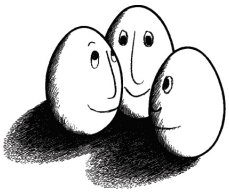
Diplomarbeit

Schwarmintelligenz in Computerspielen - Einsatzgebiet Artificial Life

Sven Becker



Diplomarbeit
am Fachbereich Informatik
der Technischen Universität Dortmund

Dortmund, 5. November 2009

Betreuer:

Prof. Dr. Katharina Morik
Zweit Betreuer Dipl.-Inform. Marco Stolpe

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Thema dieser Arbeit ist der Versuch, die Künstliche Intelligenz (KI) in Videospielen mit Hilfe des Konzeptes der Schwarmintelligenz zu verbessern. Zunächst wird eine Erläuterung und Eingrenzung der Begriffe *Videospiel* und *Spiele-KI* gegeben. Anschließend werden die Problemstellung und das Ziel dieser Arbeit dargestellt. Zum Schluss folgt eine Übersicht über die Gliederung der restlichen Arbeit.

1.1. Videospiele

Viele Menschen spielen gerne Videospiele. Zu diesem Ergebnis kam auch die Studie „Typologie der Wünsche“ im Zeitraum 2006/2007, nach welcher 38,8% der Männer und 22,3% der Frauen ab 14 Jahren Videospiele spielen[1]. Aber was genau sind Videospiele eigentlich?

Um dieser Frage nachzugehen, ist eine Erläuterung des Begriffs *Videospiel* erforderlich. Denn - gerade in Deutschland - werden die Begriffe Video- und Computerspiel nicht einheitlich benutzt. Oft werden die Spiele, die an einer Spielkonsole wie Nintendos „Wii“ oder Sonys „Playstation“ gespielt werden als *Videospiel* bezeichnet, im Gegensatz zum *Computerspiel*, welches an einem PC gespielt wird. Diese Unterscheidung ist jedoch nicht ganz korrekt, denn eigentlich bezeichnet ein Videospiel sowohl ein Computer- als auch Konsolenspiel[2]. Videospiel kann also als Oberbegriff verstanden werden für alle elektronischen Spiele, die auf einem Monitor oder Fernseher angezeigt werden und deren Eingaben über eine Benutzerschnittstelle erfolgen.

Der Aufbau von Videospielen ist in den Grundzügen für alle Spiele relativ ähnlich. So findet sich der Spieler in jedem Spiel innerhalb einer virtuellen Umgebung, der Spielwelt, wieder. Diese Welt kann je nach Spiel sehr unterschiedlich gestaltet sein. Während sie in einem Spiel wie Schach nur aus einem symmetrischen Raster von 8x8 Feldern besteht, gibt es andere Spiele, welche versuchen die natürliche Umgebung, wie wir sie in der Realität vorfinden, abzubilden. Abbildung 1.1 zeigt zwei Beispiele für die unterschiedlichen Umgebungen.

Innerhalb der Spielwelt gibt es verschiedene Arten von Objekten, die sich in ihrem Aussehen, ihren Parametern und Funktionen unterscheiden. Sie können sich gegenseitig, sich selbst oder die Spielwelt durch die ihnen zur Verfügung stehenden Aktionen beeinflussen. Die Regeln, nach denen die Objekte funktionieren, werden während der Entwicklung eines Spiels definiert. Die Entscheidungen, welche Aktionen ein bestimmtes Objekt ausführen soll, hängen davon ab, ob es sich um ein Objekt unter der Kontrolle des Spielers oder einer sog. Spiele-KI handelt. In den meisten Fällen kontrolliert der Spieler

[1] `http://de.statista.com/statistik/diagramm/studie/30395/filter/30005/fcode/1,2/umfrage/videospiele-computerspiele-spielen-in-der-freizeit/`, 30.10.09

[2] Analog zu Computerspiel: ein Spiel, dass an einer Konsole gespielt wird.

