späteren“ Classifier nur aktiviert werden, falls die richtige Situation erzeugt wurde. Die Classifier, die sich am Ende der Kette befinden, werden stärker belohnt als ihre Vorgänger. Diese erhalten, je weiter sie zurückliegen, einen immer geringeren Anteil der Belohnung des letzten Classifiers. Somit steigt die Wahrscheinlichkeit für die Classifier am Ende der Kette an, im nächsten Zyklus ausgewählt zu werden. Auf die Classifier am Anfang der Kette steigt der Eleminationsdruck[21], welcher nicht unbedingt erreicht werden möchte, da sie wesentlich für den Erfolg mitverantwortlich, wenn nicht sogar entscheidend waren. Lange Regelketten können also sehr leicht „zerbrechen“, da die Gefahr einer Abspaltung „schwächerer“ Classifier ständig wächst. Des weiteren tendieren Classifier, die sich am Ende einer Regelkette befinden dazu, häufiger belohnt zu werden, da mehrere Sequenzen mit ihnen enden.

Ein anderes Belohnungsprinzip wird in [Boe94] vorgestellt. Classifier werden nicht nur belohnt, sondern zahlen abhängig von ihrem Einsatz auch eine Gebühr. Ein normaler Ablauf der Belohnung wird im folgenden dargestellt:

- alle Classifier, die in Zeitpunkt t-1 nicht aktiv waren: $S'_{c,t} = S_{c,t-1} - \tau_{life} * S_{c,t-1}$ mit $0 \leq \tau_{life} < 1$
- alle Classifier, die in Zeitpunkt t-1 am Wettkampf teilgenommen haben: $S'_{c,t} = S_{c,t-1} - (\tau_{bid} * \tau_{life}) * S_{c,t-1}$ mit $0 \leq \tau_{bid} < 1$
- alle Classifier, die in Zeitpunkt t geboten und aktiviert wurden: $S''_{c,t} = S'_{c,t} - B_c$
- Belohnen der Classifier, die geholfen haben Classifier c zu aktivieren:$S_{x,t} = S'_{x,t} + \frac{1}{|Q_{c,t}|} * B_c$
- Belohnung des Classifiers, der die Belohnung ausgelöst hat: $S_{c,t} = S'_{c,t} + r_t$

Das Hinzufügen sogenannter Brückenregeln („bridge classifier“) versucht diesem Problem entgegen zu wirken. Die Belohnung der weiter zurückliegenden Classifier kann somit in einem Zyklus durchgeführt werden. Jedoch benötigt das System neue Operatoren, um das Prinzip durchzuführen. Eine weitere Möglichkeit besteht in der Hierarchiebildung von bucket brigade Ketten. Die Ketten an sich werden relativ kurz gehalten und durch eine

[21] Classifier geringer Stärke haben eine größere Wahrscheinlichkeit aus der Classifiermenge geworfen zu werden

modulare Vorgehensweise können komplexe Handlungsabläufe erzeugt werden[WG89]: Für genauere Informationen zum Thema „bridge classifier“ siehe [Rio87].

Wie wird dem Problem der Erzeugung langer Regelketten begegnet?

In der Theorie werden neue Regeln nur durch den EA erzeugt. Bei der Verwendung des „bucket brigade“ Konzeptes liegt es nahe, die Stärke der einzelnen Classifier als Fitnesswerte für den EA zu verwenden. Um herauszufinden, welche Classifier besonders gut und welche schlechter sind, durchläuft das CS mehrere Zyklen des „bucket brigade“ Algorithmus, bis sich die Stärkewerte stabilisiert haben. Nun werden durch den EA neue Classifier durch Mutation, Kombination oder Kreuzung erzeugt und für die nächsten Zyklen verwendet.

Tatsächlich werden aber zur Erzeugung neuer Regeln auch andere Methoden herangezogen, mit dem Ziel, den Lerneffekt des CS zu beschleunigen. Ein CS soll wenn möglich viele verschiedene Probleme lösen, wie z.B. das Finden von Nahrung in unterschiedlichen Umgebungen. Es wird schnell verständlich, dass durch den Einsatz genetischer Algorithmen nicht immer eine hinreichend gute Lösung für eine bestimmte Situation und Umwelt gefunden werden kann. So macht der alleinige Einsatz dieser nur dann wirklich Sinn, wenn die zu lösenden Aufgaben ähnlich sind. Ein zusätzlicher Faktor, der die Bildung langer Regelketten unterstützt, wird „Coverings and Couplings“ genannt. Erhöht sich die Stärke eines Classifiers zu einem Zeitpunkt extrem, obwohl die Situation sich aus Sicht des Classifiers nicht verändert hat, könnte dieser starke Zuwachs an einer vorherigen Bedingung liegen. Es wird also in der Regelkette zurückgegangen, um den Classifier zu finden, der für die hohe Belohnung verantwortlich war. Anschließend werden durch Neukombinationen mit der gefunden Regel neue Regeln in der Kette eingebaut. [WG89]

Mögliche Lösungen bestehen in der Vermeidung langer Regelketten. Es sollen relativ kleine Regelketten zu Modulen zusammengefasst werden, welche als eigenständige Regeln behandelt werden können. Welche Regeln zusammenpassen zeigt sich meist nach einigen Durchläufen des LCS. Einige Classifier haben z.B. immer die gleichen Classifier zur Folge, so dass diese zusammengefasst werden können.

Zum Beispiel könnte sich der Classifier „in die Oper gehen“ aus der Regelkette „anziehen“, „Taxi rufen“, „das Opernhaus betreten“ ableiten. Der Classifier „anziehen“ besteht wiederum aus „„Socken anziehen“, „Schuhe zubinden“ und „Mantel anziehen“.

E.3.2 Bidding and Payment

Dieser Abschnitt behandelt das Problem der Auswahl eines Classifiers in einer bestimmten Situation. Hierzu stellt jeder Classifier ein Angebot, welches beeinflusst, ob der Classifier in der vorliegenden Situation ausgewählt wird oder nicht. Auf Basis der Stärke eines Classifiers wird sein Gebot bestimmt. In [Boe94] berechnet sich das Gebot eines Classifiers durch:

$$B_c = \beta * f_{spec}(c) * S_c \ , \ 0 \leq \beta < 1$$

$$f_{spec}(c) = \frac{L - W}{L} * \alpha \ , \ 0 \leq \alpha < 1$$

L = Anzahl aller Zeichen in der Bedingung des Classifiers.
W = Anzahl der Wildcards in der Bedingung des Classifiers.
Die Entlohnung erhält er, nachdem er in einem Zyklus erfolgreich angewand wurde.[WG89]
Es wird schnell ersichtlich, dass das Angebot eines Classifiers durch die vorherigen Entlohnungen beschränkt wird. Wichtig ist jedoch auch, nicht von Anfang an nur die Classifier auszuwählen, die das höchste Gebot stellen. Es kann sinnvoll sein, erst alle Möglichkeiten zu erkunden. Dieses Problem bezeichnet man als „exploration vs. exploitation“. Es könnte für das System von Vorteil sein zu Beginn Regeln auszuwählen, die auf den ersten Blick nicht die besten Ergebnisse erzielen. Dieses ändert sich mit zunehmender Anzahl an Zyklen, so dass im späteren Verlauf die „stärkeren“ Classifier bevorzugt werden. Somit kann der Erfolg des Systems gesteigert werden. Die Wahrscheinlichkeit steigt, dass sich im Laufe der Zeit eine „default hierarchy“ bildet. Hierbei sind die Classifier hierarchisch strukturiert, wobei allgemeinere Classifier meist über den speziellen Classifiern angeordnet sind, da allgemeine Regeln eine höhere Wahrscheinlichkeit besitzen, ausgewählt zu werden. So können möglichst gute Ergebnisse in relativ kurzer Zeit erzielt werden. Um eine Kontrolle über die Auswahl der Classifier zu erhalten, eignen sich folgende Methoden:

- Roulette-Methode: Die Summe der sich anbietenden Classifier wird durch die Gebote geteilt. Classifier mit einem höheren Gebot werden bevorzugt.
- noisy auction: Die stärkeren Classifier werden zwar immer noch bevorzugt ausgewählt, jedoch besteht auch die Möglichkeit, dass sich schwächere Classifier durchsetzten. Durch Justierung des Geräuschlevels bei der Auktion können hohe Angebote der Classifier „überhört“ werden, so dass auch Classifier mit einem niedrigeren Gebot die Chance haben, ausgewählt zu werden. Durch Anpassung des Geräuschlevels kann man nun eine komplett randomisierte Auswahl oder eine deterministische Auswahl erzeugen.

Nach [Boe94] muss jedoch festgehalten werden, dass keine der beiden vorgestellten Methoden für jede Situation geeignet ist. Welche Methode ausgewählt wird hängt also von der jeweiligen Situation ab.

Classifier Systeme neigen bei der Angebots- und Entlohnungsarchitektur zur Bildung übergeneralisierter Regeln. Es ist leicht nachvollziehbar, dass übergeneralisierte Regeln häufiger ausgewählt werden als spezielle Regeln, da diese zu sehr vielen Situationen passen. Durch Rekombination dieser generalisierten Regeln kann diesem Phänomen nur in begrenztem Maße entgegengewirkt werden, da bei einer Rekombination die Generalisierung nicht vollständig aufgehoben wird. Ein gewisses Maß an Generalität ist sogar von Vorteil, um in relativ kurzer Zeit eine „default hierarchy“ zu bilden. Jedoch darf diese Generalisierung das Ergebnis nicht verfälschen oder zu ungenau werden lassen. Generalisierte Regeln sollten also einer gesonderten Beobachtung unterliegen und gegebenenfalls eine geringere Belohnung erhalten oder sogar bestraft werden, damit sich diese nicht zu stark ausbreiten. Generalisierung darf jedoch nicht komplett verhindert werden, da gerade zu Beginn so schnell eine „default hierarchy“ entsteht. Wir kommen also zu dem Entschluss, Generalisierung zu Beginn des System zuzulassen, also die Auswahl von allgemeinen Classifiern nicht zu unterbinden. Im späteren Verlauf sind Generalisierungen eher hinderlich, da nach einiger Zeit das Ergebnis durch die Wahl allgemeiner Classifier nicht mehr verbessert werden kann [Boe94].

E.3.3 ZCS nach Steward W. Wilson

Auf Basis des von Holland entwickelten LCS wurde von Steward W. Wilson das Zeroth Level Classifier System (ZCS) [Wil94] und von Tim Kovacs das eXtended Classifier System (XCS) entwickelt [Kov96]. Die beiden Ansätze sind sich relativ ähnlich und versuchen, die oben genannten allgemeinen Probleme der LCS zu entschärfen. Wilson setzt bei dem ZCS auf die bewusste Eleminierung von Elementen, welche nicht unbedingt für den Erfolg des Systems benötigt werden, um so eine Leistungsteigerung zu erzielen. Das XCS von Kovacs versucht die Regelsätze möglichst klein zu halten, um somit den Lernerfolg zu verbessern. Zusätzlich führt Kovacs in [Kov96] die Menge der optimalen Classifierpopulation ein, welche folgende Eigenschaften erfüllt:

- Sie ist vollständig, d.h., alle möglichen Eingabe- und Aktionsräume sind durch Regeln vollständig abgedeckt.
- Es gibt keine Überschneidungen. Es gibt keine zwei Regeln, die auf dieselbe Situation Anwendung finden.
- Die Population ist minimal, d.h. die kleinstmögliche Anzahl sich nicht überschneidender Regeln beschreibt die Lösung zum Problem hinreichend. Um dieses Population zu erzeugen setzt Kovac auf die Methoden, die Kondensation und die subset extraction die jedoch hier nicht näher beschrieben werden. Mehr Informationen zum Thema XCS sind in [Kov96] zu finden.

Im folgenden wird lediglich das ZCS genauer beleuchtet, da eine zusätzliche ausführliche Erklärung des XCS den Umfang der Arbeit überschreiten würde.

Das ZCS erweitert Hollands Idee um ein temporäres Gedächtnis, wodurch schneller und besser auf sich ändernde Inputdaten reagiert werden kann, bei gleichzeitiger Beibehaltung des Lernerfolgs. Abbildung 3 stellt die Funktionsweise des ZCS graphisch dar.

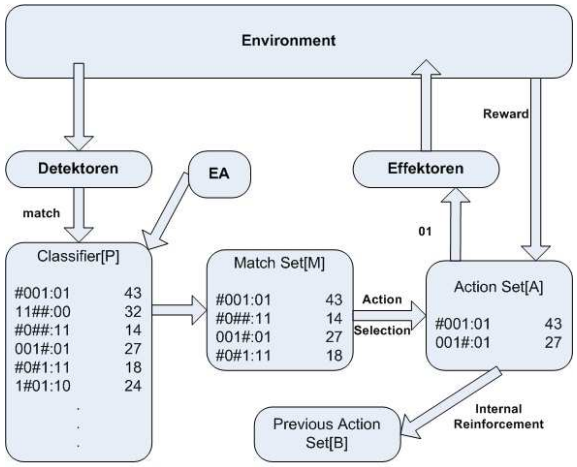


Abbildung 47: Struktur des ZCS

Das Prinzip der CS wird, wie Abbildung 3 zeigt, leicht abgeändert. Das System interagiert durch Detektoren und Effektoren mit der Umgebung. Die Umgebung belohnt das

System, wodurch ein Lerneffekt erzielt werden soll. Während eines Zyklus des ZCS wird jede Bedingung eines jeden Classifiers in der Menge der Classifier [P] mit dem Inputstring verglichen. In der Regel stimmen mehrere Classifier mit diesem überein und werden der derzeitigen Treffermenge [M] hinzugefügt. Stimmen keine Classifier mit dem Inputstring überein, so greift der EA ein, indem er einen passenden Classifier mit einer bestimmten Anzahl an Wildcards erzeugt. Die auszuführende Aktion wird randomisiert erzeugt und die Stärke des Clasifiers berechnet sich aus dem Durchschnitt aller Stärken in [P]. Diese Classifier werden anschließend dem Treffermenge [M] hinzugefügt. Mittels eines geeigneten Auswahlverfahrens (in diesem Fall meistens die Roulette-Methode) wird die Menge der Classifier, die ihre Aktion durchführen sollen(action set), [A] erzeugt und anschließend die auszuführenden Aktionen an die Effektoren weitergeleitet.[Wil94]

Die Belohnung betrifft im ZCS nur das action set [A] sowie das action set des vorherigen Zyklus $[A]_{-1}$. Das Belohnungsprinzip durchläuft folgende Schritte:

1. Jedem Classifier im action set [A] wird ein geringer Anteil $\beta * S_C, 0 < \beta \leq$ seiner Stärke abgezogen und in einen initial leeren „bucket“ gelegt.
2. Falls das System eine Belohnung durch die Umgebung (r_{imm})erhält wird sowohl die Stärke jedes sich in [A] befindlichen Classifiers um einen Anteil der Belohnung erhöht: $\beta * r_{imm}/|A|$.
3. Die Stärke der Classifier in $[A]_{-1}$ wird um einen Teil des im „bucket“ enthaltenen Wertes erhöht: $\gamma * B/|A_{-1}|, 0 < \gamma \leq 1$.
4. $[A]_{-1} := [A]$

Durch dieses Belohnungsprinzip entfällt der Konkurrenzkampf, der im klassischen „bucket brigade“ Ansatz beobachtet werden kann. Außerdem existieren keine Regelketten mehr, sondern die Belohnungen werden nur einen Schritt weiter zurückgereicht. Ein weiterer Unterschied liegt in der Bestrafung der nicht ausgwählten Classifier in [M]. Diesen wird ein Anteil ihrer Stärke abgezogen($\tau * S_C, \tau \cong \beta$), wodurch das System nach einigen Zyklen, die zur Erkundung der Möglichkeiten genutzt werden, „bessere“ Classifier auswählt, um den Erfolg zu verbessern. Die Generierung neuer Classifier übernimmt im ZCS ein gewöhnlicher EA. Sobald er aufgerufen wird, wählt er zunächst zwei Classifier aufgrund deren Stärke aus. Classifier mit einer hohen Stärke werden hierbei bevorzugt. Er kombiniert oder mutiert diese, worauf die zwei neu erzeugten Classifier wieder der Population hinzugefügt werden. Damit die Population nicht ständig wächst, werden zwei Classifier aus der Population mit einer Wahrscheinlichkeit, ihrer invertierten Stärke, entfernt. Die Wahrscheinlichkeit mit der der EA aufgerufen wird, wird im ZCS vom Benutzer ausgewählt. Diese muss so gewählt werden, dass dem System die Möglichkeit gegeben wird, gute Classifier hervorzuheben, bevor diese vom EA entfernt würden. Dies würde den Lernprozess erheblich behindern. Techniken, mit denen diese Wahrscheinlichkeit mit der der EA aufgerufen wird automatisch kontrolliert wird nennt L.B. Booker in [Boo82].

Zusammenfassend kann man sagen, dass durch ZCS im Vergleich zum Q-Learning in bestimmten Situationen sehr gute, wenn nicht sogar bessere Ergebnisse erzielt wurden. ZCS erzielten bei einigen bekannten Problemen eine nahezu optimale Lösung, jedoch sind die einzelnen verwendeten Parameter sehr empfindlich zu behandeln. [BK05]

E.4 Relevanz für die Projektgruppe PG511

Im Hinblick auf das Projektgruppenthema „CI in Games-Methoden der Computational Intelligence zur Entwicklung von Spielstrategien“ lassen sich LCS einsetzten, um den Nutzen von Aktionen[22] abzuschätzen. Somit wird der Vorteil bestimmter Aktionen hervorgehoben.

Die Classifier im System werden für ihre Präzision belohnt, wodurch das Wissen über die Welt, welche durch das System modelliert wird, sehr wirklichkeitsnah ist. Zusätzlich ist die Darstellung der Classifier aufgrund ihrer festen Struktur sehr gut zu verstehen. Jedoch haben sie den Nachteil der binären Kodierung. Die Inputdaten liegen natürlich nicht immer als Bit-String vor. Ein aufwendiges Konvertieren oder die Erweiterung der binären Darstellung scheinen auf den ersten Blick gute Lösungsansätze. Bei der Erweiterung der Darstellung gehen viele Generalisierungsvorteile aufgrund der Reduktion der Wildcards verloren. Es würde allerdings die Möglichkeit bestehen, das Wissen, welches die Entwickler über die Umwelt haben, bei der Modellierung mit einfließen zu lassen. Somit könnten bestimmte Reaktionen in günstigen Situationen berücksichtig werden.

In [SB98] wird ein anderer Ansatz der LCS gezeigt. Durch die explosive Entwicklung des Internets haben immer mehr Benutzer unterschiedlichster Anforderungen die Möglichkeit bekommen, Spiele im Internet zu nutzen. Ein statischer Algorithmus ist somit nicht mehr in der Lage, auf die verschiedenen Bedürfnisse der User zu reagieren. Dem Spiel soll ermöglicht werden, Entscheidungen in einem immer kleiner werdenden Zeitfenster zu treffen. Auf der Basis eines Online-Fussballspiels, welches mit einem „Event-driven Learning Classifier Systems“ Entscheidungen treffen soll, versuchen Sato und Kanno in [SB98]die gerade genannten Probleme zu lösen. Dieses Classifier System weicht in drei Punkten vom klassischen Ansatz ab.

1. Der „event analysis“ Abschnitt, der eine Tabelle mit den Häufigkeiten der vom Spieler verwendeten Aktionen aufzeichnet (z.B. schießen, dribbeln, passen).
2. Neue Classifier werden aufgrund der Häufigkeit der events durch einen EA erzeugt.
3. Die Stärken der Classifier werden mit dem bucket brigade Algorithmus berechnet. Es wird mit dem häufigsten verwendeten „event“ gestartet. Der Algorithmus endet, wenn keine Berechnung in „Echtzeit“ mehr durchgeführt werden kann.

In der experimentellen Auswertung wurde deutlich, dass dieser Ansatz den herkömmlichen Methoden überlegen war.

Zur allgemeinen Spieleentwicklung können LCS in vielen Situationen, sowohl bei Online- als auch bei Offline-Anwendungen eingesetzt werden, was sich durch ihre guten Lernerfolge begründen läßt. LCS lassen sich somit quasi für alle reinforcment learning Probleme verwenden, wobei die Schwierigkeit besteht, eine passende Belohnung oder Fitness zu finden.[Cha03]

Einige Bespiele für mögliche Einsatzgebiete wären die Waffenauswahl oder obstacle aviodance (Vermeidung von Hindernissen), obwohl hier in der Praxis meistens andere Verfahren bevorzugt werden. Dies lässt sich wohl durch den hohen Ressourcenverbrauch begründen, welcher sich durch die in der Arbeit beschriebenen Probleme ergibt (z.B. lange Regelketten).

[22] Bei PacMan zum Beispiel die Richtung, in die PacMan als nächstes gehen soll

Seit Hollands erstem Classifier System haben LCS eine ständige Entwicklung erfahren, wobei noch nicht alle Möglichkeiten ausgeschöpft werden konnten. Zukünftig müssen LCS bei einer Vielzahl von Problemen angewendet werden, um Charakteristiken zu identifizieren, die die Anwendung von LCS zulassen. Die Form eines adäquaten LCS für diverse Problemtypen muss ebenfalls ermittelt werden, zusammen mit kontinuierlicher Verfeinerung der Architekturen und verbessertem theoretischen Verständnis.[BK05] Abschließend kann somit festgehalten werden: „There's much to do in CFS research!"[HB99]

F Multi-Agenten Systeme

F.1 Einleitung

Thema dieser Seminararbeit sind *Multi-Agenten Systeme* (MAS). Das Forschungsgebiet ist jedoch zu komplex, um alle Aspekte ansatzweise beschreiben zu können. Stattdessen werde ich einen Überblick über die zentralen Eigenschaften und Mechanismen der MAS geben, besonders im Hinblick auf Kommunikation und Interaktion der einzelnen Agenten.

Kapitel F.2 beschäftigt sich daher zunächst mit dem Thema Softwareagenten und soll als Grundlage für die weitere Betrachtung dienen.

Kapitel F.3 wendet sich anschliessend dem Aufbau komplexer agentenbasierter Systeme, den MAS zu. Hier stehen, wie erwähnt, die Interaktionsmöglichkeiten innerhalb des Systems im Vordergrund.

Den Abschluss der Arbeit bildet Kapitel F.4 mit der kritischen Betrachtung der Anwendung von MAS in Computerspielen.

F.2 Softwareagenten

Softwareagenten besitzen eine Vielzahl möglicher Einsatzszenarien. Das Spektrum reicht von einfacher Informationsgewinnung bis hin zur Entscheidungshilfe bei der Lösung komplexer Probleme. Anwendungen im Bereich eCommerce, Simulation, Optimierung oder Computerspiele sind nur einige der Einsatzmöglichkeiten. Besonders im Zusammenhang mit Multi-Agenten Systemen heben sie sich von den üblichen, objektorienterten Paradigmen der Softwareentwicklung in Sachen Effizienz hervor.

An dieser Stelle soll zunächst der Begriff des Agenten bzw. Softwareagenten genauer definiert werden. Die Frage „Was ist ein Agent?“ oder „Was macht einen Agenten aus?“ mag leicht klingen, jedoch ist es schwierig eine zufriedenstellende Antwort zu geben.

Trotz des Fortschritts in Forschung und Entwicklung von Agenten wird man bei der Suche nach einer Definition nicht fündig werden. Bis heute existiert keine allgemein akzeptierte Definition des Agentenbegriffs. Dies liegt vor allem an der Flexibilität des Agenten selbst. Bis auf wenige Ausnahmen gibt es keine Einigung bezüglich grundlegender Charakteristiken.

Hier eine kleine Auswahl an bekannten Definitionen:

> An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors. [PN95]

> A software entity which functions continuously and autonomously in a particular environment, often inhabited by other agents and processes. [Sho97]

> Intelligent agents are software entities that carry out some set operations on behalf of a user or another program with some degree of independence or autonomy, and in so doing, empoy some knowledge or representation of the user's goals or desires. [IBM95]

F.2.1 Allgemeine Definition

Als Basis für die weitere Betrachtung gehe ich von Agenten aus, wie sie Wooldridge in [Woo02] beschrieb:

> Ein Agent ist ein, in eine *Umgebung* eingebettetes Computersystem, welches fähig ist, *autonome* Handlungen innerhalb seiner Umgebung durchzuführen um seine Zielvorgabe zu erfüllen.

Aufgrund dieser Definition kann ein Agent zunächst als eingebettetes Kontrollsystem verstanden werden. Sensoren erlauben es dem Agenten seine Umgebung wahrzunehmen. Der Zustand seiner Umwelt entscheidet dabei über sein Vorgehen, welches zielorientiert und ohne weitere Benutzerinterventionen von statten geht (autonom).

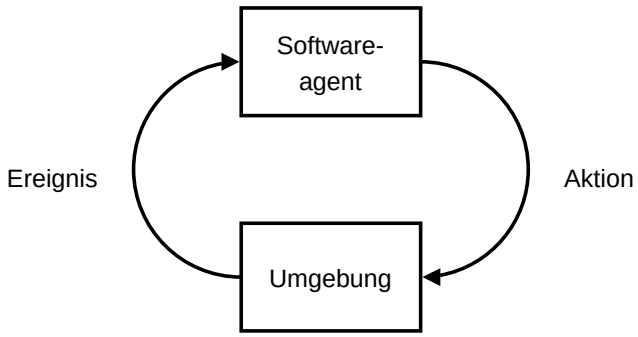


Abbildung 48: Wechselwirkung zwischen Agent und Umgebung

Ein Agent existiert also stets im Kontext einer Umgebung. Zwischen beiden Entitäten besteht eine starke Abhängigkeit, die während der Entwicklung des Agenten berücksichtigt werden muss. Jedoch besteht keine Möglichkeit auf die Ausprägung der Umgebung direkt Einfluss zu nehmen, da diese sich stets durch das Einsatzgebiet des Agenten ergibt. Die Komplexität einer Umwelt steht somit in direktem Zusammenhang zur Handlungsfähigkeit des Agenten, wie sich im folgenden Abschnitt zeigen wird.

F.2.2 Umgebung

Russel und Norvig [PN95] stellten einige Kriterien vor, mit denen sich eine Umgebung klassifizieren lässt. Das jeweils komplementäre Charakteristikum steht in Klammern.

Zugänglich (unzugänglich) ist eine Umgebung, wenn der Agent zu jedem Zeitpunkt in der Lage ist, alle relevanten Informationen zur Wahl seiner Aktion zu erfassen. Eine vollständig zugängliche Umgebung erleichtert ausserdem die Entwicklung eines Agent, da intern keine Informationen über den Zustand verwaltet werden müssen. Der Grad der Zugänglichkeit sinkt mit steigender Komplexität der Umgebung.

Deterministisch (nicht-deterministisch) bezeichnet eine Umgebung, wenn jeder Folgezustand vollständig vom aktuellen Zustand und der Aktion des Agenten abhängig ist. Dies bedeutet insbesondere, dass jederzeit, jeder Aktion ein eindeutiges Ergebnis zugeordnet werden kann.

Episodisch (nicht-episodisch) ist eine Umgebung, wenn sich die Interaktion zwischen Agent und Umgebung in einzelne Abschnitte unterteilen lässt. Wichtig ist, dass diese unabhängig voneinander sind. Der Agent entscheidet sein Handeln aufgrund der aktuellen Episode, nicht jedoch aufgrund bereits vergangener oder zukünftiger Episoden. Lernende Agenten seien hier ausgenommen, da diese natürlich aufgrund vergangener Ereignisse und Erfahrungen handeln.

Dynamisch (statisch) bezeichnet eine Umgebung, deren Zustand sich auch aufgrund externer, nicht vom Agenten ausgehender Einflüsse ändern kann. Dies bedeutet für den Agenten, dass auch während der Interaktion stets der Zustand der Umgebung überwacht werden muss, um auf mögliche kurzfristige Änderungen zu reagieren.

Diskret (kontinuierlich) ist eine Umgebung, wenn die Menge aller Wahrnehmungen und Aktionen endlich ist.

F.2.3 Intelligente Agenten

Die bisherige Definition eines Agenten ist zu grob gehalten und für den weiteren Einsatz zu schwach.

Ein Agent entscheidet sein Vorgehen stets aufgrund der Sensordaten und einer festen Regelbasis. Zwar ist es nicht einfach *Intelligenz* zu definieren, doch rein intuitiv würde man einem Agenten, wie er bisher dargestellt wurde, nur Ansätze intelligenten Verhaltens zusprechen. Sein Handeln wirkt zu statisch und er ist unfähig auf unerwartete Ereignisse zu reagieren.

Wooldridge und Jennings [WJ95] erweiterten die allgemein Definition um weitere Charakteristiken und spezifizierten den *intelligenten* Agenten. Als zentrale Eigenschaft stellten sie neben der Autonomie die flexible Handlungsfähigkeit hervor, also die Möglichkeit sich unbekannten Situationen anpassen zu können.

Autonomie bezeichnet die Fähigkeit, eigenständig handeln zu können. Dies stellt eine der zentralen Eigenschaften von Agenten dar. Der Agent kann, auf Basis seiner Zielvorgaben, sein Handeln selbständig bestimmen und agiert damit weitgehend unabhängig von Benutzereingriffen.

Reaktionsfähigkeit Reaktive Agenten reagieren auf Veränderungen ihrer Umgebung, welche sie über die ihnen zur Verfügung stehenden Sensoren wahrnehmen. Dies entspricht dem Handeln eines Agenten im allgemeinen Sinne.

Zielorientierung (Pro-Aktivität) Agenten, die zielorientiert handeln, ist es möglich, auch aus eigener Initiative heraus Entscheidungen zum Erreichen der Zielvorgaben zu treffen.

Sozialfähigkeit Die sozialen Fähigkeiten stellen ein weiteres wichtiges Kriterium intelligenter Agenten dar. Vereinfacht umfasst dies zunächst die Kommunikation mit anderen Agenten oder Benutzern. Kommunikation und Interaktion werden in Kapitel F.3 weiter behandelt.

Daneben existieren weitere Eigenschaften, die jedoch nicht allgemein anwendbar sind. Vielmehr ergeben sie sich durch das Einsatzgebiet des Agenten. Einige dieser Kriterien sind:

Mobilität Die Mobilität beschreibt die Fähigkeit eines Agenten den Ort seines Handelns zu wechseln. Ein Vorteil mobiler Agenten in Multi-Agenten Systemen ist die Minimierung der Kommunikationskosten, da der Informationsaustausch nicht zwangsläufig über weite Strecken geschehen muss. Zusätzlich erhöht der Einsatz mobiler Agenten die Ausfallsicherheit (vgl. Abschnitt F.3.1).

Rationalität Die Handlungen eines rationalen Agenten sind stets darauf ausgelegt, seine Zielvorgaben zu erreichen.

Aufrichtigkeit Dies beschreibt die Eigenschaft eines Agenten, nicht absichtlich falsche Informationen zu verbreiten.

F.3 Multi-Agenten Systeme

Die Entwicklung agentenbasierter Systeme findet ihren Ursprung im Forschungsbereich der *Verteilten Künstlichen Intelligenz* (*Distributed Artificial Intelligence*, DAI). Dieser spaltet sich, historisch bedingt, in zwei Lager: *Distributed Problem Solving, DPS* und *Multi-Agent Systems, MAS*.

Die Ausgangssituation beider Ansätze ist identisch: Ein komplexes Problem soll unter Verwendung eines agentenbasierten Systems auf effiziente Weise gelöst werden. Die Art des Problems ist dabei so gewählt, dass es die individuellen Fähigkeiten der autonom arbeitenden Agenten übersteigt und nur durch kooperatives Verhalten gelöst werden kann. Dazu wird das Problem zerlegt und in Einzelteilen an die Agenten übertragen.

DPS beschäftigt sich in erster Linie mit der Frage, wie sich Probleme zerlegen und verteilt lösen lassen. Eine Besonderheit besteht darin, dass die verwendeten Lösungsstrategien (Interaktionsvorgänge der Agenten eingeschlossen) fest in das System integriert sind.

Die Sichtweise der MAS hingegen ist eine anderen. Hier steht das soziale Verhalten der Agenten innerhalb einer Agentengesellschaft im Mittelpunkt der Betrachtung. Vorgaben über interne Abläufe, wie bei DPS, existieren nicht. Das System unterliegt der Kontrolle der Agenten. Daher spielen die Punkte Kommunikation, Koordination und Kooperation eine entscheidende Rolle und werden in diesem Kapitel genauer betrachtet.

F.3.1 Allgemeine Definition

Trivial lässt sich ein MAS folgendermaßen definieren:

> Ein Multi-Agenten System oder MAS ist ein System, welches aus mehreren autonom handelnden Agenten besteht.

Um das Konzept hinter MAS genauer zu verstehen, bedarf es jedoch einiger weiterer Charakteristiken:

- Die Agenten des Systems sind spezialisierte Problemlöser, deren Fähigkeiten jedoch so weit eingeschränkt sind, dass es ihnen nicht möglich ist das gegebene Problem eigenständig zu lösen.
- Das System besitzt eine offene Architektur. Agenten können, unabhängig von ihrer Implementierung, hinzugefügt oder entfernt werden. Damit lässt sich das System beliebig skalieren.
- Das System besitzt keine zentrale Kontrollinstanz, d. h. die Agenten unterliegen der Selbstorganisation.
- Die Datenhaltung erfolgt dezentral, d. h. jeder Agent arbeitet auf seinem lokalen Wissen.
- Alle Berechnungen verlaufen asynchron.

Auf Basis dieser Eigenschaften stellt ein Multi-Agenten System ein mächtiges Werkzeug zur Lösung komplexer Probleme dar. Das System kann als Netzwerk einzelner Problemlöser angesehen werden. Die Aufteilung des Ausgangsproblems auf einzelne Agenten nutzt die Vorteile der parallelen Verarbeitung aus. Hierdurch können Rechenleistung und andere Resourcen der zugrunde liegenden Hardware effizient genutzt werden.

Unterstützt durch die offene Struktur kann das System in hohem Maße skaliert werden, indem weitere Knoten in das Netzwerk intergiert werden. Dies kann auch im laufenden Betrieb geschehen, was die Wartung des System vereinfacht. Vor allem kritische Aufgaben profitieren von dieser Eigenschaft: Fällt einer der Agenten aus, kann dessen Aufgabe durch andere, derzeit möglicherweise nicht beschäftigte Agenten, übernommen werden.

Die hohe Flexibilität lassen diese Systeme zudem wirtschaftlich effizient arbeiten. Anstelle kostspieliger Spezialhardware reichen übliche Computersysteme aus.

F.3.2 Kommunikation

Die hier vorgestellten Systeme stellen, mangels zentraler Kontrollinstanzen, hohe Ansprüche an die eingesetzen Agenten. Dies ist insofern richtig, als dass die inter-agenten Kommunikation einen entscheidenden Faktor für die Effizienz des Systems darstellt.

Für die Agentenkommunikation müssen einige Voraussetzungen geschaffen werden, beispielsweise eine geeigneten Infrastruktur in Form von Netzwerkverbindungen (Hardwareebene) und Anwendungen und Protokollen (Softwareebene). Diese wird in der Regel bereits vom zugrundeliegenden System zur Verfügung gestellt. Weitere Voraussetzungen an die Agenten beinhalten unter anderem eine gemeinsame Kommunikationssprache und die Fähigkeit zum Austausch von Nachrichten.

Die Tatsache, dass sich Agenten eines MAS in Struktur und Implementierung stark unterscheiden können (heterogene Agenten) macht dies notwendig. Um sich dennoch verständigen zu können, ist es zwingend notwendig auf standadisierte Sprachen und Protokolle zurückzugreifen. Mit KQML wird nachfolgend eine der gängigsten Sprachen vorgestellt.

KQML Die *Knowledge Query and Manipulation Language* entstand als Entwicklung des *ARPA Knowledge Sharing Efforts*[23] im Jahr 1993, und basiert auf Ansätzen der Sprachakttheorie. Sprachakte, also Kommunikationsvorgänge, sind dabei nicht bloss als einfache Aussagen anzusehen, sondern vielmehr als Nachrichten mit einer eindeutig bestimmten Intention.

Jede Nachricht in KQML besteht aus zwei Teilen - einem Sprachakttyp, auch „performative“ genannt, der zugleich den Kontext der Informationen festlegt, und dem eigentlichen Inhalt der Nachricht. Zu den wichtigsten Sprachakttypen zählen:

Feststellungen (assertives) sind Aussagen die als Benachrichtigung oder reine Information gedacht sind.

Anweisungen (directives) dienen dazu, andere Agenten zu bestimmten Aktionen zu bewegen. Der Typ wird über das Schlüsselwort

Verpflichtungen (commissives) teilen die Aufgabenübernahme eines Agenten mit.

Deklarationen (declarations) sind Sprachakte die Fakten an andere Agenten vermitteln sollen.

Kommunikationsmechanismen Unter Kommunikationsymechanismen versteht man Methoden, Nachrichten zwischen Agenten auszutauschen. Die zwei wichtigsten Methoden sind:

Message passing bezeichnet den Vorgang, bei dem Nachrichten zwischen Agenten auf direktem Wege ausgetauscht werden. Anstelle einer zentralen Steuerung ist es Aufgabe der Agenten, den Kommunkationsablauf zu regeln. Da der Vorgang selbst stets vom sendenden Agenten initiiert werden muss, spricht man in dem Zusammenhang von einer *Bringschuld*.

Blackboard bezeichnet eine zentrale Datenstruktur, ähnlich einer Datenbank, zur Informationshaltung, die jedem Agenten des Systems zur Verfügung steht. Die Agenten können, je nach Nutzungsrecht, Informationen ablegen oder abrufen. Für den Abruf der Informationen sind die Agenten jedoch selbst verantwortlich (*Holschuld*), da das Blackboard selbst passiv arbeitet und keine Kontrolle der Kommunkationsabläufe vornimmt.

[23] Die Defense Advanced Research Projects Agency (DARPA), vormals ARPA, ist die Agentur des Verteidigungsministeriums der Vereinigten Staaten, die Forschungs-Projekte für das US-Militär durchführt

F.3.3 Kooperation / Koordination

Die Kooperation in MAS beinhaltet das gemeinsame Erarbeiten einer Lösung des Ausgangsproblems. Dabei können sich die individuellen Ziele der einzelnen Agenten widersprechen. Grundsätzlich kann zwischen kooperativem und konkurrierendem Verhalten unterschieden werden. In diesem Zusammenhang spielt die Koordination eine ebenso große Rolle.

Agenten betrachten stets nur einen Teilausschnitt des Ausgangsproblems. Ihre Ziele können sich dabei, abhängig von der Zuständigkeit, mehr oder weniger stark überschneiden. Daher ist es wichtig sich in geeigneter Form zu organisieren, um z. B. Kompetenzen festzulegen oder Teilergebnisse auszutauschen.

Die Problemlösung kann durch Möglichkeiten der Parallelisierung beschleunigt werden. Resourcen und Rechenleistung werden so optimal ausgenutzt. Dazu ist es nötig, dass Agenten mit unabhängigen Teilproblemen beauftragt werden, um Störungen durch sich überschneidende Zuständigkeiten zu vermeiden.

Um einen reibungslosen Ablauf zu garantieren, ist daher der Einsatz standadisierter Vorgehen notwendig. Mit *Contract Net* soll hier eines der bekanntesten Kooperations-Protokolle beschrieben werden.

Contract Net Das *Contract Net* Protokoll ist ein Ausschreibungsverfahren, um unbearbeitete Teilaufgaben an geeignete Agenten zu übertragen.

1. Der Vorgang der Ausschreibung wird von zentraler Stelle aus durch einen Agenten initiiert (initiator).
2. Freie, verfügbare Agenten (participants) können auf die Angebote reagieren und Kontakt aufnehmen.
3. Der Initiator wählt nun die Bewerber anhand ihrer Fähigkeiten aus, unter der Voraussetzung dass diese die gestellten Aufgaben bestmöglich lösen.
4. Die Ergebnisse der Teilnehmers werden nach Abschluss der Berechnung an den Ausschreiber übermittelt. Dieser konstruiert die Gesamtlösung.

Die Wahl des Initiators und der Teilnehmer unterliegt keinen festen Regeln. Grundsätzlich sind alle Agenten in der Lage, diese Rollen einzunehmen. Die Möglichkeit, übernommene Aufgaben selbst per *Contract Net* weiterzureichen, erzeugt eine hierarchische Struktur innerhalb der Agentengesellschaft.

Ein Kritikpunkt des vorgestellten Verfahrens sind die nicht immer optimalen Lösungen. Nicht alle Agenten nehmen am Ausschreibungsverfahren teil, somit können manche der Kompetenzen nicht berücksichtigt werden. Im schlimmsten Fall führt dies zu einer Wahl des „geringeren Übels“.

Planung Ein weiterer Punkt der Koordination und Kooperation in MAS stellt die Planung dar. Diese beinhaltet das Erstellen einer kontrollierten Abfolge einzelner Agentenaktionen und strukturiert so das Vorgehen innerhalb des Multi-Agenten Systems.

Die **zentralisierte** Planung verläuft über einen ausgewählten Agenten. Dieser besitzt eine Systemumfassende Übersicht und ist für die Erstellung des Gesamtablaufs zuständig.

Bei der **verteilten** Planung werden Aufgaben vom System auf vielen Ebenen zerlegt und an die Agenten weitergereicht. Hierzu gehört auch die Konstruktion der Gesamtlösung aus den Teillösungen.

F.4 Möglichkeiten

Viele Spiele verwenden auch heute noch Ansätze statischer regelbasierter Systeme. Das deterministische Verhalten der computergesteuerten Charaktere lässt die Spiele jedoch auf lange Sicht uninteressant werden. Einmal hinter die Taktik des Gegners gekommen, ist es leicht diesen in seine Schranken zu weisen.

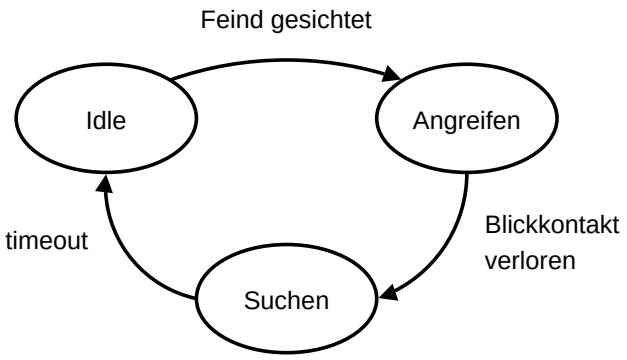


Abbildung 49: Darstellung der KI als Finite-State-Machine

Ein Nachteil vieler KI Systeme ist zudem der Mehrwaufwand bei der Anpassung an die jeweilige Spielumgebung. Wegpunkte und andere Markierungen für die Orientierung der computergesteuerten Spieler müssen sorgfältig gewählt werden. Eine kurzfristige Änderung des Leveldesigns kann gravierende Auswirkungen auf das Verhalten mit sich bringen und die Gegner orientierungslos wirken lassen. Ein rein zufälliges Manöver kann an dieser Stelle nicht mehr als intelligent bezeichnet werden und wird daher außen vor gelassen.

Alles in allem mögen diese Ansätze noch gut funktionieren, solange der Spieler keine unerwarteten Situationen provoziert. Das Fehlen von Lernmechanismen und adaptivem Verhalten, die fehlende Möglichkeit auf Basis eigener Initiative zu handeln - All dies führt im Fall eines unerwarteten Ereignisses meist zu unkotrollierten und unkoordinierten Handlungen. Agenten und Multi-Agenten Systeme könnten eine Lösung darstellen, computergesteuerten Charakteren *menschlicheres* Verhalten anzueignen. Diese wären imstande situationsabhängig zu reagieren und Lösungen abseits festgelegter Strategien zu erarbeiten. Vor allem in Mehrspielerumgebungen, bei denen Taktik und Koordination eine grosse Rolle spielen, könnte der Einsatz von MAS große Vorteile bringen.

G Wahrscheinlichkeitsrechnung, Statistik

G.1 Vorwort

Um die Methoden der Computational Intelligence richtig anzuwenden, zu analysieren und ihre Theorie zu verstehen, sind Kentnisse über bestimmte Methoden der Wahrscheinlichkeitsrechnung und mathematischen Statistik unerlässlich.

In Kapitel G.2 werden grundlegende Konzepte diskreter und kontinuierlicher Wahrscheinlichkeitsrechnung wie Verteilungsfunktionen, bedingte Wahrscheinlichkeiten, Unabhängigkeit von Ereignissen und Zufallsvariablen vorgestellt. Aufbauend darauf folgen Ungleichungen zur Berechnung asymptotischer Schranken, sowie stochastische Methoden wie das Gambler's Ruin Theorem. Einige werden, zum besseren Verständnis, an Beispielen aus der Analyse evolutionärer Algorithmen motiviert.

Kapitel G.3 gibt eine Einführung in grundlegende statistische Begriffe. Darauf aufbauend werden statistische Verfahren zum Testen von Hypothesen vorgestellt und anhand von Beispielen motiviert. Diese Methoden dienen als Grundlagen der experimentellen Analyse von Algorithmen.

Da diese Ausarbeitung als Framework für die gemeinsame Arbeit mit den Methoden der Computational Intelligence gedacht ist, wird bei aufwendigen Beweisen auf die angegebene Literatur verwiesen. Grundsätzlich soll diese Arbeit kein Lehrbuch ersetzen, sondern einen intuitiven Überblick über alle, für die PG511 relevanten, Ergebnisse der Wahrscheinlichkeitstheorie bieten.

G.2 Grundbegriffe der Wahrscheinlichkeitsrechnung

G.2.1 Einleitung

Dieses Kapitel umfasst die grundlegenden Definitionen der Wahrscheinlichkeitstheorie und damit die Basis des wichtigsten Schlüsselkonzeptes der Computational Intelligence - der Randomisierung.

Ereignisse Für jede stochastische Untersuchung von Zufallsexperimenten muss eine Menge aller möglichen Ereignisse definiert werden.

> **Definition** Ereignisraum: Die Menge Ω aller möglichen Ereignisse heisst *Menge der Elemtarereignisse* oder *Ereignisraum*.

$$\Omega := \{\omega_1, \omega_2, ..., \omega_n\},\ \Omega \neq \emptyset$$

Ein einfaches Beispiel ist der Ereignisraum eines Würfelexperiments.

$$\Omega_W := \{1, 2, 3, 4, 5, 6\}$$

> **Definition** Ereignis: Eine nicht-leere Teilmenge e des Ereignisraums[24] heisst

[24] Wobei $\mathcal{P}(\Omega)$ die Potenzmenge des Ereignisraums bezeichnet.

Ereignis.

$$e \in \mathcal{P}(\Omega)$$

Ein Ereignis gilt als *eingetreten*, falls mindestens eines der enthaltenen Elementarereignisse eingetreten ist.

Das Komplement $\overline{e}$ eines Ereignisses e, ist das Ereignis, dass keine Elementarereignisse aus e eintreten:

$$\overline{e} = \Omega - e$$

Das Ergeinis, in einem Würfelexperiment eine gerade Augenzahl zu würfeln, wäre z.B.:

$$e_{\text{gerade}} := \{2, 4, 6\}$$

und es gilt $e_{\text{gerade}} \in \mathcal{P}(\Omega_W)$.

Da man nicht an allen Mengen von Ereignissen interessiert ist, betrachten wir nur Ereignismengen mit für uns nützlichen Eigenschaften.

Definition $\sigma-$Algebra: Gilt für eine Menge von Ereignissen Σ mit $\Sigma = \{e_1, e_2, ..., e_n\}$,

$$\begin{aligned} \Sigma &\subseteq \mathcal{P}(\Omega) \\ e \in \Sigma &\Rightarrow \overline{e} \in \Sigma \\ e_k \in \Sigma,\, k \geq 1 &\Rightarrow \left(\bigcup_{i=1}^{n} e_i \right) \in \Sigma \end{aligned}$$

d.h. ist sie abgeschlossen gegenüber Komplementbildung und abzählbarer Vereinigung, so ist Σ eine σ-Algebra.

Die kleinste σ-Algebra die in Σ enthalten ist und alle anderen in Σ enthaltenen Algebren umfasst, nennen wir σ'-Algebra (auch Borelsche σ-Algebra gennant).

Wahrscheinlichkeiten Auf einer σ'-Algebra Σ, definieren wir nun eine Wahrscheinlichkeitsverteilung.

Definition Wahrscheinlichkeitsverteilung: Eine *Wahrscheinlichkeitsverteilung* ist eine Abbildung $w : \Sigma \to [0, 1]$ die jedem Element aus Σ seine Wahrscheinlichkeit zuordnet, und für die gilt:

$$\begin{aligned} w(\emptyset) &= 0 \\ w(\Omega) &= 1 \\ \forall \omega \in \Omega : w(\{\omega\}) &\geq 0 \\ w(e) &= \sum_{\omega \in e} w(\{\omega\}). \end{aligned}$$

Auf w gilt σ'-Additivität, d.h. die Wahrscheinlichkeit einer Vereinigung disjunkter Ereignisse ist gleich der Summe ihrer Einzelwahrscheinlichkeiten.

$$w\left(\dot{\bigcup_{e\in\Sigma}}\right) = \sum_{e\in\Sigma} w(e).$$

> **Definition** Wahrscheinlichkeitsraum: Für eine Menge elementarer Ereignisse Ω, einer σ'-Algebra Σ sowie einer Verteilungsfunktion w, heißt das Tupel $W = (\Sigma, \Omega, w)$ *Wahrscheinlichkeitsraum.*

Für eine sinvolle Anwendung aller stochastischen Methoden, ist die richtige Wahl des Wahrscheinlichkeitsraums von eklatanter Wichtigkeit. Betrachten wir ein einfaches Beispiel für drei Würfe einer perfekten Münze.

Beispiel eines Wahrscheinlichkeitsraums: Die Menge elementarer Ereignisse sei

$$\Omega_M := \{(x, y, z) \mid x, y, z \in \{\text{KOPF}, \text{ZAHL}\}\},$$

wobei x dem ersten, y dem zweiten und z dem drittenWurf einer Münze entspricht. Σ_M sei eine σ'-Algebra. Dann gilt für alle $(x, y, z) = \sigma \in \Sigma$:

$$w_M(\sigma) = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8}.$$

Unser Wahrscheinlichkeitsraum ist nun „3er-Münzwurf" := $(\Omega_M, \Sigma_M, w_M)$.

Haben alle Elementarereignisse die selbe Wahrscheinlichkeit, so heißt die Verteilungsfunktion *Gleichverteilung* oder *uniforme Verteilung.* Hierrauf wird im Abschnitt über Verteilungsfunktionen weiter eingegangen.

Bedingte Wahrscheinlichkeiten Betrachtet man mehrere Experimente, sollte man berücksichtigen, ob der Ausgang eines Experiments Einfluss auf den Ausgang eines anderen Experiments hat.

> **Definition** Bedingte Wahrscheinlichkeit: Sei $W = (\Sigma, \Omega, w)$ ein Wahrscheinlichkeitsraum und $e_1, e_2 \in \Sigma$ zwei Ereignisse mit $w(e_2) \neq 0$. Die Wahrscheinlichkeit das Ereignis e_1 eintritt, unter der Vorraussetzung das Ereignis e_2 bereits eingetreten ist, ist:
>
> $$w(e_1 \mid e_2) := \frac{w(e_1 \cap e_2)}{w(e_2)}.$$

Man beachte, dass e_2 hier in gewisser Weise die Menge der Elementarereignisse ersetzt, da e_2 nun alle Ereignisse enthält die eintreten dürfen. Damit entspricht die bedingte Wahrscheinlichkeit von e_1 bei gegebenem e_2 dem Verhältnis zwischen der Wahrscheinlichkeit von $e_1 \cap e_2$ und der Wahrscheinlichkeit von e_2.

Betrachten wir beispielsweise unseren 3er-Münzwurf, so ist die mindestens Wahrscheinlichkeit zwei mal Kopf zu werfen ($e_{\geq 2K}$) unter der Vorraussetzung, dass mindestens einmal Zahl geworfen wird ($e_{\geq 1Z}$):

$$\begin{aligned} e_{\geq 2K} &= \{(\text{KOPF}, \text{KOPF}, \text{KOPF}), (\text{ZAHL}, \text{KOPF}, \text{KOPF}), \\ &\quad (\text{KOPF}, \text{ZAHL}, \text{KOPF}), (\text{KOPF}, \text{KOPF}, \text{ZAHL})\} \\ \Rightarrow w(e_{\geq 2K}) &= w(\{(\text{KOPF}, \text{KOPF}, \text{KOPF})\}) + w(\{(\text{ZAHL}, \text{KOPF}, \text{KOPF})\}) \\ &\quad + w(\{(\text{KOPF}, \text{ZAHL}, \text{KOPF})\}) + w(\{(\text{KOPF}, \text{KOPF}, \text{ZAHL})\}) \\ &= \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} = \frac{4}{8} \end{aligned}$$

Wir wollen $e_{\geq 1Z}$ nicht extra aufzählen und machen uns zunutze, dass $w(e_{\geq 1Z}) = 1 - w(\Omega - e_{\geq 1Z})$.

$$\begin{aligned} \Omega - e_{\geq 1Z} &= \{(\text{KOPF}, \text{KOPF}, \text{KOPF})\} \\ \Rightarrow w(\Omega - e_{\geq 1Z}) &= \frac{1}{8} \\ \Rightarrow w(e_{\geq 1Z}) &= 1 - \frac{1}{8} = \frac{7}{8} \end{aligned}$$

$$w(e_{\geq 2K} \mid e_{\geq 1Z}) = \frac{w(e_{\geq 2K} \cap e_{\geq 1Z})}{w(e_{\geq 1Z})} = \frac{\frac{3}{8}}{\frac{7}{8}} = \frac{3}{7}$$

Da der Schnitt der Ereignisse $e_{\geq 2K}$ und $e_{\geq 1Z}$ dem Ereignis $e_{\geq 2K} - \{(\text{KOPF}, \text{KOPF}, \text{KOPF})\}$ entspricht, folgt $w(\{(e_{\geq 2K}) \cap w(e_{\geq 1Z})\})$ analog zur obigen Berechnung von $w(e_{\geq 2K})$.

Definition unabhängige Ereignisse: Sei $W = (\Sigma, \Omega, w)$ ein Wahrscheinlichkeitsraum. Zwei Ereignisse $e_1, e_2 \in \Sigma$ heißen genau dann *unabhängig*, wenn gilt

$$w(e_1 \cap e_2) = w(e_1) \cdot w(e_2).$$

Lemma. Sei $W = (\Sigma, \Omega, w)$ ein Wahrscheinlichkeitsraum. Zwei Ereignisse $e_1, e_2 \in \Sigma$, $w(e_2) \neq 0$ sind genau dann unabhängig, wenn gilt

$$w(e_1 \mid e_2) = w(e_1).$$

Beweis. "⇒":

$$w(e_1 \mid e_2) = \frac{w(e_1 \cap e_2)}{w(e_2)} = \frac{w(e_1) \cdot w(e_2)}{w(e_2)} = w(e_1)$$

"⇐":

$$\begin{aligned} w(e_1) &= w(e_1 \mid e_2) = \frac{w(e_1 \cap e_2)}{w(e_2)} \\ \Leftrightarrow \quad w(e_1) \cdot w(e_2) &= w(e_1 \cap e_2) \end{aligned}$$

□

Das letzte Ergebnis ist sehr intuitiv, da klar sein sollte, dass wenn zwei Ereignisse unabhängig sind, die Tatsache, dass eines von ihnen eintritt nichts an der Wahrscheinlichkeit des anderen ändert.

Zufallsvariablen Im Umgang mit randomisierten Algorithmen spielen Zufallsvariablen eine zentrale Rolle. Beispielsweise sind alle quantitativen Aussagen über die Rechenzeit randomisierter Algorithmen Zufallsvariablen.

> **Definition** Zufallsvariable: Sei (Σ, Ω, w) ein Wahrscheinlichkeitsraum. Für eine Menge $S \in \Sigma$, heißt jede Funktion $\chi : S \to \mathbb{R}$ *Zufallsvariable*. Für alle $x \in \mathbb{R}$ definieren wir das Ereignis $\chi = x$ als
>
> $$E(\chi = x) := \{s \in S \mid \chi(s) = x\}.$$

Für den Wertebereich einer Zufallsvariable gilt:

$$\mathbb{R}_\chi = \{x \in \mathbb{R} \mid \exists s \in S : \chi(s) = x\}.$$

> **Definition** Dichte: Die Funktion $f_\chi : \mathbb{R} \to [0, 1]$, die dem Ereignis $\chi = x$ seine Wahrscheinlichkeit zuordnet, heisst *Dichte von* χ.
>
> $$f_\chi(x) = w(E(\chi = x)).$$

> **Definiton** Verteilungsfunktion. Wir nennen die Funktion $F_\chi : \mathbb{R} \to [0, 1]$, die dem Ereignis $\chi \leq x$ seine Wahrscheinlichkeit zuordnet *Verteilungsfunktion* von χ.

Für diskrete Ω sei

$$F_\chi(x) = \sum_{y \leq x, y \in \mathbb{R}_\chi} f_\chi(y)$$

und für $\Omega = \mathbb{R}$ sei

$$F_\chi(x) = \int_{-\infty}^{x} f_\chi(x)\, dx.$$

Bemerkung: Für mehrdimensionale Zufallsvariablen sind Dichte und Verteilungsfunktion analog definiert.

Zwei Zufallsvariablen χ, ψ heißen *unabhängig*, falls die ihnen zugrundeliegenden Mengen von Elementarereignissen unabhängig sind.

Erwartungswert Der Wert, den eine Zufallsvariable im Mittel annimmt, wird *Erwartungswert* genannt. Für diskrete Zufallsvariablen χ über S, entspricht der Erwartungswert einfach der Summe ihrer Elementarereignisse, wobei jedes Ereignis mit seiner Wahrscheinlichkeit gewichtet wird:

$$\mathbb{E}(\chi) := \sum_{s \in S} s \cdot f_\chi(s).$$

Im Fall einer kontinuierlichen Zufallsvariablen ψ entspricht der Erwartungswert der Verteilungsfunktion für die Verteilung von $+\infty$:

$$\mathbb{E}(\psi) := F_\psi(+\infty) = \int_{-\infty}^{+\infty} f_\psi(x)\, dx.$$

Martingale Bei Analysen radomisierter Sucheuristiken betrachtet man häufig das Verhalten einer Zufallsvariable in Abhängigkeit von der Zeit. Dies Motiviert die Betrachtung von Folgen von Zufallsvariablen. Die folgenden Definitionen gelten normalerweise für *filtrierte Wahrscheinlichkeitsräume*. Für eine genaue Definition, siehe [Teicher97].

> **Definition** Stochastischer Prozess: Sei (Σ, Ω, w) ein Wahrscheinlichkeitsraum und $P = (\chi_t)_{t \in T}$ eine Folge von Zufallsvariablen. Für $T \in \{\mathbb{N}_0, \mathbb{R}_+\}$ heißt P *stochastischer Prozess*.
>
> **Definition** Martingal: Sei $P = (\chi_t)_{t \in T}$ ein zeitdiskreter[25] Stochastischer Prozess. Falls, für alle Zeitpunkte, der Erwartungswert von χ_{t+1} gleich dem Wert von χ_t ist, heißt P *Martingal*.
>
> $$\forall t \in T : \mathbb{E}(\chi_{t+1} \mid \chi_t, \chi_{t-1}, ..., \chi_0) = \chi_t.$$

Im Fall

$$\mathbb{E}(\chi_{t+1} \mid \chi_t, \chi_{t-1}, ..., \chi_0) \geq \chi_t$$

heißt P Sub- und im Fall

$$\mathbb{E}(\chi_{t+1} \mid \chi_t, \chi_{t-1}, ..., \chi_0) \leq \chi_t$$

Supermartingal.

> **Definition** Stoppzeit: Sei $P = (\chi_t)_{t \in T}$ ein zeitdiskreter stochastischer Prozess und τ eine Zufallsvariable mit $\tau \in \mathbb{N}_0 \cup \{\infty\}$. Ist $w(\tau = t)$ nur von $(\chi_i)_{i \leq t}$ abhängig, so heißt τ *Stoppzeit*.

Eine Stoppzeit kann man als die Wartezeit interpretieren, die vergeht, bis ein bestimmtes zufälliges Ereignis eintritt.

Optional Stopping Theorem. Sei $M = (\chi_t)_{t \in T}$ ein Martingal und τ eine Stoppzeit für M. Ist, für $n \in T$, $\tau \leq n$, $n \geq 0$, dann gilt

$$\forall i < \tau : (\chi_i \leq n < \infty) \Rightarrow \mathbb{E}(\chi_\tau) = \mathbb{E}(\chi_0).$$

D.h., falls τ eine Stoppzeit ist, ist der Erwartungswert zum Zeitpunkt des Stoppens eines stochastischen Prozesses gleich dem Anfangswert.

[25] Zeitdiskret bezeichnet hierbei eine abzählbare Folge von Zeitpunkten.

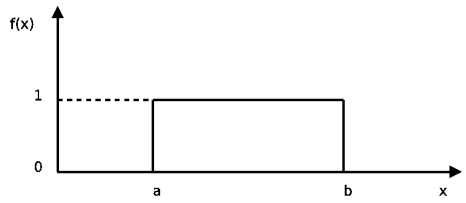


Abbildung 50: Funktionsverlauf der Dichtefunktion einer Gleichverteilung auf einem Intervall

G.2.2 Verteilungen reeler Zufallsvariablen

Bei der Arbeit mit Zufallsexperimenten und insbesondere kontinuierlichen Zufallsvariablen ist es notwendig die Verteilung einer Zufallsvariable zu kennen. In diesem Abschnitt werden die gebräuchlichsten Verteilungen vorgestellt.

Definition Indikatorfunktion. Für eine Menge M heißt die Funktion

$$1_M(x) = \begin{cases} 1 & , x \in M \\ 0 & , x \notin M \end{cases}$$

Indikatorfunktion.

Gleichverteilung Wir betrachten die Gleichverteilung (Abbildungen 50 und 51) auf dem Intervall $\Omega = [a, b]$. Dann ist ihre Dichte

$$f(x) := \frac{1}{b-a} \cdot 1_\Omega(x)$$

und ihre Verteilungsfunktion

$$F(x) := \begin{cases} 0 & , x < a \\ (x-a)/(x-b) & , a \leq x \leq b \\ 1 & , x > b \end{cases} .$$

Normalverteilung Für eine μ-σ-Normalverteilung auf $\Omega = \mathbb{R}_+$ gilt für die Dichte,

$$f_{\mu,\sigma}(x) := \frac{1}{\sigma \cdot \sqrt{2\pi}} \cdot e^{-\frac{1}{2} \cdot (\frac{x-\mu}{\sigma})^2}$$

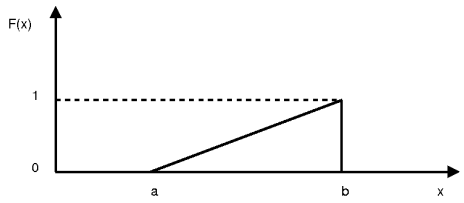


Abbildung 51: Funktionsverlauf der Verteilungsfunktion einer Gleichverteilung auf einem Intervall

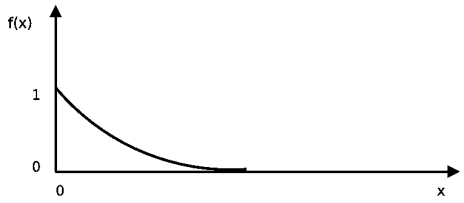


Abbildung 52: Funktionsverlauf der Dichtefunktion einer Exponentialverteilung

und die Verteilungsfunktion

$$F_{\mu,\sigma}(x) := \frac{1}{\sigma \cdot \sqrt{2\pi}} \cdot \int_{-\infty}^{x} e^{-\frac{1}{2}\cdot\left(\frac{x-\mu}{\sigma}\right)^2} dx.$$

Die Kurve der Normalverteilung wird Glockenkurve (oder Gaußkurve) gennant, da ihre Form eine Glocke ähnelt.

Exponentialverteilung Für eine Exponentialverteilung (Siehe Abbildungen 52 und 53) mit dem Parameter $\lambda \in \mathbb{R}^+$ auf $\Omega = \mathbb{R}_+$ gilt,

$$f_\lambda(x) := \lambda \cdot e^{-\lambda \cdot x} \cdot 1_\Omega(x)$$

und

$$F_\lambda(x) := \begin{cases} 0 & , x \leq 0 \\ 1 - e^{-\lambda \cdot x} & , x > 0 \end{cases}$$

G.2.3 Verteilungen diskreter Zufallsvariablen

Binomialverteilung Eine der am häufigsten genutzten diskreten Verteilungen ist die Binomialverteilung. Sie beschreibt iterierte Zufallsexperimente mit nur zwei möglichen

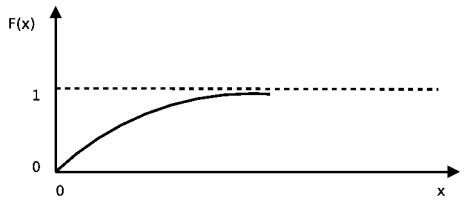


Abbildung 53: Funktionsverlauf der Verteilungsfunktion einer Exponentialverteilung

Ergebnissen. Sei p die Wahrscheinlichkeit für einen möglichen Ausgang eines binomialverteilten Zufallsexperiments (und damit $1-p$ die Wahrscheinlichkeit des anderen möglichen Ergenisses). Für die Dichte einer binomialverteilen Zufallsvariable gilt dann,

$$f_{n,p}(x) := \begin{cases} \binom{n}{x} \cdot p^x \cdot (1-p)^{n-x} & , 0 \leq x \leq n \\ 0 & , \text{sonst} \end{cases},$$

wobei $n \in \mathbb{N}_0$ der Anzahl der durchgeführten Zufallsexperimente entspricht.

Die Verteilungsfunktion ist

$$F_{n,p}(x) = \sum_{y \leq x, y \in \mathbb{R}_+} f_{n,p}(y).$$

Poissonverteilung Die Poissonverteilung wird lediglich als Approximation der Binomialverteilung erwähnt. Ihre Dichtefunktion mit Parameter $\lambda \in \mathbb{N}_0$ lautet,

$$f_\lambda(x) := \frac{\lambda^x}{x!} \cdot e^{-\lambda}.$$

Für $\lambda = p \cdot n$ (siehe Binomialverteilung) läuft ihr Grenzwert für $n \to \infty$ gegen die Binomialverteilung.

Geometrische Verteilung In der Analyse von Lauf- bzw. Wartezeiten stellt sich die Frage wie lange man warten muss, bis ein bestimmtes Ereignis eintritt. Diese Fragestellung wird mit Hilfe der geometrischen Verteilung modelliert. Für eine Wahrscheinlichkeit p eines gewünschten Ereignisses, beschreibt die Funktion

$$f(x) := (1-p)^{t-1} \cdot p$$

die Eintrittswahrscheinlichkeit nach t Schritten. Da der Erwartungswert einer geometrisch verteilten Zufallsvariable χ sehr häufig gebraucht wird, wird er hier gesondert angegeben:

$$\mathbb{E}(\chi) := \frac{1}{p}.$$

Die Herleitung wird kurz im Anhang (Kapitel G.4.1) erläutert.

Diskrete Uniforme Verteilung (Diskrete Gleichverteilung) Bei der Ziehung von Stichproben aus einer endlichen Menge M spricht man von *uniformer Verteilung*, wenn jedes Element aus M die gleiche Wahrscheinlichkeit hat, gezogen zu werden.

$$\forall m \in M : f(m) := \frac{1}{|M|}.$$

G.2.4 Stochastische Methoden und ihre Anwendung

Beispielhafte Anaylse der erwarteten Optimierzeit

> **Definition** OneMax(x): Die Funktion die einem Bitstring $x \in \{0,1\}^n$ die Anzahl der in ihm enthaltenen Einsen zuordnet heißt *OneMax(x)*.

Wie man leicht einsieht, liegt ihr Maximum im String 1^n.

$$\begin{aligned} \hat{x} &:= \underbrace{111...111}_{n\text{-mal}} \\ \text{OneMax}(\hat{x}) &= n. \end{aligned}$$

> **Definition** (1+1)-EA: Eine randomisierte Suchheuristik die genau einen Suchpunkt $x \in \{0,1\}^n$ verwaltet, in jeder Iteration einen neuen Suchpunkt durch Standard-Bit-Muation erzeugt und im Fall $f(y) \geq f(x)$ x durch y ersetzt heißt *(1+1)-EA*.

Offensichtlich ist die Anzahl der Iterationen, bis ein (1+1)-EA eine Funktion f in t-Schritten optimiert (hier: maximiert) hat, eine Zufallsvariable.

Wir berechnen nun die erwartete Optimierzeit des (1+1)-EA auf der Funktion OneMax(x). Sei x_t der Suchpunkt zum Zeitpunkt t und e_i folgendes Ereignis:

$$e_i := (\text{OneMax}(x_t) = i) \wedge (\text{OneMax}(x_{t+1}) = i + 1)$$

In jeder Iteration des (1+1)-EAs wird jedes Bit mit Wahrscheinlichkeit $\frac{1}{n}$ "geflippt", oder nicht. Betrachten wir nun ausschließlich die für uns positiven Fälle, also dass eine 0 zu einer 1 geflippt wird. Wir erhalten wie folgt für jedes Ereignis e_i eine binomialverteilte Wahrscheinlichkeit. Wir wollen eines von $n-i$ Nullbits auswählen, dieses mit Wahrscheinlichkeit $\frac{1}{n}$ flippen und alle anderen $n-1$ Bits so lassen wie sie sind. Es existieren zwar noch mehr Möglichkeiten in einer Iteration den Wert von x um genau 1 zu erhöhen, asymptotisch würde das allerdings nichts an unserer Analyse ändern. Also gilt für die Wahrscheinlichkeit eines e_i:

$$w(e_i) \geq \begin{pmatrix} n-i \\ 1 \end{pmatrix} \cdot \frac{1}{n} \cdot \left(1 - \frac{1}{n}\right)^n \geq \frac{n-i}{en}.$$

Sei χ_i die Zufallsvariable für die Anzahl der Iterationen, nach denen Ereignis e_i eintritt.

$$w(\chi_i = t) = (1 - w(e_i))^{t-1} \cdot w(e_i).$$

Durch scharfes Hinsehen identifiziert man χ_i als geometrisch verteilte Zufallsvariable. Damit gilt für den Erwartungswert von χ_i

$$\mathbb{E}(\chi_i) = \frac{1}{w(e_i)}.$$

Da wir nach Definition wissen, dass der (1+1)-EA niemals zu einem schlechteren Suchpunkt zurückkehrt, gilt, dass er spätestens nach $(n-1)$-maligem Eintreten von e_i den Punkt $\hat{x}$ gefunden und damit OneMax(x) optimiert hat. Sei nun T die Zufallsvariable der gesamten Optimierzeit des (1+1)-EAs auf OneMax(x). Es gilt:

$$\begin{aligned}
\mathbb{E}(T) &\leq \sum_{i=0}^{n-1} \frac{1}{w(e_i)} \\
&= \sum_{i=0}^{n-1} \frac{en}{n-i} \\
&= en \cdot \sum_{i=0}^{n} \frac{1}{i} \\
&< en \cdot (1 + ln(n)) \\
&= \mathcal{O}(n \cdot log(n))
\end{aligned}$$

Gambler's Ruin Theorem Seien S ein Glücksspiel mit zwei Spielern S_1, S_2 und ihrem Kapital k_1, k_2. Spieler S_1 gewinnt mit Wahrscheinlichkeit p, Spieler S_2 mit Wahrscheinlichkeit $(1-p)$. Der Verlierer einer Runde zahlt an den Gewinner den Betrag 1. Es wird solange gespielt bis einer der beiden Pleite ist.