Abbildung 1.1.: Beispiele für eine einfache und eine komplexe Spielwelt. Links Chess Titans (2006) von Microsoft, rechts The Elder Scrolls IV: Oblivion (TES 4: Oblivion) (2005) von Bethesda Softworks

ein oder mehrere Objekte aus dieser Spielwelt. Die Art seines Einflusses kann dabei sehr unterschiedlich ausfallen, je nachdem, welche Rolle der Spieler in dem Spiel einnimmt, und reicht von indirekter Beeinflussung durch Belohnung oder Bestrafung von Aktionen (Spieler als Lehrer/Beobachter) über die Steuerung eines Objektes der Spielwelt (Spieler als Protagonist) bis hin zur absoluten Kontrolle über viele Objekte der Spielwelt (Spieler als Gott oder Anführer). Alle anderen, handlungsfähigen und nicht von dem Spieler gesteuerten Objekte unterliegen der Entscheidungsgewalt der Spiele-KI.

Da Videospiele im Allgemeinen interaktiv sind, führen die Eingaben des Spielers zu bestimmten Reaktionen, die für den Spieler wahrnehmbar sind. Bisher erfolgt das Feedback bei Videospielen fast ausschließlich über die Ausgabe von Bildern und Tönen. Die Eingaben werden meistens mit Hilfe der Maus, der Tastatur (bei PC-Spielen) oder einem Gamecontroller (Gamepad, Joystick) getätigt.

Videospiele können neben einen Einspielermodus auch einen Mehrspielermodus bieten, in dem mehrere menschliche Spieler mit- oder gegeneinander antreten. In vielen Spielen kommt es auch oft zur Konfrontation zwischen zwei oder mehreren von der Spiele-KI gesteuerten Objekten mit gegensätzlichen Zielen.

Im folgenden Abschnitt soll geklärt werden, was unter Spiele-KI verstanden wird und wie sie sich von der KI im Allgemeinen unterscheidet.

1.2. Spiele-KI in Videospielen

Wenn man sich mit Videospielen beschäftigt, stößt man früher oder später auf den Begriff der KI oder der Spiele-KI. Oft wird sie im Vorfeld eines Spieles (von den Entwicklern) gelobt, im Nachhinein von den Spielern kritisiert. Es wird viel darüber geredet, aber was

genau ist die Spiele-KI?

Ein Indiz dafür, dass diese Frage nur sehr selten gestellt wurde, findet man in dem Vortrag „#define GAME_AI“ von Steve Rabin, welcher auf der Game Developers Conference (GDC) 2009 in San Francisco gehalten wurde. Steve Rabin ist der Editor der erfolgreichen „AI Game Programming Wisdom“ Serie, welche im Zusammenhang mit Spiele-KI oft erwähnt wird. Um die Schwierigkeit der Frage nach der Spiele-KI hervorzuheben, stellte Rabin die Anwesenden vor ein paar Fragen. Sie sollten sich überlegen, welche Situationen sie als Aspekte der Spiele-KI ansehen würden und welchen nicht:

Szenario 1:

- Sind Strategien in Echtzeit-Strategiespielen (Real-time strategy, kurz RTS) KI?
- Ist die Auswahl von zufälligen Strategien KI?
- Was ist, wenn die Strategie immer die gleiche ist?
- Was ist, wenn die Strategie ein Skript ist?
- Was ist, wenn die Strategie ein Skript ist, das keine Verzweigungen hat?

Szenario 2:
Ist KI:

- einen Weg durch ein Gebäude finden,
- den Spieler abfangen,
- sich in Richtung des Spielers bewegen,
- sich überhaupt bewegen,
- eine Gehen- oder Hinken-Animation auswählen,
- eine zufällige Idle-Animation auswählen?

Anhand der Fragen erkennt man schnell, dass es selbst den Entwicklern schwer fällt, eine Grenze zwischen Spiele-KI und beispielsweise der Spielphysik[3] zu ziehen. Rabin verweist darauf, dass die Spiele-KI viele andere Bereiche wie die eben genannte Spielphysik, aber auch Bereiche wie das Spieldesign, Animationen und Spieleraktionen tangiert (Abbildung 1.2).

Nach Rabin stellen nicht nur alle Techniken, die innerhalb eines Videospiels dazu benutzt werden um eine Illusion von intelligenten Verhaltensweisen der Nicht-Spieler-Charaktere (NSCs)[4] zu erzeugen, die Spiele-KI dar, sondern auch die Techniken, welche Spielinhalte generieren, die normalerweise von den Entwicklern erstellt werden, wie zum Beispiel Landschaften [Rabin, 2009].

[3] Unter Spielphysik wird hier die Simulation von physikalischen Gesetzmäßigkeiten in einem Videospiel verstanden.

[4] Ein NSC ist eine vom Computer gesteuerte Figur innerhalb der Spielwelt.

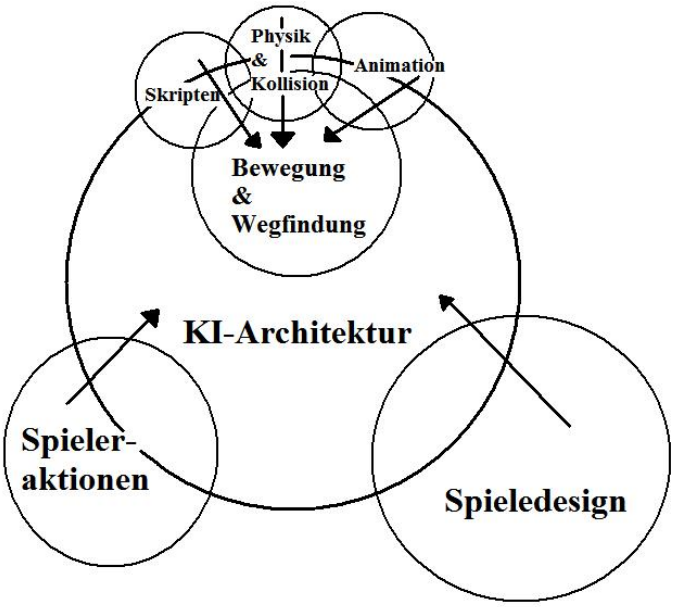


Abbildung 1.2.: Spiele-KI und ihre angrenzenden Bereiche

1.2.1. Unterschied zur klassischen KI

Aus der Definition von Rabin geht hervor, dass die Spiele-KI sich zwar der Methoden der allgemeinen KI bedient, sich jedoch auch von ihr unterscheidet. Die Spiele-KI beschreibt eine Sammlung von vielen verschiedenen Algorithmen, die auch aus anderen Teilgebieten stammen, um ein völlig anderes Ziel zu erreichen. Die klassische KI beschäftigt sich meist mit schweren Problemen wie der Simulation von kognitiven Fähigkeiten der Menschen oder dem Verstehen von natürlichen Sprachen (dazu später mehr). Die Spiele-KI hingegen hat das Ziel der Unterhaltung. Es geht weniger darum, ein Problem perfekt zu lösen oder, im Falle eines Videospiels, zu gewinnen. Die Spiele-KI soll den Eindruck von Intelligenz oder realistischem Verhalten erwecken, muss aber selber nicht intelligent sein. So ist es auch üblich, dass es der Spiele-KI gestattet wird zu schummeln, wenn es dem Zweck der Unterhaltung dient. Anders herum ist es auch nötig, die Fähigkeiten der Spiele-KI künstlich zu beschneiden, denn niemand könnte, zum Beispiel in einem First-Person-Shooter (FPS), die Spiele-KI an Zielgenauigkeit und Geschwindigkeit übertreffen, da das Zielen bei der Spiele-KI nicht auf analoge Weise vonstatten geht wie bei einem Menschen, sondern die exakte Position und die Winkel einfach innerhalb weniger Millisekunden berechnet werden können.

Nach Ansicht von Paul Tozour sollte die Spiele-KI nicht als *KI*, sondern als *Agentendesign* oder *Verhaltensmodellierung* bezeichnet werden. Der Begriff der KI wecke andere Erwartungen als das, was die Spiele-KI tatsächlich leiste [Tozour, 2002].

1.3. Problemstellung und Ziel dieser Arbeit

Wie zu Anfang erwähnt, faszinieren Videospiele schon seit Jahren viele Menschen. Dennoch haben Videospiele ein wachsendes Problem. Bereits seit Jahren hinkt die Spiele-KI der Entwicklung anderer Bereiche wie zum Beispiel dem der Grafik in puncto Realismus deutlich hinterher. Während vor etwas mehr als einem Jahrzehnt die Spielfiguren noch nicht einmal aus Polygonen, sondern aus Bitmaps bestanden, hat man heutzutage schon einen sehr beachtlichen Detailgrad an Realismus erreicht, wie Abbildung 1.3 deutlich zeigt.