Theorem. Ist $p \neq \frac{1}{2}$, gilt

$$w(\text{Spieler 1 geht Pleite}) = \frac{t^{k_1} - t^{k_1+k_2}}{1 - t^{k_1+k_2}},\ t = \frac{1-p}{p}.$$

Beweis. Betrachte stochastischen Prozess $(\chi_n)_{n\in\mathbb{N}}$ über $\{0, 1, ..., k_1 + k_2\}$. Sei $\chi_0 := k_1$, $\chi_{t+1} \in \{\chi_t + 1, \chi_t - 1\}$. Es gilt

$$\begin{aligned}
w(\chi_{t+1} = i+1 \mid \chi_t = i) &= p \\
w(\chi_{t+1} = i-1 \mid \chi_t = i) &= 1-p
\end{aligned}$$

Sei T der erste Zeitpunkt t zu dem $\chi_t = 0$ oder $\chi_t = k_1 + k_2$ gilt.

$$T := min\{t \in \mathbb{N} \mid \chi_t \in \{0, k_1 + k_2\}\}.$$

Da T nur von Ereignissen in der Vergangenheit abhängt, ist T Stoppzeit mit $\mathbb{E}(T) < \infty$.

Behauptung. Für $M_t := \left(\frac{1-p}{p}\right)^{\chi_t}$ ist M_t ein Martingal.

$$\begin{aligned}
\mathbb{E}\left(M_{t+1} \mid M_t = \left(\frac{1-p}{p}\right)^i\right) &= \mathbb{E}\left(M_{t+1} \mid \chi_t = i\right) = \\
&= p \cdot \left(\frac{1-p}{p}\right)^{i+1} + (1-p) \cdot \left(\frac{1-p}{p}\right)^{i-1} \\
&= \left(\frac{1-p}{p}\right)^i \cdot \left(p \cdot \frac{1-p}{p} + (1-p) \cdot \frac{p}{1-p}\right) \\
&= \left(\frac{1-p}{p}\right)^i \cdot (p + (1-p)) \\
&= \left(\frac{1-p}{p}\right)^i \\
&= M_t
\end{aligned}$$

□

Also ist M_t ein Martingal und es kann das Optional Stopping Theorem angewendet werden und es folgt

$$\begin{aligned}
\mathbb{E}(M_t) &= \mathbb{E}(M_0) \\
&= \left(\frac{1-p}{p}\right)^{\chi_0} \\
&= \left(\frac{1-p}{p}\right)^{k_1}
\end{aligned}$$

Aber offensichtlich gilt auch:

$$\begin{aligned}
\mathbb{E}(M_t) = &\quad w(\text{Spieler 1 Pleite}) \cdot \left(\frac{1-p}{p}\right)^0 + w(\text{Spieler 2 Pleite}) \cdot \left(\frac{1-p}{p}\right)^{k_1+k_2} \\
\Leftrightarrow \quad \left(\frac{1-p}{p}\right) = &\quad w(\text{Spieler 1 Pleite}) \cdot \left(\frac{1-p}{p}\right)^0 + w(\text{Spieler 2 Pleite}) \cdot \left(\frac{1-p}{p}\right)^{k_1+k_2}
\end{aligned}$$

Nach umstellen folgt direkt

$$w(\text{Spieler 1 geht Pleite}) = \frac{\left(\frac{1-p}{p}\right)^{k_1} - \left(\frac{1-p}{p}\right)^{k_1+k_2}}{1 - \left(\frac{1-p}{p}\right)^{k_1+k_2}}$$

und damit die Behauptung.

□

G.3 Statistik

Im folgenden wird ein Überblick über die grundlegensten Begriffe der Statistik gegeben. Abschließend werden diese anhand eines beispielhaften Hypothesentests verdeutlicht.

G.3.1 Grunglagen

Grundgesamtheit

> **Definition**. Die *Grundgesamtheit* G ist die Menge aller potentiellen Untersuchungsobjekte.

Stichprobe In der Statistik werden Datensätze analysiert und ihre Charakteristika zu Grafiken und Maßzahlen zusammengefasst. Allerdings ist man weniger an einem einzelnen Datensatz interessiert, sondern versucht auf Basis der gesammelten Daten eine Aussage über die Verteilung des betrachteten Merkmals in der Grundgesamtheit zu machen. Der betrachtete Datensatz ist Teilmenge der Grundgesamtheit und wird *Stichprobe* genannt.

Im Gegensatz zu einer Stichprobe kann auch eine *Vollerhebung* bei Betrachtung der kompletten Grundgesamtheit durchgeführt werden. Allerdings ist dies aus naheliegenden Gründen so gut wie nie möglich. Ein Beispiel einer Vollerhebung wäre eine Volkszählung.

Ist eine Vollerhebung allerdings nicht möglich, sollte gewährleistet sein, dass jedes Element der Grundgesamtheit die gleiche Wahrscheinlichkeit hat in die Stichprobe aufgenommen zu werden. In diesem Fall spricht man von einer *Zufallsstichprobe*. Jedes Element einer Zufallsstichprobe x_i kann formal als identisch verteilte Zufallsvariable χ_i behandelt werden.

Skalenniveaus Bei empirischen Messungen lassen sich, je nach Art der Messung, verschiedene Grade der Skalierbarkeit unterscheiden:

1. Nominalskaliert (Abbildung 54 links)

   Es existiert keine Ordnung auf den Ausprägungen der untersuchten Objekte. Es kann einzig über Gleichheit oder Ungleichheit entschieden werden.

2. Ordinalskaliert (Abbildung 54 rechts)

   Es existiert eine Ordnung, allerdings kann keine Aussage über Abstände zwischen den einzelnen Ausprägungen gemacht werden.

3. Intervallskaliert (Abbildung 55 links)

   Es können Aussagen über die Beträge der Ausprägungen gemacht werden. Jedoch existiert kein festgelegter Nullpunkt, und größen bzw. Einheiten sind willkürlich festgelegt.

4. Verhältnisskaliert (Abbildung 55 rechts)

   Metrische Skala mit festem Nullpunkt. Es können Aussagen über Verhältnisse von Mermalsausprägungen gemacht werden.

Signifikanz Zusammenhänge heißen *signifikant*, falls die Wahrscheinlichkeit gering ist, dass sie durch Zufall zustande gekommen sind. In der Praxis wird die Abweichung der Messergebnisse von einer Grundannahme betrachtet. Tritt diese Abweichung häufiger als ein zuvor festgelegter Schwellenwert auf, spricht man von statistischer Signifikanz. Häufig liegt dieser Schwellenwert bei 5%.

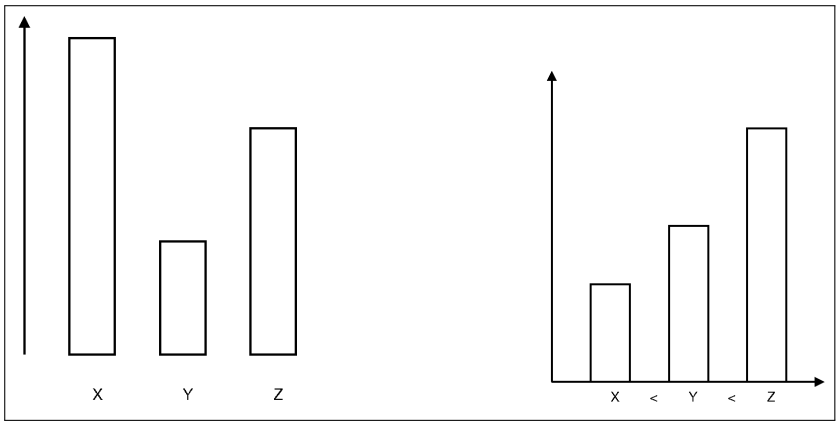


Abbildung 54: links: Nominalskalierte Daten; rechts: Ordinalskalierte Daten

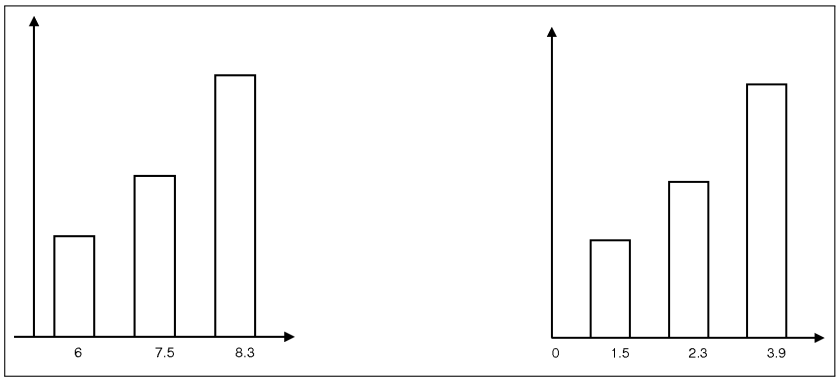


Abbildung 55: links: Intervallskalierte Daten; rechts: Verhältnisskalierte Daten

G.3.2 Hypothesentest

Einleitung Die experimentelle Analyse beliebiger Prozesse liefert üblicherweise eine Vielzahl von Beobachtungen. Basierend auf solchen Beobachtungen werden Hypothesen aufgestellt. Da eine Behauptung auf Basis von zufälligen Beobachtung (Zufallsvariablen) nicht zwingend zuverlässig sein muss, werden Hypothesentests verwendet, um die Wahrscheinlichkeit einer Fehlentscheidung einschätzen zu können.

Vorgehensweise

1. Formulierung einer anhand empirischer Daten bestehenden Annahme H_1 (Alternativhypothese) sowie die dazugehörige Nullhypothese H_0. Die Nullhypothese sagt aus, dass der in H_1 unterstellte Zusammenhang nicht besteht. Beide müssen sinnvoll gewählt werden, so dass eine der beiden übrig bleibt, falls die andere verworfen wird.

2. Berechnung einer Testgröße T aus den Werten der Stichprobe.

3. Bestimmung des kritischen Bereichs κ zu einem Signifikanzniveau α, das vor dem Ziehen der Stichprobe feststehen muss.

4. Treffen der Entscheidung.

Fehler Hypothesen können nie letztendlich verifiziert oder falsifiziert werden. Die ,,Annahme" einer Hypothese sagt nur: die vorliegende statistische Evidenz reicht nicht aus, um die Hypothese zu verwerfen.

Für die richtige bzw. falsche Schlußfolgerung eines Tests gilt folgende Fehlersystematik.

unbekannte Wahrheit → Schlußfolgerung durch Testergebnis ↓	H_0 ist wahr	H_0 ist falsch
H_0 annehmen	richtige Entscheidung	Fehler 2. Art (β-Fehler)
H_0 ablehnen	Fehler 1. Art (α-Fehler)	richtige Entscheidung

Das *Signifikanzniveau* ist die Wahrscheinlichkeit, mit der eine Nullhypothese abgelehnt wird, obwohl sie der Wahrheit entspricht. Damit entspricht das Signifikanzniveau genau dem α-Fehler (oder auch Überschreitungswahrscheinlichkeit).

Beispiel Machen wir abschließend den einfachen Test eines Würfels. Spieler S_1 (Kapitel G.2.4) ist letztendlich Pleite und behauptet der Würfel den Spieler S_2 hatte war kein idealer Würfel. Insbesondere fällt die sechs häufiger als man es erwarten würde.

Es wird $n = 100$ mal gewürfelt. Sei χ die Zufallsvariable für die Anzahl der gewürfelten 6en.

- Nullhypothese H_0 : Der Würfel ist fair, damit ist Wahrscheinlichkeit eine 6 zu Würfeln gleichverteilt und es gilt $w(6 \text{ gewürfelt}) = \frac{1}{6}$.
- Alternativhypothese H_1 : Eine 6 kommt häufiger als alle anderen Zahlen. Das hieße $w(6 \text{ gewürfelt}) > \frac{1}{6}$.

Da der eigentliche Test einfach darin besteht, dass wir eine bestimmte 6er-Grenze k festlegen und sobald $\chi > k$ akzeptieren wir H_1 ansonsten H_0, müssen wir uns vorher überlegen wie wahrscheinlich es ist, dass wir uns irren, also dass wir versehentlich einen fairen Würfel ablehnen (α-Fehler).

Da unser Experiment nur zwei uns interessierende Ausgänge hat, nämlich 6 oder nicht 6, unterstellen wir eine Binomialverteilung.

$$\begin{aligned}
\alpha(k) &:= w(\chi > k \wedge H_0) \\
&= 1 - F_{100,\frac{1}{6}}(k) \\
&= 1 - \sum_{i=0}^{k} \left(\begin{array}{c} 100 \\ i \end{array} \right) \cdot \left(\frac{1}{6}\right)^i \cdot \left(n - \frac{1}{6}\right)^{n-i}
\end{aligned}$$

Für $k = 25$ erhalten wir $0,0119$ was $1,19\%$ entspricht. Für $k = 23$ läge die Irrtumswahrscheinlichkeit bei 5%.

G.4 Anhang

G.4.1 Herleitung des Erwartungswerts geometrisch verteilter Zufallsvariablen

$$
\begin{aligned}
\mathbb{E}(\chi) &= p \cdot \sum_{k=1}^{\infty} k \cdot (1-p)^{k-1} \\
&= p \cdot \frac{\partial}{\partial(1-p)} \cdot \sum_{k=1}^{\infty} (1-p)^k \\
&= -p \cdot \frac{\partial}{\partial p} \cdot \left(\sum_{k=1}^{\infty} (1-p)^k - 1 \right) \\
&= -p \cdot \frac{\partial}{\partial p} \cdot \left(\frac{1}{p} - 1 \right) \\
&= \frac{1}{p}
\end{aligned}
$$

H Experimentelle Analyse von Algorithmen

H.1 Einführung

Üblicherweise werden Algortihmen mit den Methoden der Mathematik analysiert. Man ist bemüht, das Laufzeitverhalten des Algorithmus als eine von der Eingabegröße abhängige Funktion darzustellen. Eine solche Beschreibung liefert eine genaue Vorhersage darüber, wie sich der untersuchte Algorithmus auf eine bestimmten Eingabe verhält. Diese Methode ist jedoch nicht in der Lage, komplexe Algorithmen als Funktionen zu erfassen, denn die Möglichkeiten der Mathematik sind schnell erschöpft.

Zwei Ansätze werden praktiziert, um die Grenzen der mathematischen Analyse zu dehnen. Einerseits kann man zur asymptotischen Analyse greifen. Mit der gängigen Worst-Case Analyse kann man die Grenzen der Leistung eines Algorithmus gut einschränken. Die Laufzeit des Algorithmus lässt sich dabei als Funktion der Eingabelänge nach oben beschränken. Die asymptotische Analyse zieht jedoch einige Nachteile nach sich, wie in [MS01] zu lesen ist:

- Das asymptotische Verhalten kann sich (und meistens ist auch) auf einige wenigen Instanzgrößen beschränken, die in der realen Anwendungen nur selten (meist nie) vorkommen;
- Asymptotische Abschätzungen bergen in sich Konstanten, die in der Theorie zwar vernachlässigt werden können, in der Praxis aber einen vollständigen Durchlauf des Algorithmus in vernünftiger Zeit unmöglich machen;
- Das Worst-Case verhalten kann sich auf eine Menge von Instanzen beziehen, die in der Praxis nicht vorkommen;
- Zu guter Letzt: es ist keineswegs einfach, obere und untere Schranken zu finden, die nah genug aneinander liegen.

Ein anderer Ansatz ist das Experimentieren. Insbesondere in Hinblick auf Heuristiken wird die Bedeutung der experimentellen Analyse sichtbar, denn auf dieser Klasse von Algorithmen versagen mathematische Methoden häufig. In der Informatik werden Experimente auf zwei Weisen als Analysemittel eingesetzt. Zum Einen eignen sich Experimente, Algorithmen auf Leistungsunterschiede gegeneinander zu testen. Zu diesem Zweck werden unterschiedliche Implementierungen auf einer vorgegebenen Menge an Probleminstanzen durchlaufen und die Laufzeit wird gemessen. Dieser Ansatz, das sgn. "Competitive Testing", liefert Aussagen darüber, welcher Algorithmus besser ist. Der Vergleich zwischen Algorithmen in einem Experiment ist naheliegend; für das Verständnis der untersuchten Algorithmen birgt er jedoch nach [Hoo96] einige Mankos:

- die Fertigkeiten des Programmiers spielen eine entscheidende Rolle;
- die Parameter können so eingestellt werden, dass sie auf einer bestimmten Problemmenge gut abschneiden. Damit können aber keine Aussagen über das generelle Verhalten gemacht werden;

- jede neue Implementierung konkurriert meist mit bewährten, gut untersuchten und für Probleminstanzen, die vom Interesse sind, bereits optimierten Algorithmen. Der Vergleich ist in diesem Fall nicht fair;
- es ist nicht ersichtlich, warum der Algorithmus gerade das bobachtete Verhalten an den Tag legt. Seine Laufzeit ist sichtbar, die beeinflussenden Parameter jedoch nicht.

Warum ein Algorithmus ein bestimmtes Verhalten aufweist, lässt sich durch kontrolliertes Experimentieren herausfinden. Der intuitive Vorteil des Experiments ist darin begründet, dass ein Experiment auf einem Algorithums als Untersuchungsobjekt eine Implementierung erfordert. Durch den Prozess des Implementierens werden strukturelle Eigenschaften des Algorithmus sichtbar, mit Hilfe derer der Algorithmus besser verstanden und sein Design verbessert werden könnte. Ungeachtet dessen, ob es sich um die kompetitive oder kontrollierte Tests handelt, die experimentelle Technik ist die gleiche.

H.2 Experimentelle Analyse

In den Naturwissenschaften, die sich standardmäßig des Experiments als Mittel zum Belegen oder Widerlegen von Theorien bedienen, kommt als Resultat eines Versuchs eine allgemeingültige Aussage über ein Phänomen. Dei Gültigkeit dieser Aussagen ist durch Prinzipien über Aufbau und Durchführung von Experimenten gegeben, die sich in den Jahrhunderten der experimentellen Praxis herausgestellt haben. Bei der Untersuchung von Algorithmen sind einige Besonderheiten zu beachten. Während die Naturwissenschaften zu untersuchende Prozesse in der Natur auf ein Model abbilden und dieses Model anschließend experimentell untersuchen, liegt in der Informatik kein Naturobjekt vor. Algorithmen sind an sich bereits Modelle und die Aussagen über Sie können nicht ohne Weiteres verallgemeinert werden. In den Naturwissenschaften werden Experimente dazu benötigt, die Voraussagen einer Theorie auf ihre Gültigkeit zu testen; es ist also zuerst die Theorie dann das Experiment. Algorithmenanalyse kann losgelöst von einer Theorie eingesetzt werden. Die gängige Praxis ist eher, einen Algorithmus in allen Phasen seiner Implementierung zu testen und zu bewerten, in der Hoffnung, die verbesserungfähigen Stellen so möglichst früh in seinem Entstehungsprozess aufzuspühren.

H.3 Experimentelles Design

Um die statistische Gültigkeit der Ergebnisse zu gewährleisten, werden in der Algorithmenanalyse die Prinzipien des experimentellen Designs aus den anderen Disziplinen eingehalten. Der eigentlich Ablauf ist je nach Anwendung variabel. Es lässt sich aber eine sinnvolle Abfolge der notwendigen Schritte identifizieren, die nach Bedarf durchaus abgewandelt werden können. In Anlehnung an [MS01] und [BBPV04] lassen sich folgende Schritte festhalten:

Die ersten beiden Schritte dienen zur Vorbereitung: die Zielvorgabe des Experiments soll möglichst genau formuliert werden. Die relevanten Parameter werden in dieser Phase identifiziert. Die Parameter können endogen oder exogen sein. Endogene Parameter werden während des Experiments vom Algorithmus verändert (nicht vom Experimentator!); exogene Parameter werden vor dem Durchlauf definiert und nicht mehr verändert werden dürfen. Die Wahl der Paramter ist entscheidend für den Erfolg der Untersuchung und sie

1. Vor-experimentelle Planung
2. Festlegen der Hypothesen
3. Experimentaldesign
4. Experimentieren
   - Daten sammeln
   - Daten statistisch analysieren
   - Bewerten der ursprünglichen Hypothese(n) mit den Ergebnissen der statistischen Analyse
5. evtl. das Design anpassen und die Experimentphase wiederholen
6. Darstellung von Resultaten: Ergebnisse visualisieren und passende Schlussfolgerungen ziehen.

Abbildung 56: Ablauf experimenteller Analyse

soll mit besonderer Sorgfalt erfolgen. Die Zievorgabe wird als eine Hypothese der Art: "Der optimale Wert für den Parameter A beträgt x" oder "Es gibt einen Zusammenhang zwischen den Parametern A und B" formuliert. Falls zwei Algorithmen miteinander verglichen werden, werden in dieser Phase auch geeignete Testprobleme bestimmt, mit denen die zu untersuchende Eigenschaften kontroliert variiert werden können.

Vor dem Start des Experiments muss die initiale Parameterbelegung bestimmt und die Probleminstanzen generiert werden. Dies erfolgt im Schritt 3, dem Experimentdesign. Der Prozess wird in Algorithmendesign und Problemdesign unterteilt. Algorithmendesign umfasst Festlegung von algorithmusspezifischen Faktoren. Problemdesign dient dazu, die Parameter mit konkreten Werten zu belegen, also die Probleminstanzen zu erstellen und dient als Schnittstelle zwischen dem Algorithmus und dem konkreten Problem. So definiert das Problemdesign die initiale Belegung der Parameter und das Abbruchkriterium. Wenn man ein Experiment als einen Optimierungsdurchlauf auffasst, dann sucht man nach dem optimalen Algorithmendesign für ein gegebenes Problemdesign. Zwei Ansätze des Experimentdesigns sind Design of Experiments (DOE) und Design and Analysis of Computer Experiments (DACE). Die Wahl des geeigneten Designs kann letztlich auch nach dem Bauchgefühl getroffen werden. DOE und DACE stellen mögliche theoretische Anhaltspunkte dar; die Parametrisierung kann aber auch der Intuition folgen. So kann auch experimentelle Erfahrung des Experimentators - eine nicht zu vernachlässige Größe- das Experiment beeinflussen.

DOE DOE ist der klassische Ansatz, in dem die statistische Standardverfahren (bespielsweise lineare/ nonlineare Regression) zum Einsatz kommen. Im DOE wird nach [BB03] folgende Nomenklatur benutzt: Als Faktoren oder Designvariablen werden im DOE alle Parameter bezeichnet, die während des Experiments variiert werden (z.B. Temperatur oder Druck). Designvariablen werden als Vektor $(x_1, \cdots, x_k)^T$ dargestellt. Unterschiedliche Parameterwerte heißen Levels (beispielsweise kann die Temperatur hoch oder tief

sein). Diese können qualitativ oder quantitativ sein.

Der Designraum ist ein k-dimensionaler Raum, der durch die Ober- und Untergrenzen der Designvariablen definiert wird. Ein Designpunkt (oder eine Stichprobe) ist der Vektor einer bestimmten Belegung der Designvariablen. DOE wählt Designpunkte, die sich auf den Rändern des Designraums befinden. Man unterscheidet dabei zwischen verschiedenen Designs:

1. One-factor-at-a-time Design: bei jedem Durchlauf wird ein Paramter variiert, während alle anderen konstant bleiben. Dieses Design liefert Aufschluss über den Einfluss eines einzelnen Parameters, vorausgesetzt alle anderen Faktoren sind fest. Dadurch kann man lediglich für den Einfluss eines einzelnen Parameters auf die Performanz des Algorithmus eine Aussage treffen und nicht für die (deutlich interessantere) Interaktionen zwischen einzelnen Parametern. Die Gültigkeit der Aussagen beschränkt sich auf die festgelegte Parametereinstellungen und kann nicht generalisiert werden.

2. Faktorielle Designs: bei jedem Experimentdurchlauf werden Kombinationen von Faktoren und Stufen ausprobiert, um die Effekte der Paramter und ihrer Kombinationen zu bestimmen. Der Effekt eines Faktors ist dabei definiert als die Änderung des Resultats bewirkt durch eine Änderung im Level des Faktors. Die Levels sollen dabei möglichst extrem gewählt werden, ohne die praxisrelevanz der Einstellungen zu verlieren.

    - Vollfaktorielle Designs: es werden Versuche für alle Kombinationen von Einstellungen durchgeführt. Für k Faktoren mit 2 Stufen je Faktor sind das 2^k Experimente.
    - Teilfaktorielle Designs: einige Kombinationen werden vernachlässigt; dies ist für Kombinationen, die viele Parameter umfassen, manchmal zulässig. Insgesamt braucht man für k Faktoren mit 2 Stufen pro Faktor $2^{(k-p)}$ Durchläufe des Experiments.

Faktorielle Designs sind vorzuziehen, wenn der Einfluss von zwei oder mehr Parametern gleichzeitig untersucht wird. Dabei kann man mit Hilfe teilfaktorieller Designs einen generellen Überblick über eventuell vorhandenen Interaktionen gewinnen. Die vollfaktoriellen Designs können auch Aussagen über die genauere Art der Interaktionen liefern.

DACE DACE ist ein modernes Paradigma des Experimentdesigns. DACE Methoden beschreiben Funktionen, die das System modellieren, als einen stochastischen Prozess und ermöglichen eine Vorhersagen der unbekannten Werte in diesem Prozess. Diese Modellierung verzichtet auf die Annahme der unveränderten Varianz, wie es in DOE bei der linearen Regression vorausgesetzt wird. Da DACE ursprünglich für deterministische Experimente gedacht war, müssen Experimentalläufe wiederholt werden, um das stochastische Verhalten nachzuahmen. Im Gegensatz zu DOE Designs werden im DACE Designpunkte innerhalb des Designraumes ausgewählt.

- Monte Carlo Design: die Designpunkte werden zufällig aus einem Intervall $[x_L, x_U]$ gewähltt.

- Latin Hypercube Design (LHD): bei p zu wählenden Designpunkten wird der Definitionsbereich jeder Variable in p gleiche Abschnitte aufgeteilt. Dies ergibt (für einen 2-dimensionalen Designraum) ein Gitter. Die Designpunkte werden so gewählt, dass (1) jede Zelle des Gitters höchstens einen Designpunkt enthält und dass (2) jede Projektion auf einen der Parameter höchstens einen Designpunkt enthält. LHD stellt sicher, dass die Designpunkte gleichmäßig über den gesamten Designraum verteilt ausgewählt werden.

Problemdesign Die Initialisierung der Parameter kann unterschiedlich erfolgen. Es können Werte gewählt werden, die erfahrungsgemäß als günstig bekannt sind, oder aber auch folgendermaßen:

- Deterministisch: alle initiale Suchpukte werden mit deterministisch ausgewählten Werten belegt. Dabei können entweder alle Vektoren gleiche Belegung haben oder aber unterschiedliche.
- Randomisiert: die Suchpunkte werden mit zufälligen Werten belegt. Die Werte können dabei aus einer Menge an unabhängigen gleichverteilten Variablen initialisiert werden. Eine andere Möglichkeit ist, jeden Suchpunkt mit einem anderen Vektor aus unabhängigen Zufallsvariablen zu initialisieren.

Über Maße und Gütekriterien Zu definieren, was und wie gemessen wird, ist ein zentraler Punkt im Aufbau eines Experiments. Im Zusammenhang mit Algorithmen sind unterschiedliche Maße denkbar. Naheliegend ist es, die Laufzeit oder den Wert der Lösung zu betrachten. Es kann aber auch nützlich sein, sich die strukturellen Eigenschaften des Algorithmus anzuschauen. Dies ist möglich über die Betrachtung von der Anzahl von Iterationen oder Anzahl der Aufrufe bestimmter Prozeduren. Ungeachtet des eingentlichen Maßes ist es wichtig sicherzustellen, dass die Messung reproduziert werden kann. Beispielsweise ist die Laufzeit von den Eigenarten der Maschine abhängig, an der die Simulation durchgeführt wurde.

Unter dem Begriff Gütekriterium sind die Maße für die Qualität der Ergebnisse zu verstehen. Je nach der Fragestellung, können unterschiedliche Maße benutzt werden:

- Mean Best Fitness (MBF): Aus den besten Fitness-Werten mehrerer Läufe wird der Mittelwert bestimmt. Die beste Fitness kann als der beste Wert in der Population am Ende eines Durchlaufs definiert werden;
- Sucess Rate (SR): Dieses Maß kann benutzt werden, wenn das optimale Ergebniss bekannt ist (oder eine hinreichend gute Lösung angegeben werden kann). In diesem Fall kann der Anteil erfolgreicher Läufe (success rate) ermittelt werden und zur Bewertung des Algorithmus benutzt.
- Average Evaluations to Solution: hierbei wird die durchschnittliche Anzahl der Zielfunktionsauswertungen, bis die optimale (oder ausreichend gute) Lösung gefunden wurde, als Maß gewählt. Die besondere Schwierigkeit liegt hier darin, dass eine Lösung angegeben werden muss.

Andere Gütekriterien sind denkbar und können beliebig (solange auch sinnvoll) definiert werden.

H.4 Ergebnisse präsentieren

Die Präsentation der Ergebnisse der experimentellen Untersuchung sowie der gesamten Vorgehensweise zielt auf bessere Zugänglichkeit der benutzten Methodik für alle späteren Replikationsversuche. In [Pre07] sind für den Aufbau der Berichte über experminetelle Analysen sieben Schritte vorgeschlagen:

- den Untersuchungsgegenstand formulieren;
- vorexperimentelle Planung beschreiben. Dieser Schritt beinhaltet alle Untersuchungen, die zur Formulierung der Hypothesen führen, sowie alle Daten, die auf diesem Wege gesammelt und analysiert wurden. Insbesondere negative Resultate sollen berücksichtigt werden;
- die Frage, auf die eine Antwort gesucht wird, und alle daraus abgeleiteten statistischen Hypothesen formulieren;
- das Experimentaldesign erfassen. Diese Beschreibung soll detalliert genug sein, um den Versuch nachbauen zu können;
- die resultierenden Daten (grafisch) präsentieren;
- alle gemachten Beobachtungen beschreiben. Hier sind besonders die Abweichungen von dem erwarteten Verhalten interessant.
- Dikussion und Bewertung. In diesem Schritt soll über das Akzeptieren oder Ablehnen der Hypothesen entscheden werden, die Entscheidung begründet, sowie mögliche Erklärungen für das beobachtete Verhalten des Algorithmus wiedergegeben werden.

Ein standardisiertes Vorgehen beim Berichten über die experimentellen Untersuchungen in der Algorithmenanalyse sind essentiell für die wissenschaftliche Berechtigung experimenteller Analysen.

I Spieltheorie

I.1 Grundlagen

I.1.1 Die wichtigsten spieltheoretischen Begriffe im Überblick

Ein *Spiel* Γ ist jede Situation, in der ein Individuum Entscheidungen treffen muss, die eine deutliche Auswirkung auf das Ergebnis dieser Situation haben. Es wird nach [Dav72] vollständig beschrieben durch das Tupel $(\mathcal{S}, \mathcal{A}, e)$ mit

- der endlichen Menge $\mathcal{S}$ der Spieler, deren Elemente mit $s_a \in \mathcal{S}$ mit $a \in \mathbb{N}$ und $a \leq |\mathcal{S}|$ bezeichnet werden,
- der Menge $\mathcal{A}$ aller möglichen Zugmöglichkeiten des Spiels,
- einer Auszahlungsfunktion e, die für jeden Spielausgang den Spielern s_a je eine Auszahlung e_a zuweist, somit ist $e(\cdot) = (e_1, \ldots, e_{|\mathcal{S}|})$ und
- den Spielregeln, soweit sie durch die Zugmöglichkeiten $\mathcal{A}$ festgelegt sind.

Besteht ein Spiel aus mehreren aufeinander folgenden gleichförmigen Teilspielen, nennt man diese *Partien*. Ein Beispiel für ein Spiel mit Partien ist das dem Schach ähnliche Spiel Thud [PTP01]. Hierbei treten die Spieler zweimal gegeneinander an, wobei beim zweiten Mal die Figuren getauscht werden. Siegt ein Spieler in beide Partien, gewinnt er das Spiel, ansonsten gilt das Spiel als unentschieden.

Die Entscheidungen, die während eines Spieles zu treffen sind, bezeichnet man als *Züge* $\mathcal{A}$. Die einzelnen Züge eines Spieles werden bezeichnet als $\mathfrak{m}_v$ mit $v \in \mathbb{N}$ und $v \leq |\mathcal{A}|$ [Neu28]. Hierbei ist zu beachten, dass nicht Züge als zeitlicher Abfolge gemeint sind, sondern jeder Zug eine Möglichkeit eines Spielers darstellt, durch eine Zugmöglichkeit aus $\mathcal{A}$ von einer Spielkonfiguration zu einer anderen zu wechseln.

Die Spieltheorie unterscheidet Spieltypen anhand ihrer Teilnehmerzahl n mit $n = |\mathcal{S}|$. Typische Spiele sind Ein- (Solitär, Patience), Zwei- (Mühle, Schach) und Dreipersonenspiele (Die drei Magier [Rü85], Skat).

Eine *Strategie* [Dav72] ist eine vollständige Beschreibung des Verhaltens eines Spielers für ein Spiel. Eine Strategie eines Spielers beschreibt also genau, wie sich ein Spieler in welcher Spielsituation verhalten wird; mit ihrer Hilfe könnte auch ein Beauftragter des Spielers an seiner statt am Spiel teilnehmen.

Beim jeweiligen Zug $\mathfrak{m}_v$ können die Alternativen $\mathcal{A}_v \subset \mathcal{A}$ gespielt werden. Die Menge $\mathcal{A}_{av} \subseteq \mathcal{A}_v$ ist diejenige Menge möglicher Züge, aus der der Spieler s_a in dieser Spielkonfiguration seine Zugmöglichkeit wählen kann. Es ist $\mathcal{A}_{av} = \mathcal{A}_v$, wenn der Spieler in dieser Spielkonfiguration alle Elemente aus $\mathcal{A}_v$ wählen kann und $\mathcal{A}_{av} \subsetneq \mathcal{A}_v$, wenn ihm gewisse Züge aufgrund der Spielregeln vorenthalten sind. Der konkret gewählte Spielzug für $\mathfrak{m}_v$ des Spielers s_a wird dann bezeichnet als α_{avi} mit $\alpha_{avi} \in \mathcal{A}_{av}$ und $i \in \mathbb{N}$ mit $i \leq |\mathcal{A}_{av}|$.

Die verschiedenen Strategien der Spieler s_a werden als $\mathcal{B}_{a,m}$ mit $m \in \mathbb{N}$ bezeichnet. Die Strategie des Spielers ist die Menge $\mathcal{B}_{a,m} = \bigcup_{i \in I} \mathcal{A}_{va}$ über die Indexmenge I von $\mathcal{A}$, die alle Züge α_{avi} enthält, die der Spieler verwendet, wenn er seine Strategie m spielt. Durch die Anwendung verschiedener Strategien der am Spiel beteiligten Spieler kann es

zu unterschiedlichen Spielverläufen kommen, die mit $\mathcal{B}_o$ mit $o \in \mathbb{N}$ bezeichnet werden. Eine Menge $\mathcal{B}_o$ ist durch Kombination der von den Spielern gewählten Strategien mit $\mathcal{B}_o = \{\mathcal{B}_{a,m} | a, m \in \mathbb{N}, a \leq |\mathcal{S}|\}$ beschrieben.

Sind alle $\alpha_{avi} \in \mathcal{B}_{a,m}$ von dem Spieler selbst ausgewählt (so genannte „persönliche Züge“ [Dav72] oder „Züge erster Art“ [HI96]), nennt man $\mathcal{B}_{a,m}$ eine *reine Strategie*. Ist jedes α_{avi} durch einen Zufallszug zustande gekommen, wird $\mathcal{B}_{a,m}$ als *Zufallsstategie* bezeichnet. Enthält eine Strategie $\mathcal{B}_{a,m}$ sowohl zufällige wie persönliche Züge, wird sie *gemischte Strategie* [Nas50] genannt.