Abbildung 1.3.: Links Eye of the Beholder (1990) von Westwood Studios und rechts TES 4: Oblivion (2005) von Bethesda Softworks

Hinsichtlich der Spiele-KI gibt es seit langer Zeit so gut wie keine Weiterentwicklung, mit der Folge, dass sie auf dem Niveau von vor etwa zehn Jahren stehen geblieben ist. Das trifft zwar nicht auf alle Bereiche der Spiele-KI zu, ein Bereich ist davon jedoch besonders stark betroffen. Die Rede ist von Artificial Life (A-Life), der Simulation von Lebewesen in der Art, dass ihre Verhaltensweisen natürlich wirken. Eine ausführlichere Beschreibung von A-Life folgt in Kapitel 2. Da es sehr viele verschiedene Arten von Spielen gibt, fällt die unzureichende Entwicklung in diesem Bereich unterschiedlich stark auf. Während in Wirtschaftssimulationen oder Sportspielen A-Life selten Verwendung findet, stellt sich die Situation in Rollenspielen ganz anders dar. Hier sind die Lebewesen und ihre Verhaltensweisen ein wichtiger Aspekt um die Illusion einer anderen Welt aufzubauen, was als das Ziel dieser Spiele angesehen werden kann. Die atmosphärische Tiefe dieser Illusion hängt u.a. von der A-Life-Umsetzung ab. In den meisten Rollenspielen fehlt eben dieser Aspekt jedoch vollständig. Viele Welten wirken trotz ihrer optisch ansprechenden Umgebung leer und unbelebt. Das liegt nicht etwa daran, dass der Spieler allein in dieser Welt umher wandert, sondern vielmehr daran, dass sich die Bewohner der Welt als leblose Dekorationsobjekte entpuppen. Häufig stehen sie einfach nur regungslos herum oder rufen immer wieder die selben, vorgefertigten Animationen ab. Damit sieht die Welt zwar nicht leer aus, sie „fühlt“ sich jedoch leer an. Denn Lebewesen ohne eigenen Antrieb oder Motivation, egal ob Mensch oder Tier, wirken wenig realistisch oder lebendig, was sich sehr negativ auf die für das Spiel wichtige Atmosphäre auswirkt.

Diese Arbeit visiert genau dieses Problem an. Es soll untersucht werden, ob es möglich ist mit Hilfe des Konzepts der Schwarmintelligenz (SI) diesen Missstand zu beheben und

ihr damit ein neues Anwendungsgebiet in Videospielen zu eröffnen: Artificial Life.

Dazu soll ein Editor entwickelt werden, welcher es ermöglichen soll das Verhalten eines NSCs zu definieren und (halb)automatisch zu verbessern. Der Fokus soll dabei nicht, wie bei der klassischen KI, auf der Intelligenz des Einzelnen, sondern - entsprechend dem Paradigma der Schwarmintelligenz - auf dem Zusammenspiel der Interaktionen der Individuen liegen. In der Grundversion sollen fünf verschiedene NSC-Gruppen implementiert werden: Bauern, Soldaten, Goblins, Raubtiere und Beutetiere. Jede Gruppe hat ihre eigenen Ziele, die im Konflikt mit den Zielen der anderen Gruppen stehen können. So ist das Ziel der Goblins, wehrlose Bauern zu überfallen, das Gegenteil vom Ziel der Wachen, eben jene zu beschützen.

Dazu wird diese Arbeit in drei Schritte eingeteilt.

Der *erste Schritt* besteht darin einen Simulator zu entwickeln, welcher als Basis für die Tests der KI benutzt werden kann. Dieser wird sehr nahe an das Spiel Neverwinter Nights (NWN) (2002) von BioWare[5] angelehnt sein.

Der *zweite Schritt* sieht die Entwicklung eines Editors vor, welcher dazu dienen soll, die Regeln und Regelsätze der einzelnen NSC-Gruppen zu erstellen, zu verwalten und zu optimieren. Innerhalb des Editors werden die Simulationen gestartet, welche Daten über die aktuellen Regeln sammeln. Diese Daten sollen später bei der Verbesserung der Regelsätze behilflich sein.