Nach Definition von Strategie und Spielausgang ist eine spezifiziertere Darstellung der Auszahlungsfunktion e möglich. Die Auszahlungsfunktion eines Spieles ist definiert als $e(\mathcal{B}_o) = (e_1, \ldots, e_{|\mathcal{S}|})$ mit den *Auszahlungen* [HI96] oder Gewinne e_a der Spieler s_a bei gegebenem Spielverlauf $\mathcal{B}_o$. Die Auszahlung eines Spielers kann negativ sein, dies ist zum Beispiel der Fall, wenn er ein Spiel verliert und seinen Einsatz bezahlen muss.

Ist die Summe $\varepsilon = \sum_{j=1}^{|\mathcal{S}|} e_j$ aller Auszahlungen unabhängig von den gespielten Strategien konstant, spricht man von einem *Spiel mit konstanter Summe*, ist $\forall \mathcal{B}_o \; \varepsilon = 0$, sprich man von einem *Nullsummenspiel* [Dav72]. Ein Spiel mit konstanter Summe lässt sich leicht in ein Nullsummenspiel umformen, so dass im Folgenden ausschließlich Nullsummenspiele und nicht-Nullsummenspiele betrachtet werden.

Anhand des Wissens, welches jeder Spieler über die Züge aller Spieler besitzt, werden Spiele in zwei Kategorien aufgeteilt [HI96]: Sind den Spielern zu jedem Zeitpunkt alle möglichen Züge aller Spieler bekannt, wie es zum Beispiel beim Schach der Fall ist, spricht man von Spielen mit *perfekter* oder *vollständiger Information*. Bleibt mindestens ein Spieler über bestimmte Züge der anderen Spieler im Ungewissen (zum Beispiel bei Kartenspielen, bei denen jeder Spieler nur diejenigen Karten sehen kann, die er selbst auf der Hand hat, die der anderen hingegen nicht), so spricht man von Spielen mit *imperfekter* oder *unvollständiger Information*. Unvollständige Information liegt insbesondere in jedem Fall vor, in dem das Spiel ein Zufallselement beinhaltet.

Besteht die Möglichkeit der Kommunikation zwischen den Spielern, können sie Absprachen treffen. Existiert zudem eine Kontrollinstanz, welche die Einhaltung dieser Absprachen einfordert, so nennt man diese Absprachen auch *bindende Abmachung* [Dav72], *bindende Verpflichtung* [HI96] oder *unverletzliche Absprache* [NM67]. Man spricht von *kooperativen Spielen*, wenn bindende Abmachungen getroffen werden können. Andernfalls nennt man ein solches Spiel ein *nicht-kooperatives Spiel* [NM67].

Zur Darstellung von (Zweipersonen-)Spielen existieren zwei Normalformen: die *strategische Form* [HI96] wird auch Auszahlungsmatrix genannt und ist eine tabellarische Darstellung der Strategien mit ihren Auszahlungen (s. Abbildung 57). Liegt ein Nullsummenspiel vor, genügt es, die Gewinne eines Spielers anzugeben, ansonsten müssen beide Gewinne genannt werden.

Die zweite Darstellungsform ist die *extensive Form* [HI96], auch *Spielbaum* genannt. Spielbäume sind Bäume mit der Menge der Knoten $\mathcal{V}$ und der Menge der Kanten $\mathcal{E}$, in denen jeder Knoten $v \in \mathcal{V}$ eine Spielkonfiguration und jede Kante $e \in \mathcal{E}$ einen Zug eines Spielers darstellt. Da Kanten in Bäumen immer in Richtung der absteigenden Ebenen gerichtet sind, sind sie insbesondere geeignet, um Spiele mit sequenziellem Charakter zu visualisieren.

$s_2 \downarrow \ s_1$	$\mathcal{B}_{1,1}$	$\mathcal{B}_{1,2}$
$\mathcal{B}_{2,1}$	4/3	2/1
$\mathcal{B}_{2,2}$	1/6	2/2

$s_2 \downarrow \ s_1$	$\mathcal{B}_{1,1}$	$\mathcal{B}_{1,2}$
$\mathcal{B}_{2,1}$	4/2	3/3
$\mathcal{B}_{2,2}$	2/4	5/1

$s_2 \downarrow \ s_1$	$\mathcal{B}_{1,1}$	$\mathcal{B}_{1,2}$
$\mathcal{B}_{2,1}$	2 /(-2)	0 /(0)
$\mathcal{B}_{2,2}$	0/(0)	1/(-1)

Abbildung 57: Auszahlungsmatrizen für ein nicht-Nullsummenspiel (links), Spiel mit konstanter Summe (Mitte) und Nullsummenspiel (rechts) mit zwei Spielern s_1 und s_2 und je zwei Strategien $\mathcal{B}_{\cdot 1}$ und $\mathcal{B}_{\cdot 2}$. Die Auszahlung der nicht-Nullsummenspiele ist in der Notation „Auszahlung für Spieler s_1“ / „Auszahlung für Spieler s_2“ angegeben. Im Falle des Nullsummenspiels sind die Auszahlungen für Spieler s_2 in Klammern angegebenen um deutlich zu machen, dass ihre Angabe optional ist.

Die Wurzel stellt den initialen Spielzustand dar, die Blätter stehen für die Endkonfigurationen und werden mit den Auszahlungen für beide Spieler (liegt ein Nullsummenspiel vor, genügt die Auszahlung eines Spielers) in der entsprechenden Konfiguration beschriftet. Jede Ebene im Spielbaum repräsentiert hierbei einen Zug eines Spielers in Abhängigkeit von den vorherigen Zügen (= Knoten geringerer Tiefe). Um deutlich zu machen, welcher Spieler am Zug ist, können die Knoten oder Ebenen mit dem entsprechenden Spieler markiert werden (s. Abbildung 58).

Liegt unvollständige Information vor, bedeutet das im Baummodell, dass der Spieler den Knoten, in dem er sich „befindet“, nicht von anderen auf dieser Ebene unterscheiden kann. Dieses wird verdeutlicht durch eine (von den Kanten deutlich unterscheidbare) Verbindung dieser Knoten, wie Abbildung 58 (rechts) zeigt.

Welche Darstellungsform die bessere ist, hängt vom zu beschreibenden Spiel ab. Die strategische Form ist, weil sie in beliebiger Reihenfolge (Spalte-Zeile oder Zeile-Spalte) ausgelesen werden kann, besser geeignet, Spiele mit gleichzeitigen Zügen zu modellieren, als Spielbäume, deren Richtung Wurzel → Blätter eine Auswertungsreihenfolge vorgibt und daher eine Spiel- oder Zugreihenfolge impliziert, die nicht notgedrungen vorliegt. Aus den selben Gründen eignet sich die extensive Form besser für sequenzielle Spiele als Auszahlungstabellen.

Durch den Kunstgriff, Verbindungen zwischen nicht-unterscheidbaren Knoten zu zeichnen, sind Spielbäume zwar grundsätzlich geeignet, Spiele mit unvollständiger Information zu modellieren, diese Darstellung wird aber bei größeren Bäumen unübersichtlich. Darum ist bei Spielen mit unvollständiger Information die strategische Darstellungsform vorzuziehen. Aus den selben Gründen eignet sich die extensive Form besser für sequenzielle Spiele als Auszahlungstabellen.

Durch Einführung des Spielbaumes als Darstellungsform kommen wir zu einer weiteren Möglichkeit, die Strategie eines Spielers darzustellen: die Strategie $\mathcal{B}_{a,m}$ mit $a, m \in \mathbb{N}, a \leq |\mathcal{S}|$ des Spielers $s_a \in \mathcal{S}$ ist die Menge $\mathcal{E}_{a,m} \subset \mathcal{E}$ der Kanten des Spielbaumes, die die Züge repräsentieren, die der Spieler s_a in „seinen“ Knoten wählt, wenn er seine Strategie $\mathcal{B}_{a,m}$ spielt.

I.1.2 Lösungskonzepte

Ein *Lösungskonzept* ist eine Menge von Kriterien, mit denen das Verhalten der Spieler beschrieben wird, und die dieses (vernünftig handelnde Spieler vorausgesetzt) vorhersagen. Ein Auszug gängiger Lösungskonzepte wird im Folgenden beschrieben.

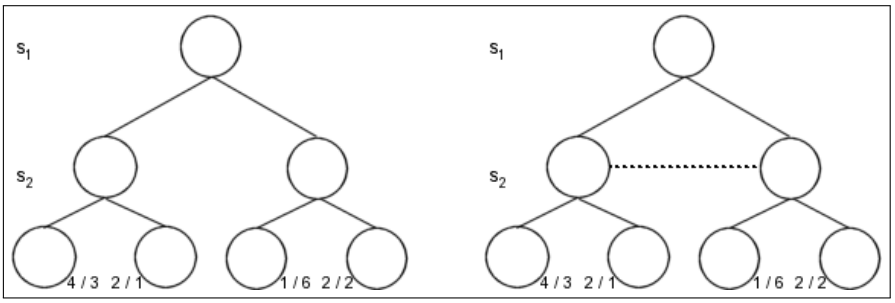


Abbildung 58: Spielbäume für ein Zweipersonenspiel mit vollständiger (links) und unvollständiger (rechts) Information und je einem Zug. Die Ebenen zeigen die Züge der Spieler s_1 und s_2, die Blätter die Endzustände mit der jeweiligen Auszahlung für s_1 / Auszahlung für s_2 in den jeweiligen Zuständen.

Dominante Strategie Eine *dominante Strategie* [HI96] ist eine Strategie, die dazu führt, dass ein Spieler ungeachtet der Züge seiner Gegenspieler immer ein bestmögliches Spielergebnis erreicht. Eine Strategiekombination $\mathcal{B} = \{\mathcal{B}_{1,m^{(1)}}, \ldots, \mathcal{B}_{|\mathcal{S}|,m^{(|\mathcal{S}|)}}\}$ nennt man ein *dominantes Gleichgewicht* [HI96], wenn alle Strategien $\mathcal{B}_{a,m} \in \mathcal{B}$ dominante Strategien aller Spieler $s_a \in \mathcal{S}$ sind. Dominante Strategien existieren nur für einen kleinen Teil von Spielen, so dass dieses Lösungskonzept oftmals keine Anwendung findet. Ein Beispiel für ein Spiel mit dominanter Strategie ist das Gefangenendilemma (I.1.4)

Minimax-Theorem Das *Minimax-Theorem* [Neu28] ist ein Lösungskonzept für Zweipersonen-Nullsummenspiele mit perfekter Information. Die Spieler s_1 und s_2 spielen ein Spiel mit den Auszahlungen e_1 für s_1 und e_2 für s_2. Bei einem Nullsummenspiel ist die Summe der Auszahlungen 0, also gilt $e_1 + e_2 = 0$ und somit $e_1 = -e_2$, und es genügt, die Auszahlung e_1 zu betrachten. Besitzt s_1 eine Strategie $\mathcal{B}_{1,m}$, mit der er sicherstellen kann, dass er für alle Strategien $\mathcal{B}_{2,\cdot}$ e_1 mindestens eine Auszahlung e erreicht, und besitzt s_2 eine Strategie $\mathcal{B}_{2,n}$, für die für alle Strategien $\mathcal{B}_{1,\cdot}$ $e_1 \leq e$, wird das Spiel mit der Auszahlung $e_1 = e$ beendet, da jeder Spieler bei Spielen mit perfekter Information alle möglichen Züge des Gegners kennt und vorauszusetzen ist, dass sich jeder Spieler immer größtmöglichen Gewinn oder kleinstmöglichen Verlust zu sichern versucht.

Nash-Gleichgewicht Im Gegensatz zum Minimax-Theorem ist das Nash-Gleichgewicht [Nas50] auch für nicht-Nullsummenspiele mit perfekter Information anzuwenden, in denen es keine dominanten Strategien gibt. Abbildung 59 zeigt ein solches Spiel; es gibt weder für Spieler s_1 noch für Spieler s_2 eine Strategie, die ihm unabhängig von der Strategie des Gegners eine Mindestauszahlung garantiert.

Eine Strategiekombination $\mathcal{B} = \{\mathcal{B}_{1,m^{(1)}}, \ldots, \mathcal{B}_{|\mathcal{S}|,m^{(|\mathcal{S}|)}}\}$ ist dann ein Nash-Gleichgewicht, wenn die Strategie $\mathcal{B}_{a,m} \in \mathcal{B}$ jedes Spielers $s_a \in \mathcal{S}$ ihm einen bestmöglichen Spielausgang garantiert, vorausgesetzt, dass alle anderen Spieler ihre Gleichgewichtsstrategien spielen. Die Spieler haben nach Nashs Gleichgewichtstheorie kein Interesse daran, von ihrem Gleichgewicht abzuweichen, da sie sich dann (vernünftig handelnde Gegenspieler vorausgesetzt) schlechter stellen würden. Es ist leicht zu sehen, dass in Abbildung 59 die Kombination $(\mathcal{B}_{1,2}, \mathcal{B}_{2,2})$ ein (und das einzige) Nash-Gleichgewicht sind. Da perfekte Information vorliegt, kann der jeweilige Gegenspieler auf eine Abweichung von der Gleichge-

$s_1 \downarrow \ s_2 \rightarrow$	$\mathcal{B}_{2,1}$	$\mathcal{B}_{2,2}$	$\mathcal{B}_{2,3}$
$\mathcal{B}_{1,1}$	8 / -8	1 / 1	-8 / 8
$\mathcal{B}_{1,2}$	1 / 1	2 / 2	1 / 1
$\mathcal{B}_{1,3}$	-8 / 8	1 / 1	8 / -8

Abbildung 59: Nicht-Nullsummen-Zweipersonenspiel der Spieler s_1 und s_2 ohne dominante Strategie. Die Auszahlung ist in der Notation „Auszahlung für Spieler s_1“ / „Auszahlung für Spieler s_2“ angegeben.

wichtsstragie reagieren und seinerseits eine Strategie spielen, die dem abweichenden Spieler ein schlechteres (und sich selbst ein besseres) Ergebnis einbringt.

Weitere Gleichgewichte, wie zum Beispiel das von Thomas Bayes, sind der angegebenen Literatur, insbesondere [NM67] und [HI96], zu entnehmen.

I.1.3 Nutzen

Der Nutzen u eines Spiels für einen Spieler s_a ist nach [Dav72] eine individuelle Funktion $u_a(\cdot)$, die die „Quantifizierung“ der Präferenzen dieses Spielers für bestimmte Spielausgänge oder Spiele ausdrückt. Bei Spielen mit quantifizierten Auszahlungen kann die Auszahlungsfunktion e eine Nutzenfunktion der Spielausgänge sein, wenn sie die folgenden Axiome erfüllt.

Hat ein Objekt a einen höheren Nutzen als ein Objekt b, wird erwartet, dass der Spieler ein Objekt a einem Objekt b vorzieht, $a \succ b :\Leftrightarrow u(a) > u(b)$; haben beide Objekte den gleichen Nutzen, ist der Spieler ihnen gegenüber indifferent, $a||b :\Leftrightarrow u(a) = u(b)$.

Ein einfaches Beispiel für ein Spiel mit Nutzenfunktion ist eine Lotterie, bei der der teilnehmende Spieler s_a mit bestimmten Wahrscheinlichkeiten $p(\cdot)$ mit $p \in [0, 1]$ verschiedene Objekte g_q aus einer Menge $\mathcal{G}$ mit $q \in \mathbb{N}, q \leq |\mathcal{G}|$ gewinnen kann, wobei $\sum_{q=1}^{|\mathcal{G}|} p(g_q) = 1$. Der Nutzen eines Spieles ist dann definiert als $u = \sum_{q=1}^{|\mathcal{G}|} p(g_q) \cdot u(g_q)$.

Beispiel: Gibt es in einer Lotterie einen Apfel, eine Birne und eine Clementine zu gewinnen, der Spieler schriebe dem Apfel einen Nutzen von 4, der Birne einen Nutzen von 6 und der Clementine einen Nutzen von 8 zu, und die Wahrscheinlichkeit p, einen Apfel zu gewinnen, wäre $p(a) = 0,5$, die einer Birne und einer Clementine je 0,25, so wäre der Nutzen dieses Spiels $0,5 \cdot 4 + 0,25 \cdot 6 + 0,25 \cdot 8 = 5,5$.

Bedingungen für die Existenz einer Nutzenfunktion Wenn die Präferenzen eines Spielers durch eine Nutzenfunktion ausgedrückt werden sollen, müssen (nach [Dav72] und [May01]) diese Präferenzen bzw. die Nutzenfunktion die folgenden Axiome erfüllen:

(1) Vollständigkeit Je zwei Objekte (zum Beispiel Gewinne, Geldsummen oder allgemein Auszahlungsbeträge) a und b sind vergleichbar, ein Spieler muss immer einem den Vorzug geben oder indifferent sein, also $a \succ b$, $b \succ a$ oder $b||a$.

(2) Transitivität Präferenz und Indifferenz sind transitiv, gilt $a \succ b$ $(a||b)$ und $b \succ c$ $(b||c)$, so gilt auch $a \succ c$ $(a||c)$.

Beweis: $>$ und $=$ sind transitiv, es gilt $u(a) > u(c)$ $(u(a) = u(c))$, wenn $u(a) > u(b)$ $(u(a) = u(b))$ und $u(b) > u(c)$ $(u(b) = u(c))$; Einsetzen in die Definition ergibt die geforderte Transitivität.□

(3) Unabhängigkeit Vorzug und Indifferenz bleibt erhalten, wenn die Objekte gegen gleichwertige Objekte ausgetauscht werden.

Beweis: Zwei Objekte sind gleichwertig, wenn sie den gleichen Nutzen haben. Gelte $a \succ b$ und a wäre gegen c auszutauschen, gilt $u(a) = u(c)$ und $u(a) > u(b) \Rightarrow u(c) > u(b) \Rightarrow c \succ b$. □

(4) Stetigkeit Gibt es in einer Lotterie α drei Objekte a, b, und c mit den Wahrscheinlichkeiten $p(a)$, a zu gewinnen, $p(b)$, b zu gewinnen und $p(c)$, c zu gewinnen, gilt $a \succ b$ und $b \succ c$ sowie $p(a) = \varphi$ und $p(c) = (1 - \varphi)$, so ist bei $\varphi = 0$ die $\alpha||c$ und es wird b dem Spielen der Lotterie vorgezogen, bei $\varphi = 1$ ist $\alpha||a$ identisch und es wird a vorgezogen, daher muss es ein φ mit $0 < \varphi < 1$ geben, das den Spieler zwischen b und dem Spielen in α indifferent macht.

Beweis: $b||\alpha \Leftrightarrow u(b) = u(\alpha)$. Es gilt nach Definition $u(\alpha) = p(a) \cdot a + p(c) \cdot c$, also $u(\alpha) = \varphi \cdot u(a) + (1 - \varphi) \cdot u(c)$, woraus folgt $b||\alpha$ bei $\varphi = \frac{u(b)-u(c)}{u(a)-u(c)}$.□

(5) Präferenzen bei steigenden Wahrscheinlichkeiten Je größer die Gewinnchancen für den bevorzugten Preis sind, desto besser ist die Lotterie. Gebe es o.B.d.A zwei Lotterien α und β mit der Menge der zu gewinnenden Objekte $\mathcal{G} = \{a, b\}$ sowie die Wahrscheinlichkeit $p_\alpha(a)$, a in α zu gewinnen und $p_\beta(a)$, a in β zu gewinnen, ist $a \succ b$ und gilt $p_\alpha(a) > p_\beta(a)$, folgt daraus auch $\alpha \succ \beta$.

Beweis: Nach Definition gilt $a \succ b \Leftrightarrow u(a) > u(b)$, der Nutzen eines Spieles ist definiert als $u = \sum_q^{|\mathcal{G}|} p(g_q) \cdot u(g_q)$, somit ist der $u(\alpha) = p_\alpha(a) \cdot u(a) + p_\alpha(b) \cdot u(b)$ und $u(\beta) = p_\beta(a) \cdot u(a) + p_\beta(b) \cdot u(b)$. Da $p_\alpha(a) > p_\alpha(b)$ und $u(a) > u(b)$ folgt $u(\alpha) > u(\beta)$ und somit $\alpha \succ \beta$.□

(6) Indifferenz gegenüber dem Spiel Die Spieler sind gegenüber dem tatsächlichen Spielmechanismus indifferent, einzig der Einzelnutzen der Preise und ihre Wahrscheinlichkeiten bestimmen die Einstellungen der Spieler zu Preisen und Lotterien.

I.1.4 Klassische Beispiele

Gefangenendilemma Das Gefangenendilemma (nach [HI96]) ist das klassische Beispiel für ein Zweipersonen-, nicht-Nullsummenspiel mit dominanter Strategie, die Beschreibung dieses Spiels lautet wie folgt:

Zwei Gefangene werden verdächtigt, Komplizen bei einer schweren Straftat gewesen zu sein, ohne weitere Hinweise hätten beide eine Haftstrafe von zwei Jahren zu verbüßen. Der Staatsanwalt spricht mit jedem Verdächtigen allein und macht ihnen das folgende Angebot: gesteht einer, wird dieser mit nur einem Jahr Gefängnis bedacht und sein Komplize bekommt eine Haftstrafe von zehn Jahren. Gestehen beide, werden sie beide mit fünf Jahren Haft bestraft. Die beiden Delinquenten können sich bis zu ihrer Entscheidung nicht

$s_1 \downarrow$ / $s_2 \rightarrow$	$\mathcal{B}_{2,1}$	$\mathcal{B}_{2,2}$
$\mathcal{B}_{1,1}$	2/2	10/1
$\mathcal{B}_{1,2}$	1/10	5/5

Abbildung 60: Das Gefangenendilemma in strategischer Form; als „Auszahlung“ wird die Haftzeit in Jahren in der Notation „Haft für Spieler s_1“ / „Haft für Spieler s_2 “ angegeben.

Frau $\downarrow$ / Mann $\rightarrow$	Kino	Sport
Kino	3/1	0/0
Sport	0/0	1/3

Abbildung 61: Das Spiel „Kampf der Geschlechter“ in strategischer Form mit den Spielern Mann und Frau und den Strategien „ins Kino“ bzw „zum Sport gehen“ sowie den Auszahlungen in der Notation Frau / Mann

miteinander besprechen. Abbildung 60 zeigt dieses Spiel in strategischer Form mit den Strategien $\mathcal{B}_{1,1} = \mathcal{B}_{2,1} = \textit{nicht gestehen}$ und $\mathcal{B}_{1,2} = \mathcal{B}_{2,2} = \textit{gestehen}$.

Es handelt sich um ein Spiel ohne Kooperationsmöglichkeit, da keine Kommunikation besteht. Augenscheinlich gibt es für beide Spieler eine dominante Strategie, nämlich zu gestehen.

Kampf der Geschlechter Das Spiel „Kampf der Geschlechter“ („Battle of sexes“ nach [HI96]) ist ein typisches Beispiel für das Nash-Gleichgewicht. Es beschreibt (extrem verkürzt) folgende Geschichte: Ein Mann und eine Frau treffen sich zufällig und werden getrennt, bevor sie sich für den kommenden Abend verabreden können, obwohl sie sich wiedersehen wollen. Beide wissen, dass *er* am Abend dieses Tages zu einem Sportereignis, *sie* zu einer Kinopremiere gehen möchte und haben die Möglichkeit, ins Kino oder zum Sport zu gehen. Die Auszahlungen symbolisieren die Freude, die jeder von ihnen an diesem Abend hat: 0 - ein völlig misslungener Abend, 1 - langweiliges Ereignis, aber wenigstens mit einem netten Menschen zusammen, 3 - ein großartiger Abend. Wie in Abbildung 61 zu sehen ist, gibt es zwei Nash-Gleichgewichte, denn (Kino, Kino) und (Sport, Sport) sind wechselseitig die besten Antworten, ohne weiteres Wissen ist nicht klar, welches dieser Gleichgewichte (wenn überhaupt eines) realisiert wird. Die Lösung dieses Spieles ist nicht eindeutig; weitere Faktoren, die in der Spielbeschreibung nicht angegeben sind, könnten diese beeinflussen.

I.1.5 Ein modernes Beispiel

Im Text „John Nash auf der Damentoilette“ untersucht Rieck [Rie07] die Frage, warum Herrentoiletten immer so dreckig sind, warum Damen dafür länger warten müssen und ob Unisex-Toiletten die Lösung darstellen:

- Warum sind Herrentoiletten (es geht hierbei um Urinale) so dreckig? Die Antwort ist einfach: es ist eine dominante Strategie, mehr Abstand zum Urinal zu halten. Tritt man näher heran, so riskiert man, in den Urin seines Vorgängers zu treten, und das alleine zu dem Zweck, dass dem später Kommenden dies erspart bleibt.

Eine vorhergehende Inspektion des Bodens ist aufwändiger, als einfach Abstand zu halten (mehr Aufwand → höhere Kosten → geringerer Nutzen), es kommt also zum selben Problem wie beim Gefangenendilemma: Auch wenn es für alle besser wäre, wenn alle näher heran träten, sieht die dominante Strategie anders aus.

- Warum brauchen Frauen so lange? Damentoiletten sind angeblich sauberer als Herrentoiletten. Das dortige Dilemma ist hingegen, dass, nachdem Frau lange in der Schlange gestanden hat, um eine Kabine zu besetzen, es keinen Nutzen bringt, sich zu beeilen; hier ist die dominante Strategie also, sich Zeit zu lassen.

Sind Unisextoiletten die Lösung? Die Antwort ist nein, denn

- Männer werden sich diesem Modell nach nicht hinsetzen, denn die Toilette könnte vom vorherigen Nutzer beschmutzt sein, der sich nicht hingesetzt hat. Wie bereits für die Frauen beschrieben, gibt es aber keinen Grund, die einmal durch lange Wartezeit ergatterte Kabine schnell wieder zu verlassen. Wenn sie sich also hinsetzten, bräuchten sie ebenso lange wie Frauen (und müssten dementsprechend auch längere Wartezeiten in Kauf nehmen), blieben sie stehen, verschmutzten sie die Toilette.
- Frauen würden sich weiterhin nicht beeilen und müssen trotzdem unter den verschmutzen Toiletten leiden, sie können also entweder dreckige Toiletten mit kürzeren Wartezeiten oder saubere Toiletten mit ebenso langen Wartezeiten erwarten.

Damit sieht man, dass durch die Einführung von Unisex-Toiletten die Situation für alle schlechter würde.

Es versteht sich von selbst, dass dieses Beispiel ein unvollständiges, pointiertes Modell zur Verdeutlichung ist, das weder die anatomischen noch andere Faktoren berücksichtigt.

I.2 Theorie der evolutorischen Spiele

Falken und Tauben tränken sich aus derselben, sehr begrenzten Wasserstelle – wer wird sich durchsetzen und überleben? Die Falken, die um das lebensnotwendige Nass zu kämpfen bereit sind, oder die pazifistischen Tauben?

Diese und ähnliche Fragestellungen sind die der *Theorie der evolutorischen Spiele* [HI96]. Diese Theorie unterscheidet darin von der klassischen Spieltheorie, dass die Spieler nicht die Möglichkeit zur bewussten Strategiewahl haben, weil sie sich dem Vorliegen einer strategischen Entscheidungssituation nicht bewusst sind; daher sind die Entscheidungen sind nicht voneinander abhängig. Aus diesem Grund werden in der evolutorischen Spieltheorie Spieler und Strategie gleichgesetzt und die Spieler allein durch ihre Strategie repräsentiert. Diese drückt Verhaltensstandards, Ideen, Symbole sowie angeborenes Verhalten aus.

Mehr noch als die klassische bietet die evolutorische Spieltheorie ein Modell für Spiele mit großen Gruppen von Teilnehmern, wie zum Beispiel die Evolution als eines verstanden werden kann.

I.2.1 Das Modell der evolutorischen Spiele

Die folgenden fünf Spielregeln (nach [Mai92] beziehungsweise [HI96]) beschreiben das Grundmodell des evolutorischen Spiels:

1. Jeder Spieler sieht sich als Mitglied einer sehr großen Grundgesamtheit von Spielern, von der er unterstellt, dass entscheidungsrelevante Merkmale in ihr *zufällig* verteilt sind.
2. Die Spieler entscheiden nicht strategisch und gehen bei ihren Entscheidungen nicht davon aus, dass sich ihre Gegenspieler optimierend verhalten.
3. Die Spieler lernen in dem Sinne aus dem Spielverlauf, als dass die Vergangenheit des Spiels (also alle Strategieentscheidungen, die vor dem betrachteten Zeitpunkt stattgefunden haben) darüber bestimmt, welche Strategien zu diesem Zeitpunkt realisiert werden.
4. Ein Spieler geht bei seiner Entscheidung stets davon aus, dass sie keinen Einfluss auf zukünftige Perioden hat – weder in Bezug auf die Auszahlungen (Fitness), noch auf das Verhalten (Strategie) der Mitspieler.
5. Die Spieler treffen im Spielverlauf immer paarweise aufeinander.

Des Weiteren verwendet die evolutorische Spieltheorie die im Folgenden beschriebenen Begriffe zur Beschreibung der von ihre betrachteten Spiele.

Population Eine Population [Mai92] kann als ein (nicht abgeschlossener) Strategieraum zu einem bestimmten Zeitpunkt des Spiels verstanden werden. Eine Population nennt man *monomorph*, wenn alle Mitglieder identisch sind beziehungsweise die selbe Strategie verfolgen, *polymorph*, wenn die Mitglieder sich unterscheiden.

Fitness Die Beziehung eines Spielers zu seiner Umwelt und damit sein Potential, in dieser Population erfolgreich zu sein, drückt die evolutorische Spieltheorie durch die *Fitness* [HI96] eines Spielers aus. Diese ist in hohem Maße abhängig von der Situation und von der Umwelt und wird gleichgesetzt mit den Auszahlungen des Spieles. Die Fitness beziehungsweise Auszahlung eines Spielers s_a wird berechnet durch die Fitness- oder Nutzenfunktion $u_i(\cdot)$.

Wird biologische Evolution aus spieltheoretischer Sicht betrachtet, sieht man oft die Zahl der Nachkommen als Fitness an.

Mutant Die evolutorische Spieltheorie nennt einen Spieler einen *Mutanten* [HI96], wenn er eine andere Strategie verfolgt als die bisher in der Grundgesamtheit der Spieler (die man in diesem Kontext auch *etablierte Spieler* [HI96] nennt) vorkommenden. Hierbei ist es gleich, ob dieser Spieler sich aus einer Population entwickelt oder als neuer Spieler von außen herantritt.

I.2.2 Koevolution

Koevolution bezeichnet in der Evolutionsbiologie die Abhängigkeit der Evolution zweier oder mehrerer konkurrierender Spezies, die in ihrer Entwicklung jeweils einen Einfluss auf die andere nehmen.

In der Informatik spricht man, nach [Wie03], von Koevolution, wenn die Fitness eines Individuums von der Beziehung dieses Individuums zu anderen Individuen abhängt. Fitness muss, in seinen Worten, eine *subjektive*, *internale* Messgröße sein.

Die evolutorische Spieltheorie führt als Fitnessmaß den Spielausgang oder die Auszahlung ein. Dies ist eine Messgröße, die nach Wiegand [Wie03] subjektiv, also unabhängig von der Auszahlung des anderen Spielers, und external, also den weiteren Weg der Evolution nicht beeinflussend, ist. Dies ist konform mit den Spielregeln nach [Mai92], da die vierte Regel der evolutorischen Spieltheorie besagt, dass eine Entscheidung keine Auswirkung auf die zukünftigen Auszahlungen haben darf. Bei nicht-Nullsummenspielen sind weiterhin die Auszahlungen in der Regel nicht aneinander gekoppelt. Wiegands Forderung für koevolutionäre Algorithmen, die Fitness müsse eine internale, subjektive Messgröße sein, ist bei Spielen naturgegebener maßen erfüllt, da die Auszahlungsfunktion jedem Spieler eine Auszahlung *in Abhängigkeit aller gespielten Strategien* zuweist (Vgl. I.1.1).

Aufgrund dieser Gleichheit scheint es so zu sein, dass die evolutorische Spieltheorie nach [HI96] und die Koevolution nach [Wie03] gleiches aus unterschiedlichen Blickwinkeln beschreiben.

Der *Red Queen Effect*

> „»Well, in *our* country,«, said Alice, still panting a little, »you'd generally get to somewhere else – if you run very fast for a long time as we've been doing.«
>
> »A slow sort of country!«, said the [Red] Queen. »Now, *here*, you see, it takes all the running *you* can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!«“ [Car78, Seite 147]

Die Verwendung eines Fitnessmaßes, dass einzig aus der Abhängigkeit der Individuen zueinander bestimmt wird, bringt gewisse Probleme im Verständnis der Ergebnisse mit sich. Eines dieser Probleme, der *Red Queen Effect*, benannt nach dem eingangs zitierten Abschnitt aus [Car78], wird im Folgenden beschrieben; weitere Beispiele finden sich in der angegebenen Literatur zu evolutionären Algorithmen und Koevolution.

Dadurch, dass man über die Individuen nur weiß, wie sie gegeneinander abschneiden, kann man, wenn sich das Fitnessmaß über längere Zeit nicht ändert, nicht feststellen, ob die Entwicklung stagniert, sich die Individuen also nicht weiter verbessern, oder ob sich sich gleichförmig verbessern, so dass trotz gestiegener „Siegesfähigkeit“ einer Population die gestiegenen Fähigkeiten der „Gegner“ dies ausgleichen.

Gerade wegen dieses möglichen Wettrüstens hat sich die Bezeichnung *Red Queen Effect* aus der Evolutionsbiologie hierfür durchgesetzt. Biologen haben allerdings (im Gegensatz zu Informatikern) die Möglichkeit, einzelne Individuen unter Laborbedingungen zu untersuchen und neben dem beschriebenen Fitnessmaß weitere, individuelle Fähigkeiten zu bestimmen.

Das prominente Beispiel für dieses „evolutionäre Wettrüsten“ und den *Red Queen Effect* in der Biologie ist der Rauhäutige Gelbbauchmolch (Taricha granulosa) [Wik06b] beziehungsweise sein einziger Fraßfeind, die Gewöhnliche Strumpfbandnatter (Thamnophis sirtalis) [Wik06a]: Rauhäutige Gelbbauchmolche produzieren in ihrer Haut das Nervengift Tetrodotoxin. Die Gewöhnliche Strumpfbandnatter ist der einzige Organismus, der dieses Gift unbeschadet zu sich nehmen kann und sich von daher auch von Rauhäutigen Gelbbauchmolchen ernährt. In den Gebieten, in denen diese Tiere vorkommen, zeigt sich, dass die Produktionsrate des Giftes und die Kompensationsfähigkeit abhängig von der Populationsdichte variieren: Je mehr Gift die Molche produzieren, desto mehr können die Nattern kompensieren (und umgekehrt).

Wiegand schließt seine Darstellung dieses Phänomens mit folgendem Satz: „[...]der *Red Queen Effect* [ist] weder gut noch schlecht, oder, präziser formuliert, kann er gut oder schlecht sein, es ist allerdings unmöglich zu sagen, was genau.“. [Wie03, S.17]

I.3 Analyse von Spielen

Abschließend soll nun auf die Frage eingegangen werden, wie konkrete Spiele im Bezug auf eine Gewinnstrategie analysiert werden können.

Zur spieltheoretischen Analyse sind Spiele im spieltheoretischen Modell zu erfassen, es gilt also, die Menge $\mathcal{S}$ der Spieler (mit ihrer Anzahl $n = |\mathcal{S}|$) und die Zugmöglichkeiten $\mathcal{A}$ zu identifizieren und zu beschreiben. An dieser Stelle ist es möglich, zu bestimmen, ob es sich um kooperatives oder nicht-kooperatives Spiel, sowie ein Spiel mit vollständiger oder unvollständiger Information handelt. Danach gilt es, die möglichen Strategien $\mathcal{B}_{a,m}$ der Spieler zu bestimmen und aus den sich ergebenden m^n Strategiekombinationen $\mathcal{B}_o$ und den sich daraus ergebenen Auszahlungen die Auszahlungsfunktion e zu erstellen. Wird eine Nutzenfunktion benötigt, ist zu überprüfen, ob e die Axiome in I.1.3 erfüllt. Ist dies nicht der Fall, ist für jeden Spieler eine Nutzenfunktion u über die Spielausgänge zu erstellen. Anhand der Auszahlungs- oder Nutzenfunktion können dann die verschiedenen Strategiekombinationen in Hinblick auf ihre Auszahlungen oder ihren Nutzen verglichen und Strategiegleichgewichte identifiziert werden.