Im *dritten Schritt* sollen mit Hilfe des Editors und Simulators für die fünf NSC-Gruppen geeignete Beispielregelsätze entworfen werden, welche dem Ziel, realistische und dynamische Verhaltensweisen zu simulieren, gerecht werden.

1.4. Gliederung

Zunächst wird in *Kapitel 2* auf die Grundlagen der künstlichen Intelligenz eingegangen. Dabei geht es sowohl um die KI im Allgemeinen, als auch um die speziellen Teilgebiete der KI, welche für diese Arbeit relevant sind.

Kapitel 3 beschäftigt sich mit der künstlichen Intelligenz in Videospielen. Es wird ein Einblick in die Vielfalt der verschiedenen Aufgaben der Spiele-KI gegeben und darein wie diese Aufgaben bisher gelöst werden. Anschließend wird kurz auf die geschichtliche Entwicklung der Spiele-KI, so wie auf die Gründe für die Unterschiede zwischen Theorie und Praxis eingegangen.

Ein neuer Ansatz für eine Spiele-KI im Rahmen von A-Life, die Idee dieser Arbeit, wird in *Kapitel 4* beschrieben. Dabei wird auf das zu Grunde liegende System und die Methoden zur Verbesserung des Systems eingegangen.

Kapitel 5 beschäftigt sich mit dem Simulator und dem Regeleditor, die im Rahmen dieser Arbeit implementiert wurden. Es werden auch die Regeln vorgestellt, die innerhalb dieser Arbeit erarbeitet wurden.

In *Kapitel 6* werden abschließend die Ergebnisse und Erkenntnisse dieser Arbeit zusammengefasst.

[5] NWN ist ein Rollenspiel aus dem Jahre 2002 und bietet dank des umfangreichen Editors (Aurora-Toolset) genügend Flexibilität, um eigene Programminhalte herzustellen. So stellt das Toolset einen Editor und einen Compiler für die stark an C(++) angelehnte Skriptsprache NWN-Script zur Verfügung, welche ausreichend ausdrucksstark ist, um eine neue KI auf Basis der SI zu implementieren.

2. Grundlagen der Künstlichen Intelligenz

Der Fokus dieser Arbeit liegt auf der künstlichen Intelligenz, die zur Steuerung von Softwareagenten (hier: NSCs) in Videospielen eingesetzt werden soll.

Nach [Walter Brenner, 1998] ist ein *Softwareagent* ein Programm, welches die ihm zugeteilten Aufgaben (teilweise) autonom erledigen und sinnvoll mit seiner Umgebung interagieren kann. Es gibt mehrere Typen von Softwareagenten:

- *Reaktive Agenten*: Dieser Typ reagiert nur auf Veränderungen seiner Umwelt, um dann mit Hilfe von Regeln zu entscheiden, welche Aktion ausgeführt werden soll.
- *Beobachtende Agenten*: Dieser Typ ist ein *reaktiver Agent*, welcher sich die Veränderungen der Umwelt und die Auswirkungen seiner Aktionen merkt. Mit Hilfe dieser Informationen werden seine Regeln angepasst.
- *Zielorientierte Agenten*: Dieser Typ versucht das ihm vorgegebene Ziel zu erreichen. Er speichert, wie der *beobachtende Agent*, Informationen über die Umwelt und die Auswirkungen seiner Aktionen und besitzt die Fähigkeit nicht-triviale Ziele durch Planung von Aktionsfolgen zu erreichen.
- *Nutzenorientierte Agenten*: Dies ist eine erweiterte Form des *zielorientierten Agenten*. Er bildet Zustände seiner Umgebung auf Zahlen ab, die den Nutzen für den Agenten darstellen, um so immer die Aktion durchzuführen, die für ihn den größten Nutzen bringt. Es fließen auch Faktoren wie die Unsicherheit, ob ein Ziel überhaupt erreicht werden kann, in die Berechnung des Nutzens mit ein.

Zu Beginn nun ein Überblick über das Thema KI im Allgemeinen, ohne besonderen Wert auf den Bezug zu Videospielen zu legen. Anschließend wird auf einzelne Teilgebiete der KI, welche für diese Arbeit von Interesse sind, näher eingegangen. Zu diesen Gebieten zählen:

- *Künstliches Leben* (KL)
- *Schwarmintelligenz* (SI)
- *Evolutionäre Algorithmen* (EA)

Wie zu Anfang schon erwähnt, beschäftigt sich diese Arbeit mit der Simulation von Lebewesen. Der Teilbereich der KI, der sich mit dieser Simulation beschäftigt, wird *Künstliches Leben* (KL, bzw. *Artificial Life* kurz, A-Life) genannt. Diese Simulation soll mit Hilfe von Methoden der SI durchgeführt werden. Der Lösungsansatz wird also einen Bottom-Up, und nicht wie in der klassischen KI üblich, einen Top-Down Ansatz verfolgen. Evolutionäre Algorithmen werden später eingesetzt um die Parameter der NSCs zu verbessern.

2.1. Künstliche Intelligenz

Der Begriff *Artificial Intelligence* (AI) wurde 1955 von John McCarthy geprägt, welcher ein Jahr später eine Konferenz in Dartmouth mit eben diesem Titel veranstaltete, an der renommierte Wissenschaftler wie Marvin Minsky, Nathaniel Rochser, Claude Shannon, Allan Newell und Herbert Simon teilnahmen.

Auch wenn es schon vor dieser Konferenz einige Schritte in diese Richtung gegeben hat, wie zum Beispiel Alan Turings Aufsatz von 1950 „Computing Machinary and Intelligence" [Turing, 1950], in welchem er auch seinen nach ihm benannten „Turing-Test" (dazu später mehr) beschrieb, so gilt die Konferenz in Dartmouth dennoch als Geburtsstunde der KI.

Die Definition von *Künstlicher Intelligenz* stellt sich aber noch heute, über 50 Jahre nach der Konferenz, als sehr schwierig heraus, da es keine genaue, alle Aspekte umfassende Definition von Intelligenz gibt. Es wurden viele Versuche unternommen um den Begriff der Intelligenz zu definieren. Jedoch gibt es nach Ansicht von Irrgang und Klawitter keine erschöpfende Definition des Begriffes, die seine gesamte Komplexität erfassen würde [B. Irrgang, 1990].

Nach Görz

> *„besteht weitgehender Konsens darüber, dass Intelligenz zu verstehen ist als Erkenntnisvermögen, als Urteilsfähigkeit, als Erfassen von Möglichkeiten, aber auch als das Vermögen, Zusammenhänge zu begreifen und Einsichten zu gewinnen."* [G.Görz, 2003]

Ausgehend von diesem Konsens stellt sich im Anschluss die Frage, was dann unter Künstlicher Intelligenz zu verstehen ist. Es sei darauf hingewiesen, dass die Disziplin der KI nicht nur einen Teilbereich der Informatik darstellt, sondern auch eng verknüpft ist mit anderen Disziplinen wie beispielsweise der Philosophie, Psychologie, den Neurowissenschaften und der Linguistik. Diese Arbeit konzentriert sich jedoch auf die informationstechnische Komponente. In diesem Bereich haben sich im Laufe der Zeit zwei verschiedene Vorgehensweisen herausgebildet, wie man den Begriff der KI definieren könnte. Zum einen gibt es die *starke KI* (oder *Artificial General Intelligence*, kurz AGI), welche den Menschen als Vergleich heranzieht und von der Vision des Menschen als Maschine geleitet wird. Es geht primär um die Nachbildung von menschlicher Intelligenz als solcher mit all ihren Aspekten mit dem Ziel, Computer in die Lage zu versetzen, alle Aufgaben, die ein Mensch erledigen könnte, zu bewältigen.

Auf der anderen Seite gibt es die *schwache KI*, welche das Ziel hat, konkrete Probleme zu lösen, zu deren Lösung im Allgemeinen Verständnis Intelligenz benötigt wird. Dabei wird der Anspruch auf ein künstlich erschaffenes Bewusstsein oder eine allgemeine Intelligenz, die der des Menschen ähnelt, aufgegeben. Hierbei versucht man, die menschliche Herangehensweise bei der Problemlösung zu imitieren [Uwe Lämmel, 2008].