Es ist klar zu sehen, dass dieses Vorgehen nur bei Spielen mit sehr wenigen Zugmöglichkeiten zum Erfolg führen kann, da allein die zu modellierenden Spielkonfigurationen riesige Ausmaße annehmen können.

I.3.1 Münzwurfspiel

Das hier angegebene Beispiel ist (leicht abgewandelt und gekürzt) zitiert aus [Dav72, S. 45f].

Anna und Ben legen jeweils 5 € auf den Tisch und werfen verdeckt eine Münze, die Punktzahl des Wurfes wird mit 1 (Kopf) oder 2 (Zahl) bestimmt und bleibt geheim.

Anna spielt zuerst. Sie kann entweder „passen“ oder weitere 3 € setzen, wenn sie passt, wird die Punktzahl der Spieler verglichen. Der Spieler mit der höheren Zahl bekommt die kompletten 10 €, bei Gleichheit erhalten beide 5 €.

Anna ↓ / Ben →	$\mathcal{B}_{b1}$	$\mathcal{B}_{b2}$	$\mathcal{B}_{b3}$	$\mathcal{B}_{b4}$
$\mathcal{B}_{a1}$	0	0	0	0
$\mathcal{B}_{a2}$	5	0	0,5	4,5
$\mathcal{B}_{a3}$	1,25	0,75	0	2
$\mathcal{B}_{a4}$	3,75	-0,75	0,5	1,5

Abbildung 62: Auszahlungsmatrix für das „Münzwurfspiel“. Die Tabelle zeigt die Strategien von Anna ($\mathcal{B}_{a\cdot}$) und Ben ($\mathcal{B}_{b\cdot}$) sowie (weil ein Nullsummenspiel vorliegt) die durchschnittlichen Auszahlungen von Anna aus $\frac{\text{Auszahlung bei Kopf+Auszahlung bei Zahl}}{2}$.

Wenn Anna setzt, ist nun Ben an der Reihe und kann entweder „passen“, wobei Anna den kompletten Einsatz erhält, oder „sehen“ (wobei er den bereits vorhandenen 13 € weitere 3 € hinzufügen muss), dann werden die Münzwürfe verglichen, und der Spieler mit der höheren Zahl bekommt die kompletten 16 €, bei Gleichheit erhalten beide 8 €.

Jeder Spieler hat vier mögliche Strategien, um das Spiel zu lösen: Beide können immer passen ($\mathcal{B}_{\cdot 1}$), immer setzen beziehungsweise sehen ($\mathcal{B}_{\cdot 2}$) sowie passen bei 1 und setzen/-sehen bei 2 ($\mathcal{B}_{\cdot 3}$) und setzen/sehen bei 1 und passen bei 2 ($\mathcal{B}_{\cdot 4}$). Abbildung 62 zeigt die Auszahlungsmatrix des Spiels.

Es zeigt sich, dass $\mathcal{B}_{a2}$ die Strategien $\mathcal{B}_{a4}$ und $\mathcal{B}_{a1}$, und $\mathcal{B}_{b3}$ die Strategien $\mathcal{B}_{b4}$ und $\mathcal{B}_{b1}$ dominiert. Das formale Modell verdeutlicht, was intuitiv klar war: wenn Anna Zahl wirft, sollte sie setzen, wenn Ben Zahl wirft, sollte er sehen; zweifelhaft ist nur, was geschehen soll, wenn einer der Spieler Kopf wirft. Eine Berechnung zeigt: In diesem Fall wäre die beste Strategie für Anna, einen zufälligen Zug zu machen und in 0,6 der Fälle $\mathcal{B}_{a2}$ und in 0,4 $\mathcal{B}_{a3}$ zu spielen, das würde für sie einen durchschnittlichen Gewinn von 0,30 € bedeuten. Für Ben gilt der umgekehrte Fall, mit dem er seinen durchschnittlichen Verlust auf 0,30 € beschränken kann.

I.3.2 Pac-Man

Um das Spiel „Pac-Man“ von Namco zu analysieren, gilt es, die gleichen Schritte zu machen: Es handelt sich um ein Zweipersonenspiel (Mensch gegen Computer), bei dem der menschliche Spieler eine, der Computerspieler vier Spielfiguren bewegen kann. Das Spiel ist ein Spiel imperfekter Information, da zwar dem menschlichen Spieler die Position aller Objekte auf dem Spielfeld bekannt sind, der Computer allerdings nur über so viel Wissen verfügt, wie seine Spielfiguren gemäß ihrer Sichtweite zur gegebenen Zeit beobachten können. Es handelt sich weiterhin um ein Spiel ohne Kooperationsmöglichkeit, da es den Spielern weder eine Möglichkeit gibt, miteinander zu kommunizieren, noch Verträge einzugehen.

Definiert man die höchste Auszahlung (*perfektes Spiel* mit 3.333.360 Punkten) als möglichen Spielgewinn, den sich Mensch und Computer je nach Erfolg ihrer Strategien teilen, ist Pac-Man ein Spiel mit konstanter Summe. Ein Spiel wäre ein Durchgang vom Start bis zu Level 255, eine Partie jeweils ein Level.

Betrachtet man die 780 möglichen Positionen auf dem Bildschirm und die Tatsache, dass dort, wo eine Spielfigur steht, keine andere stehen kann, gibt es $\frac{780!}{775!} \approx 2,85 \cdot 10^{14}$ verschie-

dene Kombinationen von Positionen, von denen jede Figur in maximal vier Richtungen auf ein benachbartes Feld ziehen kann. Da es zudem 776! verschiedenen Möglichkeiten gibt, auf welchen Positionen noch zu verzehrende Punkte und 4! mögliche Kombinationen von übriggebliebenen Kraftpillen gibt, ist $|\mathcal{A}| \approx 1,58 \cdot 10^{1923}$, weshalb auf eine Darstellung als Spielbaum oder Auszahlungsmatrix verzichtet wird.

Strategien „Pac-Man“ als Spieler zu gewinnen, müssen das Fressen aller Punkte beinhalten, ohne sich dabei von den Geistern fangen zu lassen. Möglichkeiten hierzu sind zum Beispiel die als „Pac-Man Mazes“ bekannten Wege, das Verzehren der den „Power-Pills“ nahen Punkten in der Zeit, in der nach Einnahme dieser „Pac-Man“ die Geister fressen kann, oder „Pac-Man“ in Gegenden des Labyrinthes zu navigieren, in denen er möglichst weit von möglichst vielen Geistern entfernt ist.

Eine gute Strategie für den Computer wäre, den Spieler mit den vier Geistern so einzukreisen, dass es keinen Weg mehr gibt, zu entkommen. Dies sollte ein Leichtes sein, wenn jeder Geist bei jedem Zug ein Feld weit gehen kann und perfekte Information vorliegt (und damit sehr frustrierend für den menschlichen Spieler, weshalb Namco dies wohl auch nicht vorsieht).

I.4 Anwendungsgebiete & Fazit

Die Spieltheorie findet Anwendung in der Ökonomie und in vielen weiteren Gebieten, so zum Beispiel in den Politikwissenschaften, Evolutionsbiologie, Soziologie und in der Analyse von randomisierten Algorithmen. Als Einstieg in diese Themen empfehle ich Downs, A.: „An Economic Theory of Democracy“, 1957; Smith, M. J.: „Evolution and the Theory of Games“, 1982; Kaminski, M. M.: „Games Prisoners Play“, 2004 und Jansen, T.: Materialien zur Vorlesung Evolutionäre Algorithmen, Wintersemester 2006/2007, Universität Dortmund.

Betrachtet man die Spieltheorie in Hinblick auf die Aufgabe in der PG, CI-Methoden zur Optimierung von Spielweise in Computerspielen einzusetzen, scheint sie als Methode hierzu nicht geeignet zu sein. Schon um einfachste Spiele mit nur einer Entscheidungsmöglichkeit zu beschreiben, bedarf es großem Aufwand; kompliziertere Brettspiele (wie zum Beispiel Schach) vollständig zu beschreiben ist bis heute auch mit elektronischen Hilfsmitteln unmöglich. Die vollständige Beschreibung von Computerspielen, noch mehr Zustände haben können als Brettspiele, ist daher gar nicht zu erwarten. Möglich ist, Auszahlung beziehungsweise Nutzen zu definieren, um Gewinnen oder Verlieren fassen zu können und allgemeine Überlegungen zu Lösungskonzepten und Strategien anzustellen, um ein Spiel besser zu begreifen. Jedoch: ein komplettes formales Modell aufzustellen, scheint, gerade vor diesem Hintergrund, nicht sinnvoll.

J Experimentelle Bewertung/Vergleich von Spielstrategien und Spielspaß

J.1 Einleitung

Spiele lösen einen Reiz beim Benutzer aus und wollen ihn möglichst lange erfreuen. In der heutigen Zeit kann man sich als Spieler an immer besseren Graphiken und neuen (Sound-)Effekten erfreuen, wohingegen sich die „Intelligenz“ der Computergegner nicht besonders weiterentwickelt hat. Eine Erklärung für diese Entwicklungstendenz liefert [Nar04]:

> „Menschen sind sehr optisch orientierte Lebewesen, und ein schöner Sonnenuntergang ist ihnen leichter zu verkaufen, als irgendeine Fähigkeit des Gegners zumindest teilweise intelligent zu handeln.“

In unserer PG soll das Hauptziel sein, Computergegner interessanter und unvorhersehbarer zu gestalten.

Hierfür muss der Entertainmentfaktor hoch sein, das heißt ein Spiel darf nicht zu schwer zu erlernen und mit zu starken Gegnern ausgestattet sein. Auf der anderen Seite darf es den Spieler aber nicht nach kurzer Zeit unterfordern und somit langweilen. Die Ursachen hierfür können in der begrenzten Aufgabenstellung liegen. Meistens sind aber die Computergegner der Grund für ein sinkendes Interesse am Spiel, da sie mit etwas Erfahrung allzu leicht zu durchschauen und somit zu besiegen sind. Abhängig ist der Spielspaß ferner von der Zeit, in der sich ein Spieler mit dem Spiel beschäftigt, denn alles Neue ist auch erstmal interessant.

Möglichkeiten, den sogenannten „Interessantheitsgrad“ (Entertainment Factor) eines Spiels zu messen, zeigen Yannakakis und Hallam in ihren Papern [YH04, YH05]. Einen weiteren Ansatz, den Entertainment Faktor auf nicht wissenschaftliche Weise zu messen, findet sich in den diversen Computer-Spiele-Zeitschriften. Hier habe ich mir von einigen Redakteuren die Bewertungskriterien ihrer Zeitschriften erklären lassen. Die Ergebnisse finden sich am Ende dieser Ausarbeitung (Kapitel J.7).

Nach einer kurzen allgemeinen Einleitung (Kapitel J.2) erläutere ich das Beispielspiel Pacman und beschreibe kurz unterschiedliche Spielstrategien innerhalb dieses Spiels (Kapitel J.3). In den folgenden Kapiteln erkläre ich Kriterien, die ein Spiel interessant machen (Kapitel J.4 und J.6) und anschließend, wie man solche Kriterien in ein Interessantheitsmaß umsetzen kann (Kapitel J.5), an Hand dessen Spiele miteinander verglichen werden können, ohne dass eine subjektive Meinung eines Testers Einfluß nimmt (Kapitel J.7).

Es entsteht ein Interessantheitsmaß, mit dem unsere PG den Erfolg der entwickelten Strategien messen und verifizieren kann. Auf jeden Fall wird das entstehende Maß insoweit flexibel gehalten, dass an Hand von Parametern die einzelnen Faktoren, die ein Spiel interessant machen (vgl. Kapitel J.4), unterschiedlich gewichtet werden können.

J.2 Fachfremde Verwendung wichtiger Begriffe

J.2.1 Definition von Spiel

Die allgemeine Beschreibung eines Spiels zeigt viele verschiedene Aspekte auf, zusammenfassend lassen sich folgende Eigenschaften finden:

Ausschlaggebend für ein Spiel ist, dass es nicht aus existenziellen Gründen oder zwanghaft durchgeführt wird, sondern zum Zeitvertreib, zur Entspannung oder zum persönlichen Vergnügen. Für Spiele mit mehreren Spielern müssen gemeinsame Regeln festgelegt werden, um einen Ablauf zu gewährleisten.

J.2.2 Definition von Spaß

Spaß lässt sich nicht einfach mit wenigen Worten beschreiben. Hier fließen unterschiedliche Begriffe zusammen, die Elmar Hillel alphabetisch aufzählt [Hil07]:

> „Spaß besteht aus Elementen wie: Ausgelassenheit, Entzücken, Ekstase, Freude, Fröhlichkeit, Frohlocken, Frohsinn, Gekicher, Gelächter, Geselligkeit, Glück, Heiterkeit, Herumalbern, Hingerissensein, Jubel, Lebensmut, Lebhaftigkeit, Scherzen, Schwung, Seelenruhe, Sorglosigkeit, Spiel, Spott, Sticheleien, Stimmung, Ulk, Unbeschwertheit, Unterhaltung, Witzen, Witzigkeit und Wonne.“

J.2.3 Definition von Spielspaß

Spielspaß an sich läßt sich nun mit Kombinationen der oben angeführten Beschreibungen definieren. Interessant ist abschließend allerdings noch die Frage, warum wir auch im erwachsenen Alter noch gerne spielen. Die Ursache ist für Peter Vorderer [Kle06] in der„ ... evolutionsbiologische(n) Sichtweise: Der frühe Mensch habe durch die Fähigkeit zum Spielen, zum Entdecken und Ausprobieren neuer Möglichkeiten, einen Vorteil im Überlebenskampf der Arten gehabt. Für den zivilisierten Menschen sei der verbleibende Spieldrang etwas, 'das wir nicht brauchen, das die Evolution aber überstanden hat“. „

J.3 Das Beispielspiel Pacman

Das Interessantheitsmaß ist immer von einem bestimmten Genre abhängig. Da das Ziel unserer PG primär das Spiel Pacman ist, wird im Folgenden ein kurzer Einblick in Pacman gegeben:

J.3.1 Pacman

Pacman wurde 1980 von Namco in Japan entwickelt und gehört in das Genre der Räuber-Beute-Spiele (engl. predator/prey). Ziel des Spielers ist es, Pacman (einen kleinen gelben Kreis) durch ein Labyrinth zu führen und alle Pellets (Futterpillen) einzusammeln. Vier Geister wollen ihn am Erfolg hindern, indem sie ihn durch Berühren töten können. Hat

Pacman alle Pellets gefressen, so kommt er in das nächste Level, das sich nur in der etwas höheren Geschwindigkeit von dem vorherigen unterscheidet. Als Besonderheit existieren in jedem Level noch Bonuspunkte, die Pacman einsammeln kann, um die Punktzahl zu erhöhen und vier Kraftpillen, mit denen Pacman für kurze Zeit die Geister fressen und so weitere Bonuspunkte kassieren kann. Das Spiel ist beendet, wenn Pacman keine Leben mehr besitzt, also oft genug von den Geistern getötet wurde.

J.3.2 Strategien der Spielcharaktere

Um die Auswirkungen der Änderungen an der KI messen und bewerten zu können, müssen die unterschiedlichen Strategien der Spielcharaktere automatisiert werden. Es kann keinem menschlichen Spieler zugemutet werden, Tausende von Spielen durchzuführen, um die Veränderungen zu bewerten (vgl. Kapitel 5 in [TDNL06]). Von Spielspaß wäre in diesem Fall nicht mehr zu sprechen. Im Spiel treffen folgende zwei Charaktere mit gegensätzlichen Interessen aufeinander, wofür Yannakakis und Hallam in ihrem Paper [YH05] folgende verschiedene Spielstrategien diskutieren:

Pacman Pacman kann verschiedene Strategien unterschiedlicher Komplexität anwenden, um das Level siegreich zu beenden.

1. *Ein-Feld vorausschauend*: Pacman entscheidet nur über seinen nächsten Zug, indem er schaut, ob in seiner direkten Nachbarschaft ein Feld mit Pellet ist, welches dann den Vorrang vor einem leeren Feld bekommt. Felder, die sich in direkter Nachbarschaft zu einem Geist befinden oder gar von einem Geist besetzt sind, werden vermieden.

2. *Pellet suchend*: Pacman verhält sich wie der Ein-Feld vorausschauende Pacman. Allerdings zieht er, wenn alle Felder um ihn herum leer sind, immer zum nächstliegenden Pellet.

3. *Geister vermeidend*: Pacman verhält sich wie der Ein-Feld vorausschauende Pacman. Allerdings zieht er, wenn alle Felder um ihn herum leer sind, immer weg von für ihn sichtbare Gegner. Wenn keine Gegner sichtbar sind, verhält er sich wie der Pellet suchende Pacman.

Allerdings kann mitels dieser Strategien nicht das menschliche Verhalten nachgeahmt werden, da wir nicht immer optimal handeln, sondern aus z.B. ästhetischen Gründen nicht dem langen Pelletweg folgen, sondern abweichen um zwei einzelne Pellets zu fressen, damit dann ein Gebiet komplett „gereinigt“ ist. Manchmal haben wir auch gar nicht die Fähigkeit eine Aufgabenstellung insgesamt zu überblicken. In diesem Fall handeln wir zwar dem eigenen Empfinden nach optimal. Pauschal gesehen gibt es allerdings bessere Lösungen.

Die Geister Da sich die Geister nicht schneller als Pacman bewegen können und es im Labyrinth keine Sackgassen gibt, ist es einem einzigen Geist normalerweise nicht möglich Pacman zu töten. Ausnahmen bildet ein Pacman, der direkt auf den Geist zuläuft, bzw. auf den Geist wartet. Auch für die Geister lassen sich verschiedene Strategien unterschiedlicher Komplexität konstruieren.

1. *zufällig bewegend*: Der nächste Zug des Geistes hängt nur von einer Zufallsvariablen ab. Jede mögliche Richtung ist gleichwahrscheinlich.

2. *verfolgend*: Der Geist versucht Pacman zu verfolgen und den relativen Abstand in beiden Richtungen zu minimieren.

3. *Abstand verringernd*: Der Geist verwendet zum Einen die Strategie des verfolgenden Geistes. Allerdings berücksichtigt er zusätzlich die Position des ihm nächsten Geistes. Ziel dieser Strategie ist es, nicht mit zwei oder sogar mehreren Geistern auf einem „Haufen“ Pacman zu jagen, sondern sich bei der Jagd besser zu verteilen und so die Wahrscheinlichkeit zu erhöhen, Pacman einzukreisen und zu töten.

J.4 Was macht ein Computerspiel interessant?

In dem Paper [YH05] werden drei Kriterien vorgestellt, die ein Computerspiel für den Benutzer attraktiv machen. Diese Kriterien sind:

J.4.1 Ein Spiel darf weder zu leicht noch zu schwer sein.

Wenn ein Spiel zu leicht ist, dann hat der Benutzer zwar am Anfang viel Spaß am Spiel, da er schnell Erfolge erzielt. Allerdings kommt bei zu starker Monotonie schnell Langeweile auf, da sich ihm keine Herausforderungen stellen. Ist ein Spiel allerdings zu schwer, so stellen sich bei dem Benutzer keine Erfolge ein und er verliert ebenfalls schnell den Spielspaß. Das Ziel bei der Entwicklung eines guten Computergegners muss also darauf liegen, dass die KI so eingestellt ist, dass sie den Benutzer manchmal besiegt und manchmal vom Benutzer besiegt wird. Hier entsteht aber genau das Problem herkömmlicher Computergegner, denn während der Benutzer seine Fähigkeiten mit der zunehmenden Anzahl an gespielten Spielen verbessert, bleiben Computergegner auf ihrer Stärkenebene stehen.

> Im Fall des bei unserer PG zu Grunde gelegten Spiels Pacman heißt das: Die Geister müssen Pacman mal besiegen und manchmal auch Pacman siegen lassen.

J.4.2 Der Computergegner muss im Verlauf mehrerer Spiele unterschiedliche Siegstrategien benutzen.

Ein Computergegner sollte möglichst viele verschiedene Strategien besitzen, um den Spieler zu besiegen. Denn je mehr Abwechselung dem Benutzer geboten wird, um so weniger Langeweile kann bei ihm auftreten. Zweitens ist der Computergegner für den Benutzer nicht so leicht durchschaubar und verlangt deshalb vom Benutzer neue Lösungsstrategien zu entwickeln.

> Bei Pacman heißt das: Die Geister sollen unterschiedliche Jagdstrategien besitzen, beispielsweise Pacman umzingeln oder ihn nur verfolgen.

J.4.3 Der Computergegner soll nicht nur statisch auf sein „Opfer“ lauern sondern sich im Spiel aktiv bewegen.

Ein lauernder Gegner ist für einen Benutzer viel einfacher vorherzusehen und daher einfacher zu besiegen als ein Computergegner, der statistisch ungefähr gleichverteilt auf jedem möglichen Feld des Spiels auftaucht.

> Bei Pacman heißt das: Die Geister sollen möglichst alle Felder des Spielfeldes mit der gleichen Häufigkeit besuchen.

J.5 Generierung eines Maßes zur Berechnung des Interessantheitsgrades

Im vorhergehenden Kapitel J.4 wurden Kriterien aufgezeigt, die ein Computerspiel interessant machen. Diese Faktoren werden nun in eine Formel gepackt, um dann ein Maß zu generieren, das den Interessantheitsgrad eines zu Grunde gelegten Computerspiels berechnen kann. Yannakakis und Hallam schlagen in dem obengenannten Paper [YH05] untenstehende Formeln vor. Die Werte der drei Teilformeln sind normiert auf das Intervall $[0, 1]$, um so leicht zu einer gleichgewichteten Gesamtformel zusammengefügt werden zu können. 0 bedeutet hierbei, dass eine Teilformel das Kriterium gar nicht erfüllt und 1 analog, dass das Kriterium optimal erfüllt wird.

J.5.1 Ein Spiel darf weder zu leicht noch zu schwer sein.

Hier wird also eine Formel gesucht, die gegen Null geht, falls der Benutzer fast jedes Spiel oder fast kein Spiel gewinnt. Das Spiel muss also über eine bestimmte Zeit betrachtet werden und die einzelnen Spiele müssen bewertet werden. Teilweise kann ein detaillierterer Wert berechnet werden, wenn nicht nur „gewonnen oder verloren“ betrachtet wird, sondern auch die Länge eines Spiels (z.B. über die Dauer bis zum Sieg / zur Niederlage oder die Anzahl der Schritte bis zum Sieg / zur Niederlage).

In Bezug auf das Beispielspiel Pacman stellen Yannakakis und Hallam folgende Formel zur Berechnung des Schwierigkeitsgrads auf:

$$T = [1 - \frac{E\{t_k\}}{max\{t_k\}}]^{p_1} \tag{20}$$

Hierbei ist unter t_k die Anzahl der Schritte zu verstehen, die ausgeführt werden, bis die Geister den Pacman im Spiel k töten. $E\{t_k\}$ ist demnach der durchschnittliche Wert, der über eine festgelegte Anzahl an Spielen (N) gemessen wurde. $max\{t_k\}$ ist analog der größte Wert, der über alle N Spiele gemessen wurde. k läuft also von 1 bis N. Der Exponent p_1 ist ein reiner Gewichtsparameter: Ist $p_1 > 1$ so geht T weiter gegen 0, andererseits kann man mit $0 < p_1 < 1$ die Funktion näher an 1 ziehen.

Bei einer Betrachtung der Funktion T stellt man fest, dass sie die gewünschte Eigenschaft aufweist. Denn wenn die Anzahl der Schritte bis zur Tötung des Pacman stark variiert, ist $\frac{E\{t_k\}}{max\{t_k\}} << 1$ und somit $T \neq 0$.

Ist das Spiel hingegen zu schwer, so ist $max\{t_k\}$ relativ klein und $T \to 0$. Gleichfalls gilt $T \to 0$, wenn die Computergegner zu leicht zu besiegen sind, denn in diesem Fall ist $E\{t_k\}$ relativ groß und es gilt: $\frac{E\{t_k\}}{max\{t_k\}} \approx 1$.

Auch $T \in [0,1]$ ist erfüllt. Da $E\{t_k\} \leq max\{t_k\}$ und beide Werte positiv sind, gilt $0 < \frac{E\{t_k\}}{max\{t_k\}} \leq 1$

J.5.2 Der Computergegner muss im Verlauf mehrerer Spiele unterschiedliche Siegstrategien benutzen.

Eine Bewertung des zweiten Kriteriums lässt sich mit der Funktion S vornehmen:

$$S = (\frac{\sigma}{\sigma_{max}})^{p_2} \tag{21}$$

Hierbei sei σ die Standardabweichung von t_k über die Gesamtanzahl von N Spielen. σ_{max} ist wie folgt definiert:

$$\sigma_{max} = \frac{1}{2}\sqrt{\frac{N}{(N-1)}}(t_{max} - t_{min}) \tag{22}$$

Hierbei ist N wie oben definiert die Anzahl an betrachteten Spielen. t_{max} ist die Anzahl an Schritten, die ein Pacman mindestens machen muss, um das Spiel siegreich zu beenden (d.h. um alle Pellets zu essen und den Geistern auszuweichen). t_{min} ist die Anzahl an Schritten, die Geister mindestens zurücklegen müssen, um Pacman zu töten. Es gilt also $t_{min} \leq t_k \leq t_{max}$. Der Parameter p_2 dient wieder der Gewichtung analog zu p_1 (s.o.).

Bei einer Betrachtung der Funktion S stellt man fest, dass sie die gewünschte Eigenschaft aufweist. Denn je größer die Standardabweichung an Schritten ist, um Pacman zu töten, desto interessanter ist das Spiel für den Benutzer. Große Unterschiede in der Länge eines Spiels entsprechen den Unterschieden in der Strategie der Geister.

Auch $S \in [0,1]$ ist erfüllt. Da $\sigma \leq \sigma_{max}$ und beide Werte positiv sind, gilt $0 < \frac{\sigma}{\sigma_{max}} \leq 1$

J.5.3 Der Computergegner soll nicht nur statisch auf sein „Opfer“ lauern sondern sich im Spiel aktiv bewegen.

Hierbei muss die Anzahl der Besuche jeder einzelnen Zelle des Spielfelds durch die Geister zu Grunde gelegt werden, denn die gleichmäßige Abdeckung des gesamten Spielfeldes durch die Geister soll belohnt werden. In v_{ik} wird die Anzahl der Geisterbesuche der Zelle i im Spiel k gespeichert. Yannakakis und Hallam schlagen folgende Entropie-Berechnung vor:

$$H_n = [-\frac{1}{\log V_n}\sum_i \frac{v_{in}}{V_n}\log(\frac{v_{in}}{V_n})]^{p_3} \tag{23}$$

Hierbei ist V_n die Gesamtanzahl von Besuchen aller Zellen ($V_n = \sum_i v_{in}$), also die Anzahl der insgesamt durchgeführten Schritte multipliziert mit der Anzahl an Geistern, da diese ja pro Schritt eine andere Zelle besuchen. Der Parameter p_3 dient wieder der Gewichtung analog zu p_1 (s.o.).

Die auf den ersten Blick „fachfremde“ Verwendung der Entropie-Formel zur Bestimmung des mittleren Informationsgehaltes ist in diesem Fall ein gutes Mittel die gleichmäßige Abdeckung des Spielfeldes durch die Geister zu berechnen. Bei Betrachtung der Funktion H_n stellt man fest, dass sie die gewünschte Eigenschaft aufweist, denn die Funktion ist 0 (Bedeutung beim Informationsgehalt: *keine neue Information, da die Zelle schon vorher*

bekannt), falls nur eine Zelle besucht wird und 1 (Bedeutung beim Informationsgehalt: *jede weitere Zelle gibt neue Information*), falls alle Zellen gleich oft von den Geistern besucht werden.

Aus den gegebenen normierten Entropiewerten H_n aller N Spiele wird nun der Durchschnittswert $E\{H_n\}$ gebildet, um das dritte Kriterium abzudecken.

Auch $H_n \in [0,1]$ ist erfüllt. Mit Hilfe des Vorfaktors $-\frac{1}{\log V_n}$ wird die Entropie auf das gewünschte Intervall normiert. Der Erwartungswert von Werten des Intervalls $[0,1]$ befindet sich ebenfalls im Intervall $[0,1]$.

J.5.4 Generierung des Interessantheitsgrads

Jetzt müssen nur noch die Einzelergebnisse zusammengefügt werden.

Die Werte aller drei oben aufgeführten Berechnungen liegen im Intervall $[0,1]$. Sie können nun linear wie folgt kombiniert werden:

$$I = \frac{\gamma T + \delta S + \epsilon E\{H_n\}}{\gamma + \delta + \epsilon} \tag{24}$$

I ist somit der Interessantheitsgrad des Spiels. γ, δ und ϵ sind wieder Gewichtsparameter, mit denen der Einfluß von jedem einzelnen der drei Kriterien auf das Ergebnis gesteuert werden kann. Auch I befindet sich im Intervall $[0,1]$, wobei ein größerer Wert von I einen höheren Interessantheitsgrad des Spiels aufzeigt.

J.6 Weitere Spielcharakteristika in anderen Genres

In anderen Genres haben verschiedene Eigenschaften ebenfalls Einfluß auf den Interessantheitsgrad:

J.6.1 Mehrspieler-Spiele

In dem Paper [BH06] beschreiben die Autoren ein Spiel, bei dem mehrere Spieler im Wettbewerb stehen und möglichst viel Gewinn machen wollen. Der Interessantheitsgrad des Spiels ist bei diesem Spiel genau dann am größten, wenn der in Führung liegende Spieler oft wechselt. Zusätzlich hat auch der Abstand der Punktzahl zwischen dem Gewinner und dem Letzten des Spiels einen Einfluß auf den Spielspaß, denn je geringer die Differenz ist, um so knapper war der Spielausgang und um so interessanter war das Spiel.

J.6.2 Teamsportspiele

In dem Paper [TB06] beschreiben die Autoren ein Fussballspiel für einen Spieler. Einfluß auf den Interessantheitsgrad des Spiels hat hier die „Intelligenz“ der Mitspieler des Spielers. Ausschlaggebend für eine gute Bewertung des Spielspaßes ist die Adaptivität der Mitspieler zu der Spielstrategie des Spielers. Wenn die Mitspieler zu passiv sind, schränken sie die Möglichkeiten des Spielers ein, stören seine Strategie und verderben somit den Spaß.

J.6.3 Spiele für Kinder

Untersuchungen in [YLH04] zeigen, dass gerade bei Spielen für Kinder ein „Phantasie"-Faktor nicht unberücksichtigt werden darf. Hierbei ist es ausschlaggebend inwieweit Kinder aus den ggf. abstrakten Spielumgebungen sich eine Phantasiewelt aufbauen können, in der es vieles zu entdecken gilt. Je interessanter und abwechslungsreicher eine Story aufgebaut wird, desto größerer und länger anhaltender Spielspaß zeigte sich bei den jungen Testern.

J.7 Die Bewertung in Computer-Spiele-Zeitungen

Außerhalb der wissenschaftlichen Sicht an Hand der hergeleiteten Formel im vorhergehenden Kapitel J.5 werden auch in anderen Bereichen Bewertungen des Spielspaßes benötigt. So müssen Spielentwickler die von ihnen entworfenen Produkte verifizieren und viele verschiedene Zeitschriften vergeben Noten und erstellen Ranglisten, um die neu erschienenen Spiele einzuordnen. Ich habe versucht, mit einigen Spielentwicklern und Spielezeitschriften Kontakt aufzunehmen, um ihre Bewertungsstrategien zu erfahren.

Folgende Ergebnisse entstanden:

Die Redakteure von *PC Player forever* und *GameCaptain* erklärten mir sehr direkt, dass ihr Bewertungssystem subjektiv ist. Bei beiden Zeitschriften vergeben die Tester nach ihrem eigenen Eindruck die Punkte für das untersuchte Spiel. Während bei *PC Player forever* der Gesamteindruck bewertet wird, untersucht *GameCaptain* wenigstens 5 Unterkategorien, die jeweils 1-20 Punkte bringen können. In der interessanten Unterkategorie Spielspaß stellen sich dem Bewerter folgende zu beurteilende Fragen:

1. Macht ein Spiel lange Spaß?
2. Macht ein Spiel immer wieder Spaß?
3. Macht ein Spiel auch/nur zu mehreren Spielern Spaß?
4. Macht ein Spiel Augen und Ohren Spaß?
5. Macht ein Spiel dem Gehirn (Story, Anspruch) Spaß?

> „Letzten Endes ist die Gesamtwertung auch immer ein in Bezug setzen zu anderen Spielen. Das Spiel finde ich besser als Spiel X aber schlechter als Spiel Y, also kriegt es eine Wertung dazwischen."
>
> *Jörg Benne - Redaktion GameCaptain*

Auch *SFT* verweist auf ein ähnliches Bewertungssystem wie die beiden oben angeführten Zeitschriften. Allerdings wehrt man sich hier gegen meinen Vorwurf mittels Notenvergabe subjektive Ergebnisse zu erzeugen, indem die Redaktion darauf besteht, dass durch die 10 eingebundenen Tester ein *„objektiver Mittelwert"* entsteht. Zusätzlich erklärt *SFT*, dass man wegen der jahrelangen Erfahrung nun auch gute Referenzlisten hat, mit deren Hilfe man die Spiele jetzt sehr leicht einordnen kann. Interessant bleibt aber folgende Aussage:

> „Spielspaß in Magazinen ist eine absichtlich schwammig gehaltene Formulierung. Der Spielspaß versucht einzuschätzen, wie viel Spaß eine bestimmte Person bei einem bestimmten Spiel haben wird. Der Spielspaß lässt sich nicht errechnen. Er macht auch keine Aussage über die Qualität der einzelnen Bestandteile (Optik, Sound, usw.), obwohl diese Qualitätsmerkmale in der Regel bei Spielen mit hohem Spielspaß zumindest dem jeweiligen, aktuellen Standard des Systems (PC, Konsole, Handheld) entsprechen.“
>
> *Alexander Geltenpoth - Redakteur Ressort Spiele SFT*

Zusammenfassend läßt sich hier allerdings feststellen, dass keine Zeitschriftenredaktion ein Maß zur Berechnung des Spielspaßes besitzt, sondern hier nur der subjektive Eindruck der (erfahrenen) Tester ausschlaggebend für die Bewertung ist. Eine Objektivität wäre aus meiner Sicht nur dann gegeben, wenn man eine (genrespezifische) Formel benutzen würde, um die Spiele vergleichen zu können.

Für andere Faktoren eines „guten Spiels“, wie beispielsweise Sound und Grafik existieren teilweise Formeln, die eine gute Vergleichbarkeit der Spiele ermöglichen. An diesem Aspekt spiegelt sich also die allgemeine rückständige Behandlung und Bewertung einer guten KI wieder, auf die ich schon in der Einleitung hingewiesen habe (vgl. Kapitel J.1).

J.8 Zusammenfassung

Der speziell für das Spiel Pacman entwickelte Interessantheitsgrad lässt sich leicht auf andere Spiele adaptieren. Hierzu müssen analog zu dem oben gezeigten Vorgehen zuerst einmal die Kriterien extrahiert werden, die dieses spezielle Spiel interessant machen. Möglichkeiten für diese Kriterien wurden in Kapitel J.6 diskutiert. Anschließend muss hierfür eine Formel aufgestellt werden, die die gewünschten Eigenschaften aufweist. Eine Normierung auf das Intervall $[0, 1]$ ermöglicht, dann das Zusammenfassen einzelner Aspekte zum Entertainmentfaktor.

Die ersten beiden Kriterien von Pacman sind grundsätzlich bei jedem Computerspiel ausschlaggebend für den Interessantheitsgrad. Ein Computerspiel lebt vom richtigen Schwierigkeitsgrad und der Variabilität der Computergegner. Der dritte Punkt, das Ausnutzen des gesamten Spielfeldes, ist charakteristisch für die sogenannten Räuber-Beute (prey/predator) Spiele, zu denen Pacman gehört. Bei Spielen, die zu anderen Genres gehören, kann man den Faktor unberücksichtigt lassen (setze $\epsilon = 0$ in Formel 24) oder durch andere spielspezifische Kriterien ersetzen.

Natürlich machen auch Faktoren jenseits der KI ein Spiel interessant. Besonders eine gute Graphik oder Toneffekte fesseln den Benutzer eines neuen Spiels. Doch gerade diese Faktoren, in die zur Zeit die meiste Entwicklungsarbeit gesteckt wird und die in vielen Spielemagazinen (vgl. Kapitel J.7) die Bewertung eines Spiels überproportional beeinflußt, nutzen sich relativ schnell ab. Den Interessantheitsgrad über eine große Zeitspanne hochzuhalten geht nur mittels variabler Computergegner.

Insbesondere muss es aber immer das Ziel sein, Computergegner so zu generieren, dass sie menschliche Verhaltensweisen nachahmen. An dem stetig wachsenden On-line-Spielinteresse kann man erkennen, dass Benutzer lieber realistische Gegner haben möchten,

als durchschaubare deterministische Computergegner. Mit Hilfe der Formel 24 für den Interessantheitsgrad kann/muss nun versucht werden, die Computergegner an das menschliche Verhalten anzupassen.

J.9 Vielen Dank

Ich möchte Georgios N. Yannakakis und John Hallam danken, dass sie so schnell auf meine Anfragen geantwortet und mir ihre Ergebnisse zur Verfügung gestellt haben. Nicht so positiv war die Resonanz bei den von mir angeschriebenen Spielentwicklern und Spielzeitschriften. Von 12 Mails an die Computerspielefirmen kam keine einzige Antwort zurück. Ebenso wenige Antworten bekam ich von den 10 internationalen Spielezeitschriften. Von 18 angeschriebenen deutschsprachigen Zeitschriften antworteten mir immerhin 3 Magazine, die somit auch alle von mir in obigem Kapitel J.7 erwähnt wurden.

K Module von intelligenten Systemen in Spielen

K.1 Einführung

Wenn man eine KI designen möchte, muss man sich darüber im Klaren sein, wie diese KI aussehen soll und welche Aktionen bzw. Reaktionen man von dieser KI erwartet. Eine mögliche Endzielsetzung wäre zum Beispiel der Endsieg, also das Erreichen bestimmter Sub-Siege, die in der Summe zum Gesamtsieg beitragen. Dafür soll die KI die Ressourcen, die ihr zur Verfügung stehen, in einer koordinierten und anspruchsvollen Art und Weise kontrollieren. Außerdem sollte die KI in der Lage sein strategisch und taktisch zu agieren. Gleichzeitig sollte sie dabei aber möglichst flexibel sein, damit die Routinen modular bleiben und das Expandieren von KI später leichter wird. Das Design von KI ist zwar ein iterativer Prozess, die Handhabung sollte aber realistisch sein. Zusätzlich sollte dabei natürlich der Spielspaß für den Player nicht außer Acht gelassen werden.

Im Folgenden werden zunächst sieben Levels, die für die KI-Entwicklung wichtig sind, vorgestellt. Es ist zu beachten, dass die einzelnen Levels aufeinander aufbauen.

K.2 Erzeugung einer KI in sieben Levels

K.2.1 Level 1 - Einheiten bewegen (FSM) und Gruppierung

Die KI soll die Steuerung von den Basisobjekten im Spiel übernehmen. Die zentralen Basisobjekte im Spiel (zum Beispiel Strategie) sind die Einheiten. Die Aufgabe der KI ist also, die „sinnvolle" Einsetzung ihrer Einheiten (bzw. Ressourcen). Die KI soll also viele „kleine" Probleme lösen können, die in der Summe die Lösung des gesamten Problemes darstellen. Sie muss wissen, was mit den Einheiten passieren soll und wie diese eingesetzt werden sollen. Dafür soll die KI die möglichen Auswirkungen ihres Handels analysieren und abschätzen können, aus denen sie eine Art Prognose für die zukünftige Lage herleiten kann.

Zuerst muss die KI dafür sorgen, dass die Einheiten bewegt werden. Als erstes sollte man sich überlegen, welche Art von Aktionen und Reaktionen die Einheiten im späteren Spiel haben sollen. Oder man sollte sich selber fragen: „Was würde ich an dieser oder jener Stelle tun?". Dieses Verhalten wird daraufhin auf die Einheiten der KI übertragen[26]. Dieses würde eventuell dazu führen, dass das Verhalten für den Spieler durchschaubarer würde, d.h. man könnte sich zum Beispiel prozentuale Abweichungen bei den möglichen Entscheidungen für die jeweiligen Einheiten in den gegeben Situationen überlegen. Diese zu treffende Entscheidung, wie die Einheiten im Spiel agieren sollen, ist entscheidend für den Spielspaß des späteren Spielers und sollte gut durchdacht werden. Nachdem man sich für Aktionen und Reaktionen der Einheiten entschieden hat, braucht man einen „Zustands-Automaten", der die getroffenen Entscheidungen steuern sollte[27]. Hierfür kann man zum Beispiel die Finite State Maschinen nutzen. Diese Maschinen gehen von einer limitierten Anzahl von Zuständen und Operationen für diese Einheiten aus. Die Maschine entscheidet also nach festprogrammierten Entscheidungsmöglichkeiten (im Unterschied zu dem

[26] `http://www.gamedev.net/reference/articles/article784.asp` A Practical Guide to Building a Complete Game AI: Volume 1 by Geoff Howland

[27] `http://www.gamedev.net/reference/articles/article1351.asp` Fun Games vs. Realistic Games

Player, der eine freie Entscheidungsmöglichkeit besitzt) [Sch94]. Mögliche Zustände für eine Einheit können zum Beispiel stehen (initialer Zustand), patrouillieren oder verfolgen sein. Die jeweiligen Zustandsübergangsbedingungen können dabei zum Beispiel Gegner gesehen = True sein. Das Endsystem und damit auch die FSM sollen möglichst einfach und übersichtlich gehalten werden, falls dies möglich ist.

Es können zwei Arten von FSM eingesetzt werden. Eine für das Game Interface, also ob sich das Spiel zum Beispiel im Pausenmodus (das Spiel wurde durch den Player pausiert) befindet, oder welche Schwierigkeitsstufe für das Spiel ausgesucht wurde. Hier wird auch bestimmt, was der Player sehen kann und was nicht. Die andere Art der FSM beschreibt, was im Spiel tatsächlich passiert, welchen Status die Objekte haben, ob zum Beispiel die Missionen, in denen sich die Objekte (Einheiten) befinden, erledigt wurden oder nicht und alle sonstigen Variablen, die für das Herausfordern des Players benutzt werden[28].

Außerdem sollte man sich überlegen, ob die Einheiten in Gruppen agieren sollen, oder als einzelne Einheiten fungieren. Der Trend der heutigen Spiele geht klar zu der Gruppierung (zum Beispiel Schlacht um Mittelerde 2, Age of Empire 3 oder Command and Conquer 3). [29] Weiterhin ist es nicht nur für die von dem Spieler zu sehenden Umgebungen sinnvoll Gruppierungen von Einheiten vorzunehmen, auch für die Betrachtung der KI könnte dieses von Nutzen sein. Die Vorteile einer gruppierten Programmierung von Einheiten liegen auf der Hand: In der Gruppe können die einzelnen Einheiten koordinierter handeln, die Formationsbewegung erfolgt nur durch einen Zugriff auf die Master-Liste von Bewegungs-Infos, Multi-Einheiten Bewegungen, wie zum Beispiel das Umkreisen von Gebäuden, sind mit eine Gruppe einfacher als mit einzelnen Einheiten. Außerdem behalten die Gruppen ihre Struktur, was für den Erhalten von neuen Instruktionen die gleiche konstante Zeit bedeutet und insgesamt schneller ist, als wenn man die Instruktionen zu jeder einzelnen Einheit durchstellen müsste. Wenn eine Gruppe also bis auf eine Einheit eliminiert wurde, besteht eben diese Gruppe aus nur einer Einheit und die einzige Möglichkeit der Elimination dieser Gruppe ist das Töten der eben genannten Einheit. Durch diese Methode ist der Code leichter zu lesen und zu verstehen.

Das Problem an diesem ersten Level ist, dass die Bewegungen und die Aktionen, die durch die Einheiten ausgeführt werden, nicht direkt zu einem Sieg führen. Diese Art der „noch nicht vorhandenen KI“ würde den Player zwar verwirren, aber keine ernsthaften Probleme bereiten.

K.2.2 Level 2 - Bewegungsfilter

Auf diesem Level sollte man die Aktionen der KI auch von dem Player (Gegner) abhängig machen. Denn was bringt die beste Strategie, wenn man nicht weiß, was das Gegenüber macht, sodass man seine eigenen Handlungen nicht auf die des Players anpassen kann. Natürlich sollte man schon hier versuchen, die nächsten Schritte und Aktionen des Players so weit wie möglich vorherzusehen. Dieses hätte zur Folge, dass man seine eigene Strategie oder Taktik abstimmt und anpasst.

Durch die Einbeziehung des Players sollte das Endziel natürlich beachtet werden, nämlich der Sieg über den Player. Dabei sollte man sich klar machen, durch welche Aktionen

[28] `http://www.3sat.de/neues/sendungen/magazin/99588/index.html`

[29] `http://www.gamedev.net/reference/articles/article545.asp` Project AI by Mark Lewis Baldwin and Bob Rakosky

der eigenen Einheiten man zum Sieg kommen kann. Man sucht sich also somit in der Bewegungsliste der einzelnen Einheiten die „passenden“ Aktionen und Bewegungen aus, die zu einem Sieg führen würden.

Das Problem in diesem Level ist, dass wir das Endziel zwar vor Augen haben, aber noch nicht die Möglichkeit in unsere Betrachtungen einbezogen haben, dass es recht viele verschiedene Möglichkeiten gibt, um einen Sieg zu erringen. Mit anderen Worten haben wir noch keinen Filter, der dafür sorgen könnte, dass die besten Möglichkeiten für die einzelnen Einheiten zu einem Zeitpunkt ausgefiltert werden. Diese Filterung hätte zur Folge, dass wir zu diesem betrachteten Zeitpunkt eine optimale Entscheidung treffen könnten.

Das andere Problem auf diesem Level ist die Nichtausrechenbarkeit der Handlungen des Players. Man kann zwar versuchen durch die aktuell vorliegenden Daten eine Einschätzung für die Handlungen des Players für die Zukunft zu machen, dieses würde aber so nicht klappen, da man die Handlungen des Players nicht vorhersagen kann.

K.2.3 Level 3 - Zielfilter

Man muss eine Formel für die Sieg-Chancen entwerfen. Diese soll zu einem Zeitpunkt abwägen können, wie vielversprechend eine Aktion oder eine Bewegung in Bezug auf den Sieg ist. Es sollten dabei möglichst keine True oder False Auswertungen benutzt werden, sondern lediglich numerische Werte, um somit die beste Möglichkeit leicht erkennen und auswählen zu können.

Das heißt, wenn man einen Sieg mit zwei Schritten erreichen kann, sollte man genau diese Methode auswählen und nicht eine andere, bei der man wohlmöglich zehn Schritte dafür benötigen würde. Wir suchen also das Minimum.

Problem hier ist folgendes: Bei unserer Auswertung werden nicht die Möglichkeiten in Betracht gezogen, die zwar nicht direkt zu einem Seig beitragen. Ein Beispiel hierfür wäre, wenn man zu einem beliebigen Zeitpunkt im Spielverlauf einen Soldaten töten würde und somit schneller zum Sieg kommt, als wenn man es an dieser Stelle nicht getan hätte. Also müssen wir dafür Sorge tragen, dass auch die möglichen Eventualitäten nicht außer Acht gelassen werden.

K.2.4 Level 4 - Sub-Ziele und der Gesamtsieg

Diese eben genannten, möglichen Eventualitäten, die uns schneller zum Sieg verhelfen könnten, werden als Sub-Siege bezeichnet. Die Summe von möglichen Sub-Siegen könnte uns zum Gesamtsieg verhelfen. Somit müssen wir nun bei allen Entscheidungen unserer KI die möglichen Auswirkung der Sub-Siege ab dem gegebenen Zeitpunkt bis zu einem möglichen Sieg berücksichtigen und den besten Weg finden. Wir sind also auf der Suche nach der vielversprechendsten Reihe von Sub-Siegen. Die möglichen Sub-Siege könnten lauten: Gegner töten, Verbündeten schützen, defensive Linie bilden, oder aber auch einfach die Basis schützen. Um diese Entscheidungen zu treffen, könnte man eine Art Baum betrachten (Entscheidungsbaum), in dem die Wurzel den Endsieg bedeuten würde und die Knoten und Blätter die einzelnen Sub-Siege.[30] Die Blätter repräsentieren natürlich die

[30] `http://www.gamedev.net/reference/articles/article784.asp` A Practical Guide to Building a Complete Game AI: Volume 1 by Geoff Howland

momentane Situation, in der wir uns befinden und von der wir die möglichst besten Pfade (Bottum-Up) bis zur Wurzel suchen.

Die einzelnen Sub-Siege bedeuten nichts anderes, als dass man im Baum eine Ebene nach oben kommt. Dies kann aber auch durch das Erbringen von Opfern passieren. Man stelle sich also eine Situation vor, in der man zwei Gruppen von Einheiten hat. Die eine Gruppe ist größer und stärker, die andere ist eher kleiner und schwächer. Nehmen wir die Situation, in der man sich zum Beispiel mit der größeren Gruppe vor der Basis des Players befindet und die kleinere Gruppe sich vor einem, für den Player sehr wichtigen, von der Basis entfernten, eingenommen Gebäude, befindet. Nehmen wir jetzt an, dass sich in der Basis des Players sehr viele Einheiten befinden, sodass es sogar für unsere große Gruppe von Einheiten ein Selbstmord wäre, jetzt die Basis zu attackieren. Dies ist eine wunderbare Ausganglage für die Erbringung eines Opfers. Die bedeutet, dass man die Möglichkeiten auswertet und merkt, dass es viel wichtiger ist die Basis des Players zu zerstören, als den Verlust von der kleineren Gruppe zu vermeiden. Somit kann man also mit der kleinen Gruppe das von dem Player eingenommen Gebäude attackieren. Wenn sich die Einheiten des Players auf die Rettung des von der KI angegriffenen Gebäudes machen, attackiert die KI mit der großen Gruppe die Basis des Players.

Das Problem hier ist, dass unsere Entscheidungen unabhängig vom Gegner (Player), oder unseren Alliierten (welche im Spiel vorhanden sein sollen) getroffen werden. Unsere Aufgabe für die nächste Annährung im folgenden Level ist also, die Koordination von unseren Einheiten.

K.2.5 Level 5 - Koordination der Entscheidungen

Die einzelnen Einheiten sollen ihre Entscheidungen mit anderen Einheiten koordinieren. Auf diese Weise kann man die Entscheidungen im Entscheidungsbaum besser balancieren.

Man sollte eine Informationszentrale erschaffen, in der alle nötigen Daten von den einzelnen Einheiten schnell und einfach gefunden werden können. Diese Position sollte für die KI die logischste sein, wenn man die Daten von den Einheiten erfragen oder abrufen möchte. Diese Methode hat den Vorteil, dass die Daten nicht dupliziert in verschiedenen Orten abgelegt wären und somit auch keine Duplikate erzeugt würden, was in Bezug auf zum Beispiel die Übersicht oder den Speicherplatz von Vorteil wäre.

Das Problem auf diesem Level ist, dass man zwar die einzelnen Einheiten miteinander koordiniert, aber dadurch noch bei weitem keine Strategie im Umgang mit Ressourcen durch die KI entsteht. Außerdem ist das Koordinieren von Entscheidungen zwischen allen Einheiten zu jedem Zeitpunkt im Bezug auf die Geschwindigkeit der KI etwas zu langsam.

K.2.6 Level 6 - 1.Möglichkeit - Millitär/Geschäftswelt - Strategie-Entscheidungs-Struktur

Wie wir in den vorherigen Levels gemerkt haben, ist ein Entscheidungsbaum zwar sinnvoll, erfüllt aber nicht alle nötigen Merkmale, wie zum Beispiel den optimalen Informationsfluss, oder eine taktische und strategische Handlungsweise, die für eine gute KI wünschenswert

wären. Deswegen erzeugen wir in diesem Level eine Strategie-Entscheidungs-Struktur. Diese Struktur soll die Einheiten in koordinierter und strategischer Art und Weise kontrollieren. Solche Strukturen sind auch im echten Leben zum Beispiel beim Militär oder in der Geschäftswelt bekannt und werden auch benutzt. In diesen Strukturen gibt es verschiedene Ebenen. In den untersten Ebenen befinden sich zum Beispiel die einzelnen Einheiten, die keine anderen Informationen haben, außer ihre Postionen auf der Karte. Diese Einheiten haben wenig Informationen (Überblick) über die Umwelt, doch je weiter man sich nach oben bewegt, desto mehr Informationen (Überblick) enthalten die Ebenen. Hierbei sollte natürlich nicht außer Acht gelassen werden, dass diese Hierarchische-Struktur einen Informationsverlust mit sich trägt, der sich über die Ebenen verbreitet. Dies hat zur Folge, dass die höchste Ebene zum Beispiel ein General oder ein Chef der Firma, (bei Bedarf) die gesamten Informationen (Überblick) über die Umwelt enthält. Es entsteht eine Hierarchie. Man kann das System natürlich auch auf zwei oder mehrere Hierarchie-Strukturen erweitern.

Das Problem hier besteht nun darin, dass wenn man nur eine Hierarchie benutzt, man keine optimale Nutzung von Ressourcen erhält, da man unter Umständen sich nur mit einer Aufgabe beschäftigen würde, an der sich ganz viele Einheiten (auf der untersten Ebene) beteiligen würden. Also würde hier keine Multitasking möglich sein.

Bei der Nutzung von mehreren Hierarchien entsteht das Problem des nicht kooperativen Handels.[31] Dieses heißt, wenn man zum Beispiel zwei Einheiten aus verschiedenen Hierarchien neben einem Gebäude stehen hat und das Zerstören von diesem Gebäude einen wichtigen Sub-Sieg bedeuten würde, dann würden diese beiden Einheiten gar nicht erst zusammen fungieren, was zu suboptimalen Resultaten bei der Optimalität der KI führen würde.

Der Grund wieso man das Hierarchiemodell aus der Realität nicht ohne Einschränkungen der Optimalität in der KI nutzen kann, ist der, dass im echtem Leben eine Beschränkung der Kommunikation besteht. Es ist sehr schwer (oder erst ganz nicht möglich) dafür zu sorgen, dass jeder in der Hierarchie-Struktur zu jedem Zeitpunkt alle möglichen Informationen in Realtime bekommt, die er auch bekommen sollte.

Bei der KI aber haben wir keine Beschränkung in der Kommunikation und können somit bessere Strukturen erschaffen, welche für die bessere Kommunikation und Kontrolle sorgen können.

K.2.7 Level 6 - 2.Möglichkeit - Flexibilität durch temporäre Projekte

Hier wollen wir zwar das Militär-System für unsere Zwecke verwenden, aber es vorher flexibel für unsere Bedürfnisse machen. Dafür erzeugen wir initiativ ein Mega-Einheit-Kontroll-System (ein Projekt), um ein spezielles Ziel zu erledigen. Sobald unser Ziel erreicht ist, werden die dafür benötigten Einheiten (Ressourcen) entlassen und könnten ab diesem Augenblick für neue Projekte eingesetzt werden.

An dieser Stelle stellt sich die Frage: Wer soll unsere Projekte steuern, besetzen, oder gar Prioritäten vergeben? Dieses soll eine weitere Kontrollstruktur in Form einer Funktion zu einem gegeben Zeitpunkt berechnen.

[31] `http://www.gamedev.net/reference/articles/article784.asp` A Practical Guide to Building a Complete Game AI: Volume 1 by Geoff Howland

Jedes Projekt hat ein Ziel (Sub-Sieg) und soll anhand diesem mit einem Entscheidungsprozess für jede Einheit entscheiden, ob diese bei der Verwirklichung von diesem Projekt am besten behilflich sein kann (bzw. am nützlichsten sein würde). Die Entscheidung welche Einheiten zu einem Projekt hinzugefügt werden sollen, entscheidet die Kostenfunktion, in der die Stärken und die Schwächen von allen möglichen Einheiten im Spiel enthalten sind. Es wird also eine Einschätzung getätigt, wie hoch die Wahrscheinlichkeit ist, dass bei der Durchführung des Projekts mit vielen Gegenangriffen in diesem Bereich auf der Karte gerechnet werden muss. Dafür können zum Beispiel die aktuellen Erkundungsinformationen über die Einheiten-Verteilung des Players auf der Karte benutzt werden, was die Wahrscheinlichkeit immens erhöhen würde (sofern vorhanden). Die Formel, die die Einheiten zu den einzelnen Projekten zuordnet, soll auswerten, wie effektiv zum Beispiel diese Einheit für das Projekt und damit für das Erreichen von diesem geforderten Sub-Sieg ist. Außerdem sollte beachtet werden, wo sich diese Einheit auf der Karte befindet. Würde es Sinn machen auf eine Einheit lange Zeit bei einem hochpriorisierten Projekt zu warten, falls die Einheit sich ganz weit weg von den anderen Einheiten entfernt hat? Oder ist das Auswählen einer Alternative in der Nähe eine bessere Lösung?

Ein möglicher Typ von einem Projekt könnte zum Beispiel "Beschütze die Stadt, töte die gegnerischen Einheiten oder nehme ein Gebäude einßein. Die möglichen Spezifikationen von einem Projekt sind da schon konkreter und könnten lauten: Zerstöre die 239 Panzer Division oder nehme die Stadt New York ein.

Da es auch mehrere Projekte zu einer Zeit geben kann, sollte jedes Projekt bei seiner Erschaffung eine Priorität bekommen. Bei der Vergabe von zu vielen Prioritäten sollte man auf den Speicherplatz achten, damit wegen des Mangels am Speicherplatz kein Projekt aus dem Speicher rausfliegt.

Die zentrale Kostenfunktion sollte regelmäßig ein Update der Projekte durchführen. Dabei sollte die aktuelle Lage der Projekte, ihr Fortschritt, oder die zum Beispiel möglichen, vorhandenen Verluste, betrachtet werden. Vielleicht ist die Gegenwehr in dem Zielgebiet viel stärker als erwartet? Würde es Sinn machen Verstärkung zu rufen? Ist diese weit entfernt? Oder sind die Verluste schon zu hoch und es besteht keine Chance auf den Sub-Sieg und es wäre somit besser einen Rückzug anzutreten, um die verbliebenen Einheiten zu heilen und anderen Projekten zuzuordnen?

Vielleicht sind einige Projekte nicht mehr relevant und können gelöscht werden, oder andere sollten eine höhere Priorität bekommen. Vielleicht hat man durch die neusten Erkundungen festgestellt, dass an dem Zielobjekt, bei dem ein Sub-Sieg erzielt werden soll, sich ganz andere Arten von gegnerischen Einheiten befinden. Folglich ist die Auswahl der Einheiten für dieses Projekt nicht mehr auf dem neuesten Stand und man sollte die Einheiten erneuern, die zu diesem Projekt zugeteilt wurden.

Zusätzlich sollte man in periodischen Abständen alle Einheiten durchgehen und sie gegebenenfalls zu den aktuellen Projekten zuordnen. So würden nicht so viele Einheiten ohne Arbeit herumstehen und die Projekte würden „optimal“ besetzt. Genauso wichtig ist die Re-Konfiguration von Ressourcen nachdem ein Projekt erfolgreich oder nicht erfolgreich abgeschlossen wurde.

In dieser, für unsere Zwecke angepassten Version, vom Level 6 haben wir Flexibilität, aber auch das Erreichen von speziellen Sub-Zielen, die zu einem Gesamtsieg führen werden, zusammengeführt.

Bei der Implementierung könnten aber eventuell an der Stelle, an der man die Einheiten zu den Projekten zuteilt, Probleme auftauchen. Dieses könnte passieren, wenn eine Einheit durch eine Funktion zu einem Projekt zugeordnet wurde, sie aber zu dem nächsten Zeitpunkt zu einem anderen Projekt zugeteilt werden könnte. So würde die Einheit ständig zwischen den Projekten hin und her wandern, aber in keinster Weise zu der erfolgreichen Beendigung von genau diesen beitragen können. Eine mögliche Lösung hierfür ist die folgende: In der Formel für die Zuteilung der Einheiten zu den Projekten soll ein Wert noch gerade so sein, dass die Einheit nicht immer hin und her springt, aber gleichzeitig klein genug ist, um nicht daran gehindert zu werden in ein Projekt aufgenommen zu werden.

Bei der Erzeugung von Funktionen für die KI sollte man auf die strukturelle Organisation achten. Es macht Sinn, wenn auch begrenzt, lieber einzelne und atomare Funktionen zu benutzen. Die Funktionen sollten limitiert nur für den einen Zweck (Aufgabe) eingestellt werden und für nichts anderes.[32] Der Vorteil ist, dass dadurch, falls etwas erneut gemacht werden sollte, man bereits eine Routine in Form einer Funktion, die schon getestet wurde, besitzt und bekannte Operationen durchführt werden. Ein weiterer Vorteil ist, dass wenn man später das Programm KI debuggen möchte, mit dieser Methode leicht die Routine für diese Daten zu finden ist.

K.2.8 Level 7 - Multiplayer

In diesem letzen Level sollte man sich die Option von dem Modus Multiplayer anschauen, wobei sich hier unsere oben getätigten Überlegungen natürlich auf mehrere Gegner erweitern würden.

Für die KI würde es dann noch eine weitere Verkomplizierung in dem Entscheidungsbaum bedeuten. Denn wenn die KI gegen mehrere Player spielen soll, werden die Überlegungen von oben dann von weit mehr Gegebenheiten abhängig.

Im Spiel wäre es zum Beispiel sinnvoll, dass die Projekte, die sich um den Player, dessen Basis sich näher zu der der KI befindet, lokalisieren, eine höhere Priorität bekommen. Da die Angriffswege so kürzer sind und man aber auch wegen der geringen Distanz schneller Angriffe von Seiten dieses Players erwarten kann.

K.3 KI bei Pacman

Bei Pacman wird die KI in einer einfachen Form benutzt. Der Spieler steuert den Pacman und die KI steuert die Geister, die versuchen Pacman zu töten, indem sie einfach den gleichen Platz auf der Karte einnehmen, wie die Pacman Figur selbst [Buc05]. Im Spiel wird eine einfache Form der Finite State Maschine benutzt. Die FSM könnte so wie auf dem folgendem Bild aussehen:

Die FSM hat in diesem Fall 3 Zustände. Roam steht für das Herumwandern, Evade für das Ausweichen und Chase für das Verfolgen. Die einzelnen Zustandsübergänge werden in diesem einfachen Fall mit True und False geschaltet. Sobald der Pacman eine Power-Pille geschluckt hat, gehen die Geister von dem Zustand Roam in den Zustand Evade. Sobald der Geist sich in dem Evade Zustand befindet, startet ein interner Timer für die Zeit,

[32] `http://www.gamedev.net/reference/articles/article2034.asp` Designing Great Games by Roger E. Pedersen

Figure 9-2. Ghost finite state machine diagram

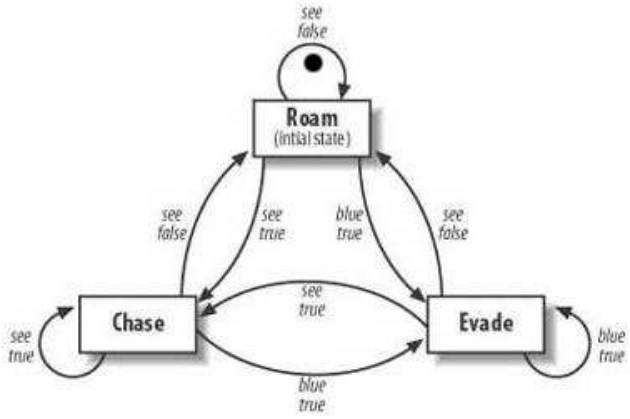


Abbildung 63: FSM für Geister

solange die Wirkung der Power-Pille noch anhält. Danach wechselt der Geist zurück in den Zustand Roam. Falls der Geist einen visuellen Kontakt mit der Pacman-Figur hat, schaltet er vom Zustand Roam in den Zustand Chase. Dieses passiert immer dann, wenn der Pacman und Geist auf demselben Wert der X oder Y Achse stehen und dabei kein Hindernis (Wand) dazwischen liegt. In diesem Zustand versucht der Geist die Pacman-Figur zu verfolgen. Die Verfolgung passiert auf folgende Art und Weise: Der Geist merkt sich die X- und die Y- Koordinaten von Pacman zu dem Zeitpunkt, an dem er ihn sieht. Dann bewegt sich der Geist auf diese Stelle und schaut von da aus, wo sich der Pacman befindet und so weiter . . . Die Verfolgung oder die Auswertung, an welcher Stelle der Geist die Pacman-Figur sehen kann, kann man leicht zum Beispiel mit einem Distanz-Vektor-Algorithmus bestimmen. Oder man verwendet die Pathfinding-Methode.

Pathfinding ist eine Methode (mit zum Beispiel dem A* Algorithmus), die dafür sorgt, dass man bei einer Eingabe von einem Start- und Zielpunkt den Weg mit den kleinsten Kosten bestimmt [PN95]. Hier ein Beispiel für die Nutzung von dem A* Algorithmus:

Der grüne Punkt (die 0) steht für die aktuelle Position der KI-Figur und der blaue Punkt (die 19) steht für das Ziel der KI-Figur. Der A*-Algorithmus vergibt Werte für alle Positionen, die die KI-Figur annehmen kann. Die möglichen Hindernisse (wie im Bsp. oben dargestellt) würden dann mit berücksichtigt und somit natürlich Extrakosten mit sich tragen.

Bei dem Entwurf der KI sollte man darauf achten, dass die Chase- Bewegungen von vier Geister möglichst nicht gleich aussehen. Die KI für die Verfolgung sollte aber gleichzeitig nicht zu schwer sein, damit der Pacman eine realistische Chance hat, zu entkommen. Dies könnte man mit der Nutzung von Wahrscheinlichkeiten regeln, indem man mit einer gewissen Wahrscheinlichkeit den optimalen und den schnellsten Weg bestimmt. D.h. der Geist würde an einer Stelle beim Chasemmit einer Wahrscheinlichkeit von zum Beispiel 60% die Verfolgung genauso nach dem A* Algorithmus von oben weiterführen. Und einer Wahrscheinlichkeit von 30% die Verfolgung mit einem anderen Pfad, der aber die höheren Kosten hat, weiterführen. Und die restlichen 9% könnten entweder wieder eine Koste-

7	6	5	6	7	8	9	10	11		19	20	21	22
6	5	4	5	6	7	8	9	10		18	19	20	21
5	4	3	4	5	6	7	8	9		17	18	19	20
4	3	2	3	4	5	6	7	8		16	17	18	19
3	2	1	2	3	4	5	6	7		15	16	17	18
2	1	0	1	2	3	4	5	6		14	15	16	17
3	2	1	2	3	4	5	6	7		13	14	15	16
4	3	2	3	4	5	6	7	8		12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15

Abbildung 64: Pathfinding mit A* Algorithmus

nerhöhung für den Pfad bedeuten, so das mit den restlicher Wahrscheinlichkeit von 1% das der Geist den Pac-Man gar nicht mehr jagt und in den Zustand „Roam“ übergeht.

K.4 KI bei NERO

Der Name des Spiels steht für Neuro-Evolving Robotic Operatives. Das Spiel wurde von den Wissenschaftlern und Studenten der University of Texas veröffentlicht. Mit diesem Spiel wird der aktuelle Stand der Forschung im Bereich KI in Spielumgebungen demonstriert. In NERO wird der Spieler zum Trainer eines Teams von KI-Soldaten, die in wechselnden Umgebungen und Situationen passende taktische Lösungen entwickeln müssen.[33] Das Spiel wurde mit der Game-Engine Torque von GarageGames realisiert. NERO passt sich den vom dem Spieler angestrebten Strategien an, während es den KI-kontrollierten Einheiten gleichzeitig erlaubt, als autonome Agenten zu agieren.

Jeder Roboter besitzt in NERO ein eigenes Gehirn. Ähnlich wie das menschliche Pendant, bestehend es aus einer Menge von simulierten Nervenzellen, die untereinander Informationen austauschen. [34] Ein solches, künstliches Gehirn bezeichnet man als neuronales Netzwerk. Grob vereinfacht verändert NERO während des Trainings die Verdrahtung der simulierten Gehirnzellen mit Hilfe eines komplexen Regelwerks. Auf diese Weise lernen die Roboter und passen ihr Verhalten der Umgebung an.

Nach einer fest definierten Zeit sterben die Roboter. NERO überprüft dann, welche von ihnen sich im Hinblick auf die vorgegebenen Ziele am geschicktesten verhalten haben. Damit ist ein genetischer Algorithmus gemeint, eine Art verstärkter Lern-Algorithmus (angelehnt an die Darwins-Theorie von der Evolution, wo der Fitteste überleben wird), der die am besten funktionierenden Agenten belohnt und diejenigen bestraft, welche am schlechtesten

[33] `http://nerogame.org/` Infos zu NERO
[34] `http://www.linux-user.de/ausgabe/2007/03/050-nero/index.html` Infos zu NERO

sind.[35] Bestrafung und Belohnung übernimmt der Spieler durch Schieberegler, der genetische Algorithmus entscheidet auf Basis des Roboter-Verhaltens, welche Hirne am besten und welche am schlechtesten geeigneten sind.

Die Software kombiniert die besten künstlichen Gehirne miteinander und pflanzt das Ergebnis einem der neugeborenen Roboter ein. Durch diese Strategie siebt NERO die weniger erfolgreichen Einheiten aus, ganz so wie es Darwin in seiner Evolutionstheorie beschreibt, der Fitteste überlebt.

Mit dem Fittesten sind natürlich die Anpassungen und damit der Fortschritt der Roboter im Spiel nach der Bewertung des Spielers gemeint. Denn der Player entscheidet durch seine „Punktevergabe“, ob die Aktionen der jeweiligen Roboter passend oder nicht passend waren.

Das in NERO für das Erlernen der KI verwendete Verfahren nennt sich „real-time NeuroEvolution of Augmenting Topologies“ und wurde maßgeblich von Ken Stanley an der Universität von Austin entwickelt. Dieses Verfahren funktioniert anders als die meisten anderen seiner Art. Es startet mit einem minimalen künstlichen neuronalen Netzwerk und die Komplexität wird nur dort hinzufügt, wo sie zur Problemlösung beiträgt. Damit werden unnötig komplexe Lösungen vermieden. Anderes als bei den bisherigen Verfahren, bei denen das ganzen Lösungsspektrum hinzugefügt hat, ohne vorher zu filtern.

K.5 Geschichte von Spielen und KI

Die ersten Spiele hatten keine KI, keine Mustererkennungen und keine FSM:

1962: Das erste Computer Spiel „Space War“(Multiplayer) das für ein PDP1 Minicomputer geschrieben wurde.

1972: Das Game „Pong“ wurde mit sehr großem Spaß und Begeisterung released.

Ab 1974 kamen die Spiele mit den ersten Mustererkennungen:

1974: Ersten Spiele mit Gegnern erscheinen. Spieler schießt auf bewegte Ziele, die sich nach einer Musterform bewegen.

1975: Das Spiel „Gun Fight“ wurde basierend auf einem Mikroprozessor veröffentlicht. Dies erlaubte eine größere Auswahl der Elemente im Spiel.

1978: Das Game „Space Invaders“. Bei diesem Spiel haben die Gegner sich nicht nur nach einem Muster bewegt, sondern konnten auch zurückschießen. Außerdem gab es im Spiel Levels, Punkte, eine einfache Steuerung und sich mit dem Spielverlauf erhöhende Schwierigkeitsstufen.