In den Anfängen der KI war man zunächst noch sehr optimistisch, was die weitere Entwicklung der KI, besonders der *starken KI*, betrifft. So ließ Herbert Simon 1957 verlauten, dass es innerhalb der nächsten zehn Jahre Computern gelingen werde, wertvolle Musikstücke zu komponieren, einen wichtigen mathematischen Satz zu beweisen und Schachweltmeister zu werden [Stuart J. Russell, 2003a].

Wie wir wissen, ist nichts davon (rechtzeitig) eingetroffen. Jedoch gelang es tatsächlich einem Computer den damals amtierenden Schachweltmeister Garri Kimowitsch Kasparow mit 3,5:2,5 zu schlagen, allerdings erst im Jahre 1998. Der Sieg von IBMs „Deep Blue“ war auch nicht seiner KI, sondern der schieren Rechenkraft des Supercomputers zuzuschreiben, da dieser durchschnittlich 126 Millionen Stellungen pro Sekunde berechnen konnte [Campbell et al., 2002].

Heutzutage ist es sehr umstritten, ob die Realisierung von starker KI jemals gelingen wird. Während die einen behaupten, der Nachbau der menschlichen Intelligenz sei unmöglich, sind andere der Meinung, dass es sich bei der menschlichen Intelligenz nur um einen, wenn auch sehr komplexen, biochemischen Vorgang handele, der sich simulieren lasse.

Alan Turing schlug 1950 den nach ihm benannten „Turing-Test“ vor. Dieser Test sollte dazu dienen herauszufinden, ob eine Maschine intelligent ist oder nicht. Dabei werden von einem Menschen beliebige Fragen gestellt, die von einem anderen Menschen oder von einer Maschine beantwortet werden. Der Fragesteller weiß dabei nicht, welche Frage von wem beantwortet wurde und muss entscheiden, ob die Antworten von dem Menschen oder von der Maschine gegeben wurden. Kann er keinen Unterschied feststellen, so gilt die Maschine als intelligent. Bisher hat keine Maschine diesen Test bestanden.

Seit den Anfängen der KI in den 50er Jahren gab es jedoch viele Erfolge im Bereich der *schwachen KI* und es entwickelten sich mehrere Teilgebiete der KI, die sich mit den einzelnen Problemen beschäftigen. Einige der Teilgebiete und Probleme sind *Robotik*, *Sprachverstehen*, *Maschinelles Lernen*, *Wahrnehmung*, *Theorembeweisen*, *Expertensysteme*, *Automatisches Programmieren* und *Spieltheorie*.

Da der Begriff der *Spieltheorie* den Anschein erweckt, für das Thema dieser Arbeit interessant zu sein, soll er kurz erläutert werden. Bei der Spieltheorie geht es darum die Entscheidung zu treffen, die den größten Nutzen für den Spieler bringt. Dabei kommt es aber, anders als bei der Entscheidungstheorie, nicht nur auf die eigene Entscheidung, sondern auch auf die Entscheidungen anderer Akteure an. Als Grundvoraussetzung wird angenommen, dass alle Beteiligten rational handeln. Zu den Anfängen der Spieltheorie wurden verstärkt Spiele wie Schach, Dame und Mühle untersucht. Jedoch beschränkt sich das Anwendungsgebiet der Spieltheorie nicht auf Spiele, sondern kann auf viele Situationen aus der Gesellschaft oder Wirtschaft, wie zum Beispiel Preisgestaltung oder Rabattaktionen, angewendet werden.

Ein sehr bekanntes Beispiel für die Spieltheorie ist das *Gefangenendilemma*: Zwei Akteure Alice und Bob werden von der Polizei beschuldigt, einen Bankraub begangen zu haben und werden in getrennten Zellen verhört. Beide haben die Wahl entweder mit der Polizei zu kooperieren oder zu schweigen. Je nachdem wie sich die beiden entscheiden, fallen die Urteile anders aus und beide haben das Ziel ihre Haftstrafe zu minimieren. Insgesamt gibt es drei verschiedene Ausgänge:

- Beide schweigen, dann werden sowohl Alice als auch Bob für zwei Jahre inhaftiert.
- Einer von beiden koopertiert und der andere schweigt, dann wird derjenige, welcher kooperiert hat freigesprochen und der andere für 15 Jahre inhaftiert.
- Kooperieren beide