1980: „Pacman“ kam raus. Made by Namco. Einfache KI für die Geister.

[35] `http://www.golem.de/0506/38899.html` Infos zu NERO

Ab den 90er Jahren wurden die FSM in den Spielen eingesetzt:

1990: Das erste FSM Spiel mit dem Namen „Herzog Zwei“ kam auf den Markt. Bei dem Spiel wurde aber eine schlechte Version von Pathfinding benutzt.

1993: Das Spiel „Doom“ kam auf den Markt. Wurde zum Hit vor allem auf den LANs ;)

Seit dem Jahr 1996 kamen die ersten Spiel KIs mit der Nutzung von zum Beispiel neuronalen Netzen auf dem Markt.

1996: Das Spiel „Battle Cruiser 3000AD“ kam raus. Bei diesem Spiel wurde zum ersten Mal der Ansatz der neuronalen Netze für die KI benutzt.

1996: Das Spiel „Creatures“ kaum raus. Der Gameplay erlaubt den Kreaturen im Spiel das Lernen mit Hilfe des Players. So wurde zum ersten Mal die Idee des Lernens durch den Spieler eingebaut.

1997: Ein Schach-KI-Programm „Deep Blue“ besiegt den Schach Weltmeister Gary Kasparov.

1998: Half-Life kommt auf den Markt. Das Spiel scheint die beste KI zu haben, die es jemals gab, aber nicht aufgrund von irgendwelchen neuartigen Methoden, sondern aufgrund des großen Skripts und fühlte sich so an, als ob die KI intelligenter war.

2001: „Black and White“ kommt raus. Das erste Mal in der Spiel-Geschichte kann der Spieler die In-Game Kreaturen dazu bringen etwas zu lernen.

Seit 2001 bis heute: Die KIss werden aufgeteilt in zum Beispiel Roaming KI (Bewegungs-KI), Verfolgungs-KI, Ausweich-KI, Strategische-KI oder Benehmungs-KI. (Verhaltensaufteilung - d.h. es gibt nicht mehr nur gehen, stehen, laufen oder schiessen, sondern die einzelnen Aktionen können wiederrum in verschiedene Unteraktionen aufgeteilt werden) Diese Aufteilung führt dazu, dass der Gegner im Spiel auf viele verschiedene Weisen agieren kann und nicht immer auf die gleichen, mit der Zeit langweiligen, Spielzüge beschränkt ist.

2004: Das Spiel „Far Cry“ kommt auf den Markt. In dem Spiel wurden verschiedene KI-Konzepte benutzt. So wurde zum Beispiel die Benehmungs-KI mit 3 Arten von Roaming-KI noch intelligenter gemacht. Desweiteren wurden verschiedene Möglichkeiten für das Agieren von KI-Gegnern zum Teil mit einer Zufallszahl benutzt, so dass zum Beispiel in 50% der Fälle der Spieler vom Gegner verfolgt/angegriffen wird, in 30% der Fälle holt der Gegner Verstärkung für einen gemeinsamen Angriff, in 20% der Fälle weicht der Gegner aus bzw. flüchtet.

2007: „Command & Conquer 3“ erschienen. Bei diesem Spiel wird dem Player die Chance gegeben, die KI in vier verschiedenen Stufen nach Belieben einzustellen. Dieses gab es zwar schon in den früheren Spielen, aber nicht in dem Umfang der KI-Stufen.

Ende 2007: „Star Wars: The Force Unleashed“ mit neuartiger Euphoria Engine.[36]

[36] `http://www.naturalmotion.com/euphoria.htm` Infos zur Euphoria Engine

K.6 Ausblick

Die heutigen Anforderungen an die KI sind: das zielgerichtete, realistische Verhalten, Robustheit in neuen Situationen, Erlernen neuer Verhaltensweisen, Anpassen an Strategie des Spielers. Außerdem die Entwicklung eigener Persönlichkeit, Emotion und soziales Verhalten, muss über abwechslungsreiche Strategien verfügen, KI des Computergegner wird nach menschlichem Vorbild programmiert.

Bei Programmierung der KI sollte man beachten, dass mehr als ein Drittel der Entwicklungszeit (und Etats) eines Computerspiels ausschließlich für Programmierung der KI aufgewendet wird → das kann schnell mehr als ein ganzes Jahr sein. Es sollten erst die Grundelemente der KI programmiert werden, dann erst die Grafik (Landschaft, Charaktere) und der Sound. Das Verhalten der Figuren muss optisch realistisch wirken und man sollte außerdem bei der Programmierung darauf achten, eventuell ab und an Fehler zu machen, denn eine perfekte KI ist keine menschliche Simulation und die KI soll ja möglichst realistisch sein, d.h. man muss als Mensch die KI auch am Ende schlagen können.

Es wird aktuell an einer Euphoria Engine für das Spiel „Star Wars: The Force Unleashed“, das im November 2007 auf dem Markt kommen soll, gearbeitet. Hier ist ein Beispiel: „Die Euphoria Engine hat Auswirkungen auf die gegnerische Intelligenz. Die Sturmtruppen von oben auf eine Holzsäule herunterfallen. Auf dem Boden werden sie sterben. So versuchen sie sich an der Säule festzuhalten. Der Spieler kann mit Hilfe der Macht die Säule zerstören, sodass die Sturmtruppen herunterfallen.“[37] Die Euphoria Engine simuliert die Action, sodass es nicht möglich ist vorherzusagen, was genau passieren wird, egal wie oft man das Szenario durchspielt. Mit Euphoria verhalten sich künstliche Charaktere so, dass das Ergebnis jedes Mal unterschiedlich ist, was ein völlig neues Gameplay ergibt.[38]

Desweiteren will man in der Zukunft das ungeskriptete menschliche Verhalten nachahmen können, um so „menschlich“ wie möglich auf situationsbedingte Veränderungen in einer Spielszene zu reagieren.

[37] `http://www.lucasarts.com/games/theforceunleashed/gameinfo/news/summary.html` Infos zur Euphoria Engine

[38] `http://news.softpedia.com/news/Star-Wars-Force-Unleashed-Practically-on-Every-Console-47084.shtml` – Informationen zur Euphoria Engine

L Kommunikation Spiel/Spielstrategie, Spiel/Spielanalysesysteme

L.1 Abstrakt

Die Anforderungen an moderne Computerspiele wachsen stetig und stellen die Entwickler vor neue Herausforderungen. Speziell hat die immer größere Beliebtheit von Multiplayerspielen, welche vermehrt über des Internet weltweit gespielt werden, sowie die Forderung nach höheren Spielspaß durch clevere und nicht-deterministisch agierende Computergegner dazu beigetragen. Letzteres stellt die Entwicklung neuer Methoden der künstlichen Intelligenz weiter in den Vordergrund. Insgesamt ergeben sich für Spiele-Entwickler immer komplexere Aufgaben, die effizient bewältigt werden müssen.

In diesem Seminarbeitrag wird die Kommunikation in Computerspielen im Allgemeinen betrachtet und speziell auf Ansätze in Multiplayer-Spielen eingegangen. Zusätzlich werden Methoden vorgestellt, die den Kommunikationsaufwand reduzieren können.

Für die Kommunikation stehen zwei grundlegende Konzepte im Vordergrund: *Event-Driven-Behavior* und *Polling*. Beim Polling stellen Spielobjekte regelmäßig Anfragen, um relevante Spieldaten zu ermitteln. Das Event-Driven-Behavior vermeidet diesen Overhead dadurch, in dem Spielobjekte durch Ereignisse über relevante Änderungen in der Spielwelt informiert werden. Diese Methoden abstrahieren jedoch von der Spielearchitektur, so dass anschließend das Thema Architekturen - speziell für Multiplayerspiele - angesprochen wird. Die klassische Server-Client-Architektur wird gerade im Bereich von Massive Multiplayer Online Real-Time Strategy Games (MMORTS) durch clevere Peer-to-Peer-Netzwerke ersetzt, welche typische Probleme der effizienten Bandbreitennutzung bei Server-Client-Anwendungen vermeiden. Zusätzlich wird in diesem Abschnitt auf die Rolle der KI eingegangen und nicht-klassische Möglichkeiten beleuchtet, wie das Auslagern der KI an die Clients.

Ein weiterer neuer Aspekt, um Resourcen optimal zu verteilen bzw. zu reduzieren und die Kommunikation effiktiv zu gestalten, ist die Methode der *Statistical Client Prediction*. Dieser Ansatz verfolgt das Ziel, den notwendigen Informationsaustausch zu verringern, in dem das Verhalten von Spielern vorhergesagt wird. Dieser Gesichtspunkt ist auch im Kontext der künstlichen Intelligenz ein nützliches Utensil, um Verhalten vorherzusagen und Entscheidungen zu treffen. Zusätzlich zu den genannten Themen stellt die Einteilung der Spielwelt in kleinere Bereiche ein Instrument zur weiteren Resourcenminderung dar. Die sogenannte *World Segmentation* ist bei sehr großen Spielwelten notwendig, um Spielern ausschließlich relevante Teile der Spielwelt zu präsentieren sowie der künstlichen Intelligenz eine eingeschränkte Sichtweise zu präsentieren.

Als letzter Punkt in diesem Seminarbeitrag wird mit den Influence Maps eine Methodik erklärt, die aufzeigen soll, wie eine Sichtweise auf die aktuelle Spielwelt für einen Non-Player-Character (NPC) verwaltet werden kann. Die Influence Maps sind räumliche zur Laufzeit erzeugte Entscheidungshilfen für die künstliche Intelligenz. Die Spielwelt wird in Zellen eingeteilt und jede Zelle erhält Informationen, die für den jeweiligen Nutzen der Influence Map von Bedeutung sind. Im Normalfall spiegelt eine Map den Einflussbereich von Spieleentitäten wider und verwaltet insbesondere im Hinblick auf die Strategiewahl wichtige Informationen.

L.2 Event-Driven-Behavior vs. Polling

In der Software-Entwicklung haben sich im Laufe der Jahre Muster (sogenannte Design Patterns) durch gewonnene Erfahrungswerte entwickelt, die sich gut auf wiederkehrende Problemstellungen anwenden lassen. Sie bilden einen Werkzeugkasten für jeden Softwareingenieur. Analog traten in der Spielerindustrie bestimmte Problemstellungen immer wieder auf, so dass an dieser Stelle ebenfalls Muster entstanden sind. Der Entwickler wird immer vor der Entscheidung stehen, wie er Spielobjekte miteinander kommunizieren bzw. interagieren lässt. Spielobjekte stellen von einfachen Entitäten bis hin zu komplizierten Non-Player-Character (NPC) alles dar, was nicht vom Spieler gezielt gesteuert wird. Diese Spielobjekte müssen ihren Zustand in bestimmten Intervallen aktualiseren und ihre Spiellogik ausführen.

Grundsätzlich gibt es zwei Methoden, wie Spielobjekte mit ihrer Umwelt in Verbindung treten können:

1. mit aktivem Betrachten der Spielwelt (Polling)
2. Warten auf Ereignisse (Event-Driven)

Idealerweise sollte jedes eigenständige Spielobjekt kontinuierlich die Spielwelt beobachten und entsprechend reagieren. Dazu wird zu jedem Zeitfenster die Spielwelt und/oder andere Spielobjekte nach interessanten Ereignissen befragt. Dieser Ansatz wird mit *Polling* bezeichnet. Es ist leicht ersichtlich, dass an dieser Stelle durch permanentes Anfragen ein unnötiger Overhead entsteht und unnötigerweise Rechenzeit verschwendet wird, denn nur wenige Veränderungen des Spielzustands sind für die meisten Spielobjekte von Interesse. Beispielsweise könnte ein Objekt auf ein ganz seltenes (vielleicht nie eintretendes) Ereignis warten und unnötigerweise in jedem Zeitfenster aktiv sein. Genauso unglücklich ist diese Vorgehensweise, wenn in der Spielwelt mehrere (hundert)tausende Objekte agieren und aktives Polling betreiben.

Die alternative Vorgehensweise zum Polling ist die ereignisgesteuerte Verwaltung von Spielobjekten. Als Beispiel stellen wir uns ein Fussballspiel vor, in dem der Ball den Schiedsrichter benachrichtigt, wenn er im Tor liegt. Es liegt auf der Hand, dass diese Vorgehensweise resourcensparender ist, als wenn der Schiedsrichter in jedem Zeitfenster überprüfen lässt, ob der Ball im Tor liegt. Objekte warten bei diesem Ansatz auf eintretende Ereignisse nicht mehr aktiv, sondern werden informiert.

Die Spielengine könnte für definierte Ereignisse Listen verwalten, in die sich Spielobjekte eintragen können. Tritt nun dieses Ereignis auf, werden genau die Spielobjekte benachrichtigt, die in der Liste aufgeführt snd. Spielobjekte welche sich nicht für das Ereignis eingeschrieben haben, werden auch nicht benachrichtigt. Alternativ könnte jedes Spielobjekt selbst Ereignislisten verwalten, in die sich andere Spielobjekte eintragen können und benachrichtigt werden, wenn das Objekt dieses Ereignis auslöst.

Ein weiteres interessantes Beispiel wäre eine Kollisionsbehandlung, in dem ein Charakter einen Pfeil abfeuert. Mit der Pollingstrategie würde jeder andere Charakter regelmäßig abtesten, ob ein Pfeil in seine Richtung fliegt. In der ereignisgesteuerten Steuerung könnte der Charakter, der den Pfeil abgeschossen hat, überprüfen, ob andere Charaktere eventuell getroffen werden. Diese werden anschließend automatisch informiert und könnten auf das

Ereignis reagieren, beispielsweise durch ein Ausweichmanöver. Eine Registrierung in einer Eventliste wäre nicht erforderlich. Diese Methodik könnte eine Art Kompromisslösung zwischen den diskutierten Techniken darstellen ([DeL00],[DeL01]).

L.3 Multiplayer-Spielearchitekturen

In Multiplayerspielen ist eine der größten Herausforderungen die Verwaltung der Spielwelt, so dass diese auf alle beteiligten Spieler in konsistenter Weise übertragen wird. Den Anfang der Entwicklung bildete Doom mit einer synchronen Peer-to-Peer Netzwerkarchitektur, in welcher jeder Spieler eine parallele Spieleengine im sogenannten *lock-step* betrieben hat. Die Maschinen waren in dem Sinne völlig gleichwertig, synchronisierten alle Inputs untereinander und führten die exakt gleiche Logik auf die exakt gleichen Eingaben aus. Die entstehenden Vorteile sind offensichtlich und folgende Nachteile ergaben sich:

- Mangel an Persistenz: Die Spieler mussten gleichzeitig das Spiel beginnen, neue Spieler konnten nicht dem Spiel beitreten (würde zum Spielstopp führen)
- Mit hoher Spielerzahl steigt die Wahrscheinlichkeit von netzwerk-induzierten Fehlern aufgrund des Kommunikationoverheads zur Synchronisation
- Mangel an Framerate (Anzahl Bilder pro Zeiteinheit) Skalierbarkeit: Alle Spieler müssen mit der gleichen internen frame-rate spielen

Der nächste Schritt waren monolitische Client-Server-Architekturen, welche zuerst in Quake eingesetzt wurden. In dieser Architektur gibt es einen ausgezeichneten Server, welcher für die Verwaltung des Spielzustands, der Eingaben und Simulation zuständig ist. Dies schließt auch die Künstliche Intelligenz mit ein, die bei Quake ausschließlich auf dem Server ausgeführt wird. Die übrigen Rechner werden als Clients bezeichnet, welche lediglich als einfache Terminals dienen. Ihre Aufgabe besteht darin, ihren Input an den Server zu senden und aus den vom Server gesendeten Objekten die Spielgrafik zu erzeugen (Rendern). Der Server verarbeitet die Eingabedaten der Clienten, aktualisiert den Spielzustand und schickt schließlich die zur Ausgabe benötigten Daten an die Clients. Der Vorteil besteht darin, dass die Architektur leicht skalierbar ist und der klassische (Internet-)Gameserver geboren wurde. Einen Überblick über die Architektur von Quake ist in Abbildung 65 dargestellt. Die Architektur wurde später in Quake2 und QuakeWorld um eine zusätzliche Simulations- und Prädikations-Logik ergänzt. Clientrechner waren neben den Rendern von Objekten damit in der Lage, die Verläufe (Trajektorien) der Spieler zu analysieren, um deren zukunftiges Verhalten zu erahnen (vgl. Abschnitt *Statistical Client Prediction*).

Zusätzlich sei darauf hingewiesen, dass die Client-Server-Architektur auch für einzelne Singleplayer-Spiele nützlich ist, denn es ermöglicht ein modulares Design mit einem fest vorgeschriebenen Kommunikationskanal. Im Hinblick auf NPCs der KI kann ein solcher Ansatz auch hilfreich sein, denn die NPCs könnten als einfache Clients agieren. Natürlich müsste dafür gewährleistet sein, dass der KI an dieser Stelle genügend Informationen zur Verfügung stehen, um gute Entscheidungen zu treffen ([Abr]).

Das Spiel Unreal führte einen Ansatz ein, der als generalisiertes Client-Server Modell bezeichnet wird. In diesem Modell ist der Server weiterhin für die Verwaltung und Simulation

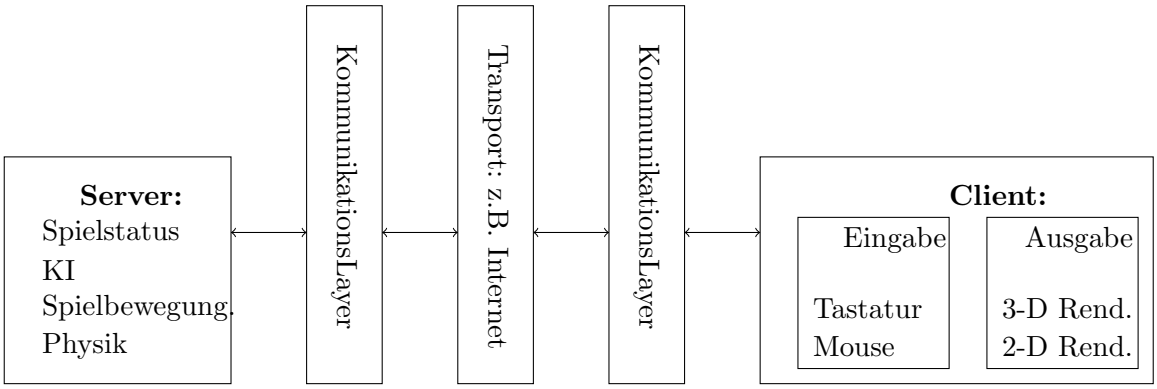


Abbildung 65: Spielearchitektur von Quake

des Spielzustands verantwortlich. Die Clients halten jedoch einen kleinen Teil des Spielzustands lokal und können den Spielfluß vorausahnen. Dies funktioniert, indem sie zum Beispiel Bewegungen vorausberechnen und mit dem annähernd gleichen Code (wie der Server) den neuen Spielstatus simulieren. Diese Vorgehensweise ist leicht zu programmieren und minimiert den Datenstrom zwischen Server und Client ([Swe]).

L.3.1 Distributed Gaming: Verteilte Ansätze für MMOG

In massive multiplayer online games (MMOG) teilen tausende von Spielern eine gemeinsame riesige Spielwelt. Solche Anwendungen laufen gewöhnlich auf performanten und zuverlässigen Server-Clustern. Trotzdem hat die Praxis gezeigt, dass solche Spiele sehr fehleranfällig sind und es häufig zu Ausfällen kommt, weil die Auslastung zeitweise sehr hoch ist. Nicht nur die Masse der teilnehmenden Spieler stellt eine große Herausforderung dar, auch die in großen Mengen vorkommende Spielobjekte und NPCs, mit denen oft eine komplexe rechenintensive Spiellogik verbunden ist. Des Weiteren ist die verfügbare Bandbreite ein großes technisches Hindernis der Spieleindustrie. Die grundlegende Idee für die folgenden Ansätze basiert darauf, die Last auf mehrere Rechner zu verteilen. Dazu wird die Spielwelt in kleinere Bereiche segmentiert (*World Segmentation*), welche auf mehreren Rechnern getrennt verwaltet werden können. Spieler bewegen sich nur in einem Bereich und werden auch hauptsächlich nur mit Änderungen aus diesem Bereich versorgt. Zusätzlich wird die Annahme gemacht, dass Spieler keine globale Sicht auf die Spielwelt besitzen und sich ihr Aktionsradius auf kleine Entfernungen beschränkt. Dies bringt den Vorteil mit sich, dass der Spieler nur mit den Ereignissen versorgt werden muss, die für seine eingeschränkten Interessen wichtig sind. An dieser Stelle spricht man auch von *Areas of Interest*, wobei zwischen den Spielertypen unterschieden werden kann. Mit dieser Strategie werden dem Spieler nicht alle Ereignisse in seinem Bereich der Spielwelt vermittelt, sondern ausschließlich die, die sich in seinem Area of Interest befinden. Dies spart enormen Aufwand beim Datenaustausch und damit Rechenzeit sowie Bandbreite.

Die einzelnen Bereiche der Spielwelt werden in vielen Ansätzen einzelnen Servern zugeordnet, womit wieder eine zentral-gesteuerte Architektur entsteht. Server müssen zusätzlich untereinander wichtige Ereignisse austauschen, so dass eine abstrakte Vorgehensweise für

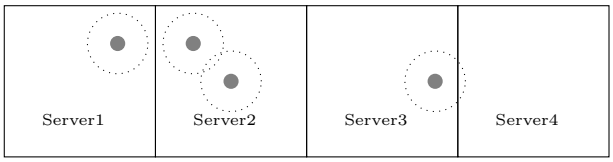


Abbildung 66: World-Segmentation über mehrere Server

das Eventhandling gewählt werden kann. Eventlisten werden normalerweise von Servern oder Spielobjekten selbst zur Verfügung gestellt bei denen sich Spielobjekte einschreiben. Allerdings ist die Beschränkung auf Spielobjekte nicht zwingend, so dass auch Server sich untereinander für wichtige Ereignisse am Nachbarserver anmelden können, um einen konsistenten Spielzustand zu evaluieren. Zum Beispiel könnte eine abgefeuerte Kugel von zwei verschiedenen Servern verwaltete Bereiche kreuzen, so dass die beiden zuständigen Server von diesem Ereignis informiert werden müssen. Als Beispiel für das Eventhandling soll hier noch die in Abbildung 66 gezeigte Situation des zwischen Server 3 und 4 lokalisierten Spielers angeführt werden. Jeder Server verwaltet das Eventhandling für seinen Bereich der Spielwelt und jeder Spieler aus diesem Bereich ist bei ihm angemeldet. Wenn nun ein Spieler von einem Server zum anderen wandert, dann löst der erste Server ein Ereignis beim Zielserver aus, in welchem mitgeteilt wird, welcher Spieler sich nun in dem neuen Bereich befindet und wie sein Spielstatus ist. In diesem Schritt meldet sich der Spieler gleichzeitig bei dem neuen Server an und wird ab sofort von ihm verwaltet ([Abd],[AA]).

In den Ansätzen bei denen die Spielwelt aufgeteilt und einzelnen Servern zugeordnet wird, können weiterhin Situationen starker Auslastung einzelner Server entstehen, wenn ein Teil der Spielwelt eines Servers überbevölkert wird, die im schlimmsten Fall in einem Serverausfall endet. Ein neuer Ansatz wäre eine Spielearchitektur basierend auf Peer-to-Peer-Netzen (s. Abbildung 67). Peer-to-Peer Systeme werden hauptsächlich in Filesharingnetzwerken eingesetzt. Sie zeichnen sich vorallem durch hohe Skalierbarkeit und Verfügbarkeit aus. Unterschieden wird zwischen strukturierten und unstrukturierten Netzwerken. In unstruktrierten Netzwerken werden Daten willkürlich in irgendwelchen Peers gespeichert und bei Anfragen das komplette Netz geflutet. Ein Knoten sendet hierbei an seine bekannten Nachbarn Anfragen, die sie wiederum an ihre Nachbarn weiterleiten, bis ein passender Peer gefunden wurde. In strukturierten Netzen wird der Zielknoten mittels einer verteilten Hashtabelle gesucht. Das gesuchte Objekt wird in der Hashtabelle auf einen Adressraum abgebildet, der die Knoten beinhaltet, welche das gesuchte Objekte speichern.

Age of Empires ist ein bekanntes Beispiel für eine Peer-to-Peer Netzwerklösung. Die Spieler sind sternförmig miteinander verbunden, so dass jeder Spieler eine Verbindung zu jedem Spieler besitzt. Jeder Spieler simuliert die komplette Spielwelt und teilt Änderungen direkt an alle anderen Mitspieler mit. Die Architektur ermöglicht geringe Latenzzeiten und der Ausfall eines Spielers führt nicht zum direkten Spielabbruch. Allerdings können Spieler den Spielzustand leicht unerlaubt beeinflussen (Cheaten).

Das Konzept hinter Peer-to-Peer-Ansätzen bei MMOG besteht darin, dass die Spielwelt - wie gehabt - in Zonen aufgeteilt wird und Verwaltungsaufwand an alle beteiligten Spieler delegiert wird. Auch hier besteht der Grund dafür in der Annahme, dass nur ein limitierter Bereich der Spielwelt für jeden Spieler von Bedeutung ist, denn die Charaktere besitzen nur eine eingeschränkte Sichtweite und begrenzte Bewegungsgeschwindigkeit. Folglich ist

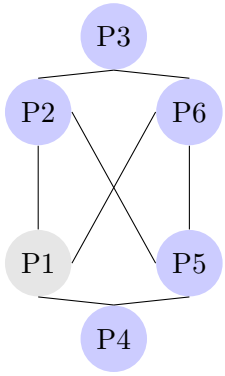


Abbildung 67: Peer-to-Peer Netzwerk

nur die gerade aktuelle Region von Bedeutung und alle angrenzenden Nachbarregionen.

Erste Ansätze teilten die Spielwelt in Zonen auf und jeder Spieler war für eine Zone verantwortlich. Ein großes Problem entstand jedoch, wenn Spieler ohne Ankündigung das Spiel verließen oder die Rechner abstürzten, so dass immer Backups des Spielstatus vorhanden sein müssten. Neuere Ansätze verteilen den transienten Spielzustand in einem P2P Netz und speichern den persistenten Spielzustand auf einem zentralen Server. Der persistente Spielzustand beinhaltet im Gegensatz zum transienten Spielzustand alle Informationen, die nötig sind um das Spiel in diesem Zustand wiederherzustellen. Für jede Region wird ein sogenannter Koordinator ausgewählt, der alle Ereignisse für eine Region betreffend abfängt und sie via Multicast an alle Spieler in der Region weiterleitet. Grundsätzlich ähnelt die Vorgehensweise hier der World Segmentation in einem Server-Cluster (vgl. Abschnitt L.3.1). In Region 1 der Abbildung 68 ist das typische Weiterleiten (Multicast) eines Kooardinators abgebildet, der alle Spieler über ein Ereignis informiert. In Region 2 sendet ein Spielobjekt an den Koordinator ein Ereignis mit Informationen über dessen Zustandsänderung. Betrifft ein Ereignis (zum Beispiel in einer Kampfszene) nur die beteiligten Spieler, dann können die jeweiligen Spielerzustände ohne einen Koordinator aktualisiert werden (siehe Region 3).

Wie bereits angesprochen besteht weiterhin die Gefahr, dass der Koordinator eines Bereiches nicht mehr verfügbar ist und ein Teil der Spielwelt verloren geht. Um dies vorzubeugen, werden normale Ereignisse zur Fehlerbehandlung genutzt. Erreicht ein Ereignisse nicht seinen Koordinator, dann wird die Nachricht direkt an einen sogenannten Replikanten weitergeleitet. Ein Replikant ist ein weiterer Peer des Adressraums für den betreffenden Bereich der Spielwelt. Dieser übernimmt nun die Rolle des Koordinators.

Kritikpunkt des vorherigen Konzeptes war die mögliche Überlastung eines Servers. Mit dem P2P-Ansatz kann dieses Risiko durch mehrere Koordinatoren, welche sich um unterschiedliche Objekte kümmern, gemildert werden. Zum Beispiel könnte ein Koordinator nur für die Charakterverwaltung zuständig sein und ein Anderer für die Spielobjekte.

L.3.2 Statistical Client Prediction

Statistical Client Prediction oder auch Dead Reckoning genannt, ist eine Methodik, um das Verhalten von Spielern vorherzusagen. Das Ziel dabei ist es, möglichst wenige In-

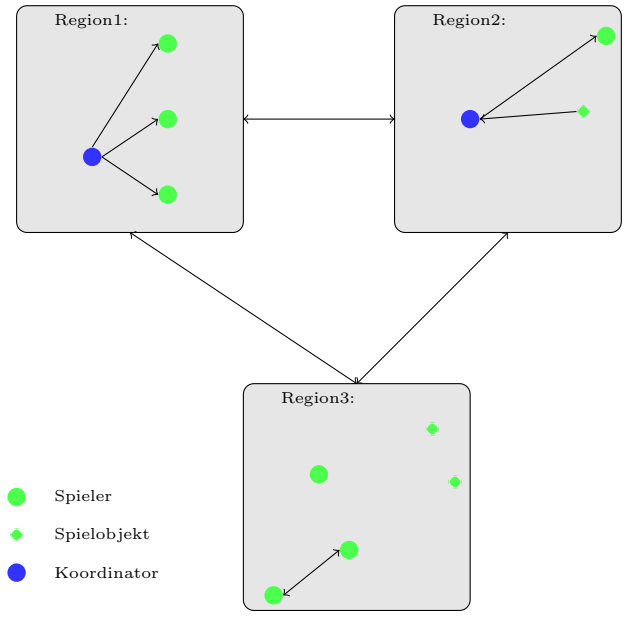


Abbildung 68: P2P-Ansatz für MMOGs

formationen zur Spielkommunikation auszutauschen und die Nutzung der Bandbreite zu Reduzieren. Voraussetzung für das Verfahren ist eine statistische Datenbank, die basierend auf einer Spielsituation das wahrscheinlichste Verhalten von Spielern vorhersagen kann. Es gibt zwei verschiedene Vorgehensweisen, wie die Datenbank aufgebaut werden kann. Zum einen können einfache Regelwerke für Spielsituationen erstellt werden. Spieler einer Fussballmannschaft, die nicht im Ballbesitz ist, werden sich höchstwahrscheinlich immer in Richtung des Balls orientieren. Zum Einen wird das Verhalten von Spielern einfach extrapoliert, um die zukünftige Bewegung von Spielern zu erahnen. Zum Anderen könnten typische Spielsituationen durch Muster erkannt werden und daraus Entscheidungen abgeleitet werden. Neuronale Netze eignen sich hervorrangend für diese Aufgabe. Im Weiteren wird diese Datenbank an alle Clients verteilt, damit jeder dieselben Entscheidungen treffen kann. Die Idee der Statistical Client Prediction ist einfach und soll im Folgenden kurz erläutert werden.

Jeder Spieler errechnet auf Basis der statistischen Datenbank die neuen Bewegungen der anderen Spieler aus. Zusätzlich errechnet jeder Spieler seine wahrscheinlichste neue Position auf die gleiche Art und Weise. Unterscheidet sich diese Position von der tatsächlichen, so wird ein Paket mit einer Korrektur an die übrigen Spieler verschickt. Mit einer guten statistischen Datenbank kann man annehmen, dass die neuen Positionen gut erraten werden und wenige Pakete zur Korrektur über das Netzwerk verschickt werden müssen ([MW04]).

L.3.3 Distributing AI to Clients

In Echtzeit-Strategie Spielen (RTS) wird ein großer Teil der benötigten Rechenzeit für die Berechnungen des KI-Modules verwendet. Eine intuitive Lösung, um die Performance zu erhöhen, wäre je nach Bedarf den Server-Cluster mit zusätzlichen Rechnern auszustat-

ten. Allerdings wäre es auch möglich nichtgenutzte Rechenleistung auf den Clientrechnern zu benutzen. Beispielsweise ist die Berechnung von strategischen Analysen des Terrains durch die KI eine sehr rechenintensive Aufgabe. Nun könnte man sich die Rechenleistung einiger Clients zu Nutze machen, in dem diese Algorithmen auf den Clientrechnern ausgeführt werden. Hierzu wäre es aber erforderlich, dass (möglichst) Algorithmen ausgelagert werden, die wenige Eingabedaten benötigen und geringe Ausgabedaten erzeugen, um die Bandbreite nicht zu strapazieren. Ein Client, welcher das Terrain kennt, wäre also ein guter Kandidat für eine solche Operation. Erhält der Client zusätzliche Spielinformationen, so besteht generell die Gefahr, dass der Spieler diese ausnutzen kann und einen Spielvorteil hat.

Des Weiteren könnten Einheiten von Clienten vollständig verwaltet werden. Allerdings sollten beispielsweise Armeen nicht aufgeteilt werden, sondern von einem einzelnen Clienten bearbeitet werden, weil die Einheiten untereinander interagieren und damit keine Informationen über das Netzwerk ausgetauscht werden müssten.

Im Grunde lassen sich einfache Teile der KI ohne größeren Aufwand auf teilnehmende Clienten verteilen. Wichtig ist trotzdem, dass Clienten nicht willkürlich ausgesucht werden. Es macht keinen Sinn, einerseits einen überlasteten Clienten weitere Rechenoperationen aufzudrängen und andererseits sollte der Client möglichst viele der zur Ausführung der KI benötigten Daten bereits kennen, um den Informationsaustausch über das Netzwerk so gering wie möglich zu halten ([AA]).

L.4 Influence Maps

Eine Künstliche Intelligenz betrachten wir als intelligent, wenn sie in einem bestimmten Kontext kluge Entscheidungen trifft. Gute Entscheidungsfindungen basieren aber nicht nur alleine auf den besten verfügbaren Daten, sondern oft ist die Art und Weise wie die Daten verwaltet werden von größerer Bedeutung. Es ist ersichtlich, dass alle Daten nur soweit sinnvoll sind, wie sie in einem Kontext passend eingeordnet werden. Im Folgenden wird eine Methode beschrieben, wie eine strategische Sichtweise eines Charakters auf den aktuellen Spielstatus konstruiert und verwaltet werden kann.

Influence Maps haben sich als wichtiges Instrument gerade in Strategiespielen erwiesen, um taktische Analysen zu betreiben. Eine Influence Map ist eine räumliche Repräsentation von dem Wissen eines KI-Agenten über die aktuelle Spielsituation. Die Map basiert auf die zugrunde liegenden physikalischen und geographischen Informationen der aktuellen Spielwelt und spiegelt den Einfluss der Charaktere auf der Map wieder. Beispielsweise kann eine Map aufzeigen, wo feindliche Spieler zu finden sind oder vermutet werden, wo Grenzen zwischen Charakteren liegen, welche Gegenden noch zu erkunden sind oder wo in der Vergangenheit vermehrt Kämpfe stattgefunden haben. Darüber hinaus sind Influence Maps in der Lage Positionen zu bewerten und wichtige Stellungen zu sichten, genauso wie Schwachstellen auszumachen.

Es gibt keinen Standard-Algorithmus zum Erzeugen von Influence Maps und auch keine genaue Anleitung zum Anwenden der Technik. Im Weiteren werden einfache Konzepte erklärt und dem Leser steht es frei, abhängig von den jeweiligen Anforderungen selbst Influence Maps zu erarbeiten.

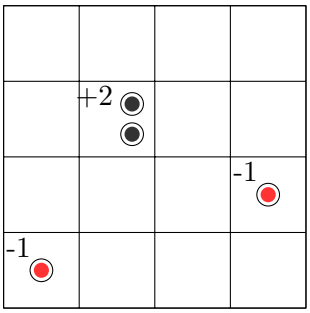


Abbildung 69: Ausgangssituation für eine Influence Map

0.51	0.79	0.47	0.06
0.66	1.66	0.53	0.06
0.07	0.40	0.03	−0.74
−0.74	0.17	−0.25	−0.39

Abbildung 70: Vollständige Propagierung und Erzeugung einer Influence Map

Influence Maps können in vielen Topologien angewendet werden, jedoch betrachten wir (wie für Strategiespiele üblich) 2D-Spielewelten. Als einfaches Beispiel nehmen wir eine quadratische Map (siehe Abbildung 5). Jede Zelle wird initial mit einem Wert 0 belegt und soll die Kampfstärke der Spieler einschätzen. Für jede Einheit auf einer Zelle bewerten wir die Kampfstärke mit $+1$ für eigene Einheiten und -1 für feindliche. Im nächsten Schritt wird der Einfluss auf die anliegenden Zellen propagiert. Die Propagierung dient dazu eine Bewertung aller Zellen der Map durchzuführen. Nehmen wir für unser Beispiel an, dass die Kampfstärke zu einer Nachbarzelle um 50 Prozent an Effektivität verliert. Zur Propagierung können prinzipiell zahlreiche Ansätze gewählt werden, von linearen Abhängigkeiten bis hin zum exponentiellen Abfall. Ein Wert von 1 würde in der Nachbarzelle in unserem Beispiel schließlich einen Einfluss von $0,5$ erzielen. In Abbildung 6 ist zu erkennen, wie sich der Einfluss der Kampfstärken auf dem Spielfeld entwickelt. Dazu werden alle Werte der Zellen zusammengezählt.

Es sollte ersichtlich sein, dass wir nun ein gutes Bild davon erhalten, wo welcher Spieler einen Einfluss ausübt. Wichtiger ist allerdings, dass wir eine Grenzlinie zwischen Einflussgebieten ziehen können und Gebiete den jeweiligen Spielern zuordnen können (in Abbildung 6 dick gekennzeichnet). Wir erhalten nun eine gute Sichtweise, um Einheiten geschickt zu postieren. Eine weitere Variante Propagierung gewichtet die Stärken der Gegner. Eine Gewichtung über 1 würde eine eher defensivere Rolle entwickeln, wobei eine Gewichtung unter 1 eine offensivere Rolle propagiert.

Normalerweise werden die standardmäßigen Statistiken, die aus den Influence Maps gewonnen werden, nicht alleine für die Entscheidungsfindung genutzt. Oft werden mehrere Werte kombiniert, um zu entscheiden, welche Zellen für eine bestimmte Aufgabe in Zu-

kunft zu besetzen sind. Dazu werden mehrere Influence Maps überlagert. Diese Werte werden als *Desirability Values* bezeichnet und ergeben sich häufig aus einer gewichteten Summe der beteiligten Werte bzw. überlagerten Influence Maps. Die Wahl der Koeffizienten ist natürlich subjektiv und abhängig von der allgemeinen Spiellogik. Nehmen wir als Beispiel ein *Desirable value*, der die Verwundbarkeit von Schlüsselpositionen auf der Spielkarte repräsentiert. Schlüsselpositionen wären Orte in der Spielwelt, die zum Beispiel hohe Resourcen vorweisen können oder Basen von Spielern beherbergen. Die Verwundbarkeit könnte nun errechnet werden aus der Kampfstärke des verteidigenden Spielers und einer Bewertung der Schlüsselposition hinsichtlich ihres Nutzens. Eine hohe Verwundbarkeit könnte demnach aus einer sehr wichtigen Position resultieren, welche nicht hinreichend bewacht ist. Das gibt einerseits allen feindlichen Spielern Indizien für mögliche Angriffspunkte, aber andererseits den verteidigenden Spieler die Information, dass er diese Position besser bewachen sollte. Abschließend kann man sagen, dass eine Influence Map ein sehr nützliches Tool bei der Entwicklung einer KI darstellt. Allerdings besteht ein natürlicher Tradeoff zwischen der Zellgröße von Influence Maps und deren Komplexität hinsichtlich des Rechenaufwands. Trivialerweise lässt eine kleine Zellgröße die KI bessere Entscheidungen treffen, allerdings wird der Gewinn an Intelligenz durch höheren Rechenaufwand gewonnen. Es ist zu beachten, dass nur bis zu einem bestimmten Grad der Gewinn an Informationen weiter ansteigt, denn irgendwann kann kein Informationsgewinn mehr erzielt werden, weil nur noch redundante Informationen berechnet werden. Als Anhaltspunkt für eine Größenordnung soll eine Größe von Zellen vorgegeben werden, die zehn bis zwanzig Einheiten beherbergen kann ([Kir04]).

M Ergebnisse der Studie auf dem Campusfest

M.1 Teilnehmerstruktur

Tabelle 10: Struktur der Teilnehmer am Campusfest

Alter		Geschlecht		Expertise	
	Anzahl		Anzahl		Anzahl
< 14	21	männlich	138	täglich	53
15 - 17	16	weiblich	45	>1mal/Woche	39
18 - 22	47			1mal/Woche	17
23 - 29	72			1mal/Monat	16
>30	27			seltener	58

M.2 Gesamte Ergebnisse

Tabelle 11: Bewertung von Spielspaß, Schwierigkeit und Klugkeit in den Spielkombinationen

Strategie-kombination		viel mehr	mehr	neutral	weniger	viel weniger
EA gegen dr	Spielspaß	4	11	11	6	1
	Schwierigkeit	2	17	7	5	2
	Klugheit	5	12	9	6	1
dr gegen EA	Spielspaß	2	7	17	5	0
	Schwierigkeit	1	7	12	10	1
	Klugheit	3	7	13	8	0
NN gegen dr	Spielspaß	0	8	12	9	1
	Schwierigkeit	1	11	10	6	2
	Klugheit	1	15	9	4	1
dr gegen NN	Spielspaß	8	7	11	6	1
	Schwierigkeit	7	10	7	8	1
	Klugheit	6	8	10	6	3
EA gegen NN	Spielspaß	4	5	7	9	0
	Schwierigkeit	3	10	7	4	1
	Klugheit	4	11	2	7	1
NN gegen EA	Spielspaß	4	8	15	4	0
	Schwierigkeit	6	8	10	6	1
	Klugheit	5	9	6	10	1

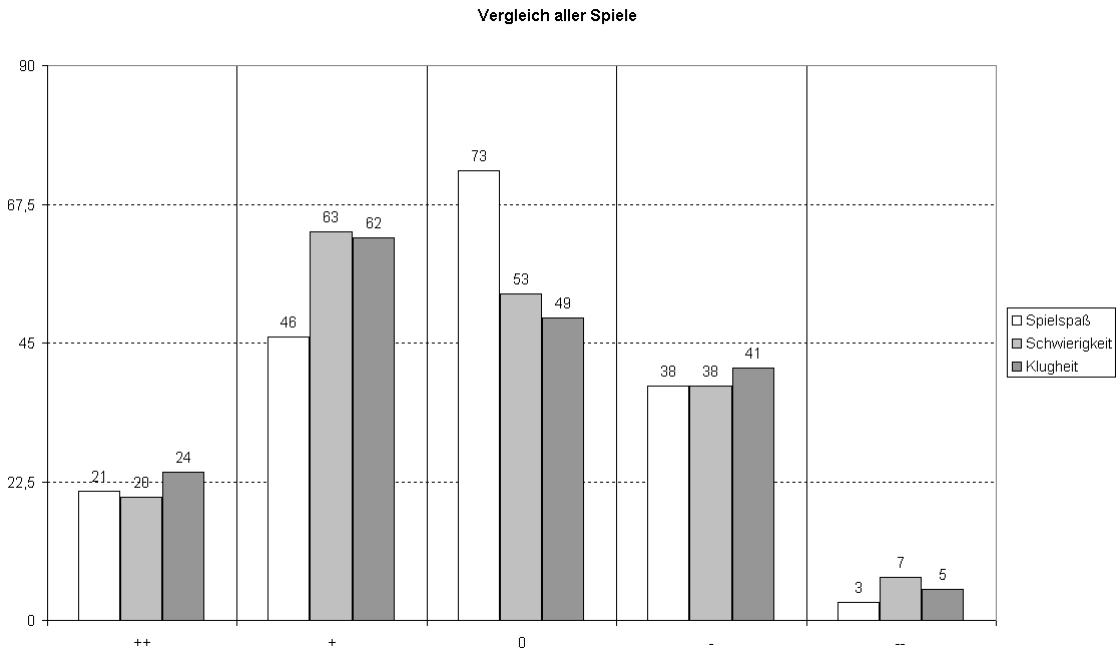


Abbildung 71: Antworten der Versuchspersonen ohne Differentierung nach Strategiekombination.

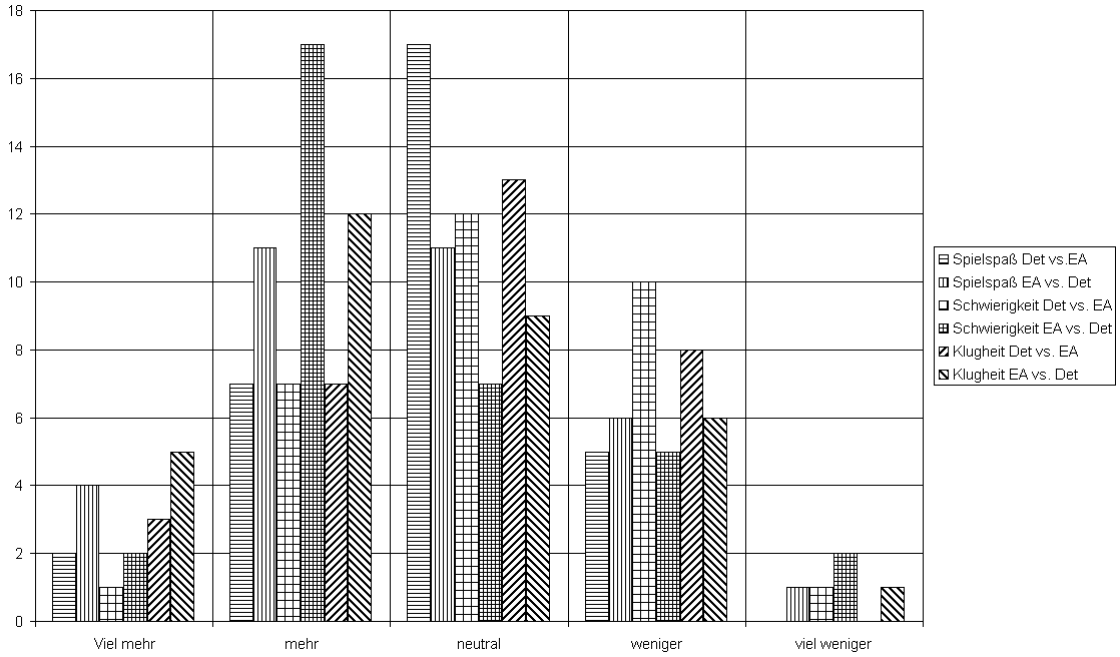


Abbildung 72: Antworten der Versuchspersonen bezogen auf die Strategiekombinationen EA dr.

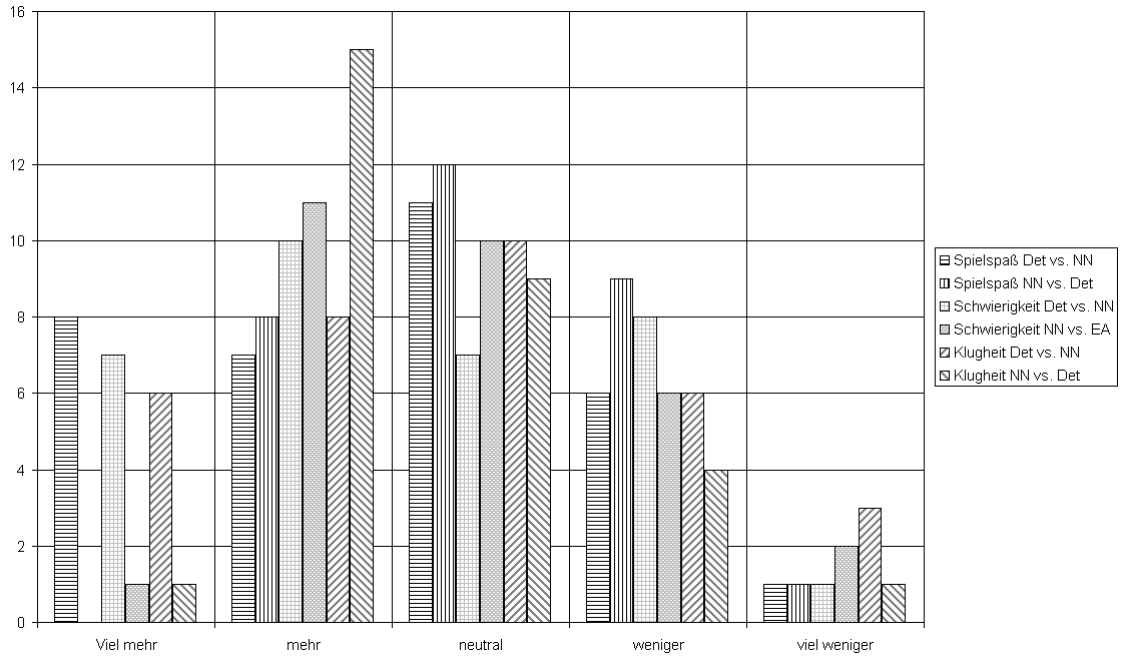


Abbildung 73: Antworten der Versuchspersonen bezogen auf die Strategiekombinationen NN dr.

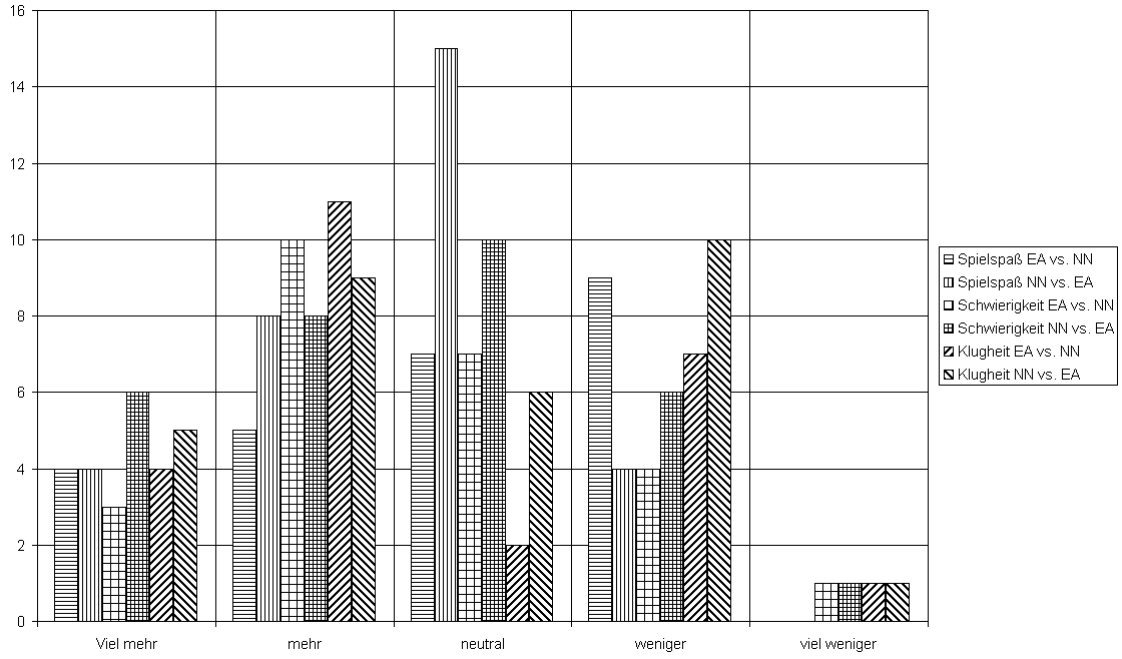


Abbildung 74: Antworten der Versuchspersonen bezogen auf die Strategiekombinationen EA NN

N PG-Ordnung

N.1 Abstimmungsmodus

- Satzungänderungen
  Können nur beschlossen werden wenn mindestens elf Teilnehmerinnen anwesend sind und neun Teilnehmerinnen für die Änderung stimmen.

- Teamentscheidungen
  Teams können in ihrem Verantwortlichkeitsbereich Entscheidungen ohne Rücksprache mit den anderen PG-Teilnehmerinnen treffen, sofern diese sich nicht auf alle auswirken. Hierfür reicht ein einfacher Mehrheitsentscheid.

- sonstige Entscheidungen
  Alle übrigen Entscheidungen werden per Mehrheitsentscheid getroffen, bei Gleichstand und nach angemessener Wartezeit, wenn niemand mehr durch Argumente überzeugt werden kann, wird per Münzwurf entschieden.

N.2 Organisatorisches

- Reaktionszeit
  Mails müssen täglich abgerufen und gelesen werden, so dass eine Reaktionszeit von maximal 24 Stunden erreicht wird. Sollte dies aus irgendeinem Grund nicht möglich sein, muss dieses der PG mitgeteilt werden. Ebenso ist das Forum sowie das Wiki der PG täglich abzurufen.

- Vorlagen
  Für sämtliche Schriftstücke sind die Vorlagen zu verwenden, die im Verlauf der PG dazu erstellt wurden.

- Protokolle
  Protokolle müssen bis zum nächsten Tag allen Teilnehmerinnen der PG zur Verfügung stehen.

- Sitzungsvorbereitung
  Die Tagesordnung sowie zur Diskussion stehendes Material muss allen PG-Teilnehmerinnen mindestens 24 Stunden vor der PG zur Verfügung stehen, damit das Material gesichtet sowie Anträge eingereicht werden können.

- Anwesenheit
  Die Teilnahme an den PG-Sitzungen ist Pflicht. Ist eine PG-Teilnehmerin verhindert, so muss dies möglichst früh mitgeteilt werden. Sprechen keine triftigen Gründe dagegen, so akzeptiert die PG die Abwesenheit.

- Anrede
  In allen Schriftstücken der PG ist die weibliche Anrede zu verwenden.

- C++ Style Guide
  Die Angaben im C++ Style Guide sind für alle PG-Teilnehmerinnen bindend.

- Kennzeichnung
  Im Betreff sämtlicher E-Mails zu einem Thema der PG ist [PG511] zu Beginn im Betreff zu vermerken.

Abbildungsverzeichnis

Literatur

[AA] Amir, G. ; Axelrod, R.: *Massively Multiplayer Game Development 2: Architecture and Techniques for an MMORTS.* `http://www.gamedev.net/reference/articles/article1948.asp`

[Abd] Abdelwahed, O. A.: *Distributed Gaming.* `http://www.gamedev.net/reference/articles/article1948.asp`

[Abr] Abrash, M.: *Quake's 3-D Engine: The Big Picture.* `http://www.bluesnews.com/abrash/chap70.shtml`

[AK89] Aarts, E. ; Korst, J.: *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing.* Wiley, Hoboken, 1989

[AP02] A. Petrus, Engelbrecht: *Computational Intelligence An Introduction.* Wiley-VCH Verlag, Weinheim, 2002

[Aus07] Austin, The U. a.: *NERO-Projekt Homepage.* `http://nerogame.org/`. Version: 2007

[AV06] Arthur, D. ; Vassilvitskii, S.: How slow is the k-means method? In: *SCG '06: Proceedings of the twenty-second annual symposium on Computational geometry.* ACM Press, New York, NY, USA, 2006. – ISBN 1–59593–340–9, 144–153

[Bab03] Babuskov, M. et a.: *NJam.* `http://njam.sourceforge.net/`. Version: 2003

[BB03] Bartz-Beielstein., T.: Experimental analysis of evolution strategies- overview and comprehensive introduction. In: *Tecnical Report des SFB531* (2003)

[BBPV04] Bartz-Beielstein, T. ; Parsopoulos, K.E. ; Vrahatis, M.N.: Design and Analysis of Optimization Algorithms Using Computational Statistics. In: *Applied Numerical Analysis and Computational Mathematics- ANACM* 1/3 (2004)

[BFM00a] Bäck, T. ; Fogel, D. ; Michalewicz, T.: *Evolutionary Computation 1 - Basic Algorithms and Operators.* Institute of Physics Publishing, 2000

[BFM00b] Bäck, T. ; Fogel, D. ; Michalewicz, T.: *Evolutionary Computation 2 - Advanced Algorithms and Operators.* Institute of Physics Publishing, 2000

[BH06] Baba, N. ; Handa, H.: Utilization of Evolutionary Algortíthms to Increase Excitement of the COMMONS Game. In: *Proceedings of the SAB'06 Workshop on Adaptive Approaches for Optimizing Player Satisfaction in Computer and Physical Games* (2006), Oct

[BK05] *Kapitel* 1. In: Bull, L. ; Kovacs, T.: *Foundations of Learning Classifier Systems: An Introduction.* Springer, 2005, S. 1 – 17

[BKI03] Beierle, C. ; Kern-Isberner, G.: *Methoden wissensbasierter Systeme.* 2., überarb. und erw. Aufl. Vieweg, 2003

[Boe94] Boer, B. de: *Classifier Systems: A useful approach to machine learning?*, Leiden University, Diss., August 1994

[Boo82] Booker, L. B.: *Intelligent behavior as an adaptation to the task environment*, The University of Michigan, Ann Arbor, MI., Diss., 1982

[Boo07] Boost: *Boost C++ Libraries*. `http://www.boost.org`. Version: 2007. – Stand: 09.07.2007

[Bri05] Brill, M.: *Mathematik für Informatiker*. 2. Auflage. Hanser-Verlag, München Wien, 2005

[BS04] Bourg, D. M. ; Seemann, G.: *AI for Game Developers*. O'Reilly Media, Inc., 2004. – ISBN 0596005555

[Buc05] Buckland, M.: *Programming Game AI by Example*. 1. Auflage. Wordware Publishing Inc.,U.S., U.S., 2005

[Car78] Carrol, L.: *Through the Looking-Glass*. Neuauflage 2003. PENGUIN Group Inc, London, 1978

[Cha03] Champandard, A. J.: *AI Game Development: Synthetic Creatures with Learning and Reactive Behaviors*. New Riders Publishing, 2003

[Dav72] Davis, M. D.: *Spieltheorie für Nichtmathematiker*. 2. Auflage. R. Oldenbourg, München Wien, 1972

[DD97] D. Dubois, H. P.: The Place of Fuzzy Logic in the AI. In: *Lecture Notes in Artificial Intelligence* 16 (1997), S. 9–20

[DeL00] DeLoura, M.: *Interdisciplinary Systems Research*. Bd. 26: *Game Programming Gems*. B&T, Basel, 2000

[DeL01] DeLoura, M.: *Interdisciplinary Systems Research*. Bd. 26: *Game Programming Gems 2*. B&T, Basel, 2001

[Dij59] Dijkstra, E. W.: A note on two problems in connexion with graphs. In: *Numerische Mathematik. 1*, 1959, S. 269–271

[Eng02] Engelbrecht, A.: *Computational Intelligence - An introduction*. Wiley, 2002

[For82] Forgy, C.: Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem. In: *Artificial Intelligence* 19 (1982), Nr. 1, S. 17–37

[GHJV95] Gamma, E. ; Helm, R. ; Johnson, R. ; Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[GKI06] G. Kern-Isberner, C. B.: *Methoden wissensbasierter Systeme*. Vieweg Verlag, 2006

[GMS04] Graham, R. ; McCabe, H. ; Sheridan, S.: Neural Networks for Real-Time Path Finding In Computer Games. In: *ITB Research Conference* (2004)

[HB99] Heitkotter, J. ; Beasly, D.: *The Hitch-Hiker's Guide to Evolutionary Computation*. `http://web.tiscali.it/LCS/`. Version: 1999

[HBC+99] Holland, J. H. ; Booker, L. B. ; Colombetti, M. ; Dorigo, M. ; Goldberg, D. E. ; Forrest, S. ; Riolo, R. L. ; Smith, R. E. ; Lanzi, P. L. ; Stolzmann, W. ; Wilson, S. W.: What Is a Learning Classifier System? In: *j-LECT-NOTES-COMP-SCI* (1999), 3 – 32. `http://www.springerlink.com/content/bnajvvc1wawh154h/fulltext.pdf`

[HI96] Holler, M. J. ; Illing, G.: *Einführung in die Spieltheorie*. 3. Auflage. Springer, Berlin Heidelberg New York, 1996

[Hil07] Hillel, E.: *Die Spaß-Richtlinie*. `www.hillel.de/spass/spass.html`. Version: 2007

[HNR68] Hart, P. E. ; Nilsson, N. J. ; Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. In: *Transactions on Systems Science and Cybernetics SSC4 (2)*, 1968, S. 100–107

[Hoo96] Hooker, J. N.: Testing Heuristics: We Have It All Wrong. In: *Journal of Heuristics* 39(2) (1996)

[IK95] Ivanova, I. ; Kubat, M.: Initialization of neural networks by means of decision trees. In: *Knowledge-Based Systems* 8 (1995), Nr. 6, 333-344. `citeseer.ist.psu.edu/ivanova95initialization.html`

[IR04] Ip, Y.L. ; Rad, A.B.: Incorporation of feature tracking into simultaneous localization and map building via sonar data. In: *Journal of Intelligent and Robotic Systems* 39(2) (2004), S. 149–172

[Jan04] Jansen, T.: *Evolutionäre Algorithmen, Vorlesungsfolien*. Universität Dortmund, 2004 `http://ls2-www.cs.uni-dortmund.de/~jansen/EvoAlg2004/skript-evoAlg.pdf`

[Kel00] Keller, H.: *Maschinelle Intelligenz*. Vieweg, 2000

[Kir04] Kirmse, A.: *Interdisciplinary Systems Research*. Bd. 26: *Game Programming Gems 4*. B&T, Basel, 2004

[Kle06] Kleinz, T.: Punktejagd und Professoren. In: *c't, S. 42: Spiele und Wissenschaft* (2006), August

[Koh01] Kohonen, T.: *Self-Organizing Maps*. 3. Auflage. Springer-Verlag, Berlin, 2001

[Kov96] Kovacs, T.: *Evolving Optimal Populations with XCS Classifier Systems*. `http://web.tiscali.it/LCS/Papers/LCS10.pdf.zip`. Version: 1996

[KS01a] Kalyanpur, A. ; Simon, M.: *Pacman Using genetic Algorithms and Neural Networks*. University of Maryland, 2001

[KS01b] Kalyanpur, A. ; Simon, M.: *Pacman using Genetic Algorithms and Neural Networks*. `http://www.ece.umd.edu/~adityak/Pacman.pdf`. Version: 2001

[Lam01] Lampel, J.: *Einsatz von neuronalen Netzen in einem Bot für Counterstrike*. `http://johannes.lampel.net/bll137pub.pdf`. Version: 2001

[LC01] Leuf, B. ; Cunningham, W.: *The Wiki Way. Collaboration and Sharing on the Internet.* Addison Wesley, Reading, 2001

[Lee05] Lee, R. S. T.: *Fuzzy-neuro approach to agent applications.* Springer-Verlag GmbH, 2005

[Lip07] Lippe, Prof. Dr. W.: *Einführung in neuronale Netze, Vorlesungsfolien.* `http://wwwmath.uni-muenster.de/SoftComputing/lehre/material/wwwnnscript/startseite.html`. Version: 2007

[Mai92] Mailath, G. J.: Introduction: Symposium on Evolutionary Game Theory (Sekundärquelle). In: *Journal of Economic Theory* Bd. 57. Elsevier Inc., 1992, S. 259 – 277

[May01] Mayne, I.: Nur der Nutzen zählt oder: Was ist rational?? In: *ungewußt, die Zeitung für angewandtes Nichtwissen* Bd. 9. Institut für Angewandtes Nichtwissen e.V., Berlin Heidelberg New York, 2001, S. Onlinezeitung – keine Seitenangabe

[MBC+06] Miikkulainen, R. ; Bryant, B. D. ; Cornelius, R. ; Karpov, I. V. ; Stanley, Kenneth O. ; Yong, Chern H.: *Computational Intelligence in Games.* `http://nn.cs.utexas.edu/downloads/papers/miikkulainen.wcci06.pdf`. Version: 2006

[MS01] Moret, B. M. ; Shapiro, H. D.: Algorithms and Experiments: The New (and Old) Methodology. In: *Journal of Universal Computer Science* 7 (2001), Nr. 1

[MW04] McCoy, A. ; Ward, T.: Towards Statistical Client Prediction – Analysis of User Behaviour in Distributed Interactive Media. In: *CGAIDE* (2004). `http://www.eeng.nuim.ie/~tward/copies_of_%20publications/TowardsStatisticalClientPrediction(CGAIDE2004).pdf`

[Nar04] Nareyek, A.: AI in Computer Games. In: *Queue* (2004), Feb

[Nas50] Nash, J. F.: *Non-Cooperative Games*, Princeton University, New Jersey, USA, Diss., May 1950

[NBKK03] Nauck, D. ; Borgelt, C. ; Klawonn, F. ; Kruse, R.: *Neuro-Fuzzy-Systeme.* 3. Auflage. Vieweg-Verlag, Wiesbaden, 2003

[Neu28] Neumann, J. von: Zur Theorie der Gesellschaftsspiele. In: *Mathematische Annalen* Bd. 100. Springer, Berlin, 1928, S. 295 – 320

[NKK94] Nauck, D. ; Klawonn, F. ; Kruse, R.: *Neuronale Netze und Fuzzy-Systeme.* Vieweg, 1994

[NM67] Neumann, J. von ; Morgenstern, O.: *Spieltheorie und wirtschaftliches Verhalten.* 2. Auflage. Physica, Würzburg, 1967

[Pei65] Peirce, C. S. ; Hartshorne, Charles (Hrsg.) ; Weiss, Paul (Hrsg.) ; Burks, Arthur (Hrsg.): *Collected papers of Charles Sanders Peirce.* Bd. 2. Harvard University Press, 1965

[PN95] P. Norvig, S. J. R.: *Artificial Intelligence - A Modern Approach*. 1. Auflage. Prentice Hall, U.S., 1995

[Pre07] Preuss, M.: Reporting on Experiments in Evolutionary Computation. In: *i dont know* (2007)

[PTP01] Pratchett, T ; Truran, T. ; Pearson, B.: *Thud*. Aurient Traders, Fairford, Gloucostershire, 2001

[Rü85] Rüttinger, J.: *Die drei Magier*. Drei Magier Spiele GmbH, Uehlfeld, 1985

[Rei80] Reiter, R.: A Logic for Default Reasoning. In: *Artificial Intelligence* 13 (1980), Nr. 1-2, S. 81–132

[Rie07] Rieck, C.: *Professor Rieck's Spieltheorie-Seite*. `http://www.spieltheorie.de/`. Version: 2007

[Rio87] Riolo, R. L.: Bucket brigade performance: I. Long sequences of classifiers. Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms (pp. 184-195). In: *Hillsdale, New Jersey: Lawrence Erlbaum Assoc.* (1987), S. 184–195

[RN04] Russell, S. J. ; Norvig, P.: *Künstliche Intelligenz*. 2. Aufl. Pearson Studium, 2004

[Rob00] Robbins, Steven: *CppUnit*. `http://sourceforge.net/projects/cppunit`. Version: 2000

[Rud07] Rudolph, G.: *Vorlesungsfolien zu Fundamente der Computational Intelligence*. Wintersemester. Dortmund, 2006/2007

[SB98] Sutton, R. S. ; Barto, A. G.: *Reinforcement Learning: An Introduction*. MIT Press, 1998

[Sch94] Schwab, B.: *AI Game Engine Programming*. 1. Auflage. Quimby Warehouse, 1994

[Sho97] Shoham, Y.: An Overview of Agent-Oriented Programming. In: Bradshaw, Jeffrey M. (Hrsg.): *Software Agents*. AAAI Press / The MIT Press, 1997, Kapitel 13, S. 271–290

[Spo04] Spolsky, J.: *Joel on Software*. 1. Auflage. Computer Bookshops, 2004

[Str06] Strom, R.: Evolving Pathfinding Algorithms using Genetic Programming. In: *Online article* (2006). `http://www.gamasutra.com/features/20060626/strom_01.shtml`

[Swe] Sweeney, T.: *Unreal Networkung Architecture*. `http://unreal.epicgames.com/Network.htm`

[TB06] Thurau, C. ; Brauckhage, C.: Smater Teammates - Applying Hidden Markov Models in Sports Games. In: *Proceedings of the SAB'06 Workshop on Adaptive Approaches for Optimizing Player Satisfaction in Computer and Physical Games* (2006), Oct

[TDNL06] Togelius, J. ; De Nardi, R. ; Lucas, S. M.: Making Racing Fun Through Player Modeling and Track Evolution. In: *Proceedings of the SAB'06 Workshop on Adaptive Approaches for Optimizing Player Satisfaction in Computer and Physical Games* (2006), Oct

[Thi99] Thiele, H.: *Einführung in die Fuzzy-Logik, Vorlesungsfolien.* Sommersemester. Dortmund, 1999

[TLH07] Thompson, T. ; Levine, J. ; Hayes, G.: EvoTanks: Co-Evolutionary Development of GAme-Playing Agents. In: *2007 IEEE Symposium on Computational Intelligence and Games*, 2007, S. 328–333

[WG89] Wilson, S. W. ; Goldberg, D. E.: A Critical Review of Classifier Systems. In: *Proceedings of the 3rd International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers Inc., 1989, S. 244–255

[Wie03] Wiegand, R. P.: *An Analysis of Cooperative Coevolutionary Algorithms*, George Mason University, Fairfax, Virginia, Diss., 2003

[Wik06a] Wikipedia: *Gewöhnliche Strumpfbandnatter — Wikipedia, Die freie Enzyklopädie.* `http://de.wikipedia.org/w/index.php?title=Gew%C3%B6hnliche_Strumpfbandnatter\&oldid=28223175`. Version: 2006

[Wik06b] Wikipedia: *Rauhäutiger Gelbbauchmolch — Wikipedia, Die freie Enzyklopädie.* `http://de.wikipedia.org/w/index.php?title=Rauh%C3%A4utiger_Gelbbauchmolch\&oldid=25757473`. Version: 2006

[Wik07a] Wikipedia: *Alzheimer-Krankheit — Wikipedia, Die freie Enzyklopädie.* `http://de.wikipedia.org/w/index.php?title=Alzheimer-Krankheit\&oldid=37503002`. Version: 2007

[Wik07b] Wikipedia: *Counter-Strike — Wikipedia, Die freie Enzyklopädie.* `http://de.wikipedia.org/wiki/Counter_Strike`. Version: 2007

[Wik07c] Wikipedia: *Pac-Man — Wikipedia, Die freie Enzyklopädie.* `http://de.wikipedia.org/wiki/Pac-Man`. Version: 2007

[Wik07d] Wikipedia: *Teuvo Kohonen — Wikipedia, Die freie Enzyklopädie.* `http://de.wikipedia.org/wiki/Teuvo_Kohonen`. Version: 2007

[Wil94] Wilson, S. W.: ZCS: A Zeroth Level Classifier System. In: *Evolutionary Computation* (1994), S. 1–18

[WJ95] Wooldridge, M. ; Jennings, N. R.: Intelligent Agents: Theory and Practice. In: *Knowledge Engineering Review* 10 (1995), January, Nr. 2, S. 115–152

[Woo02] Wooldridge, M.: *Introduction to MultiAgent Systems.* John Wiley and Sons, 2002

[YH04] Yannakakis, G. N. ; Hallam, J.: Evolving Opponents for Interesting Interactive Computer Games. In: *Proceedings of the 8th International Conference on the Simulation of Adaptive Behavior (SAB'04)* (2004), July

[YH05] Yannakakis, G. N. ; Hallam, J.: A Generic Approach for Generating Interesting Interacive Pac-Man Opponents. In: *Proceedings of the IEEE Symposium on Computational Intelligence and Games* (2005), April

[YI05] Y.L. Ip, A.B. Rad und Y.K. W.: On-line segment-based map building via integration of fuzzy systems and clustering algorithms. In: *Journal of Intelligent and Fuzzy Systems(IOS Press)* 16 (2005), S. 201–212

[YLH04] Yannakakis, G. N. ; Levine, J. ; Hallam, J.: An Evolutionary Approach for Interactive Computer Games. In: *Proceedings of the 2004 Congress on Evolutionary Computation* (2004), June

[Zel94] Zell, A.: *Simulation Neuronaler Netze*. Assison-Wesley, Reading Mass., 1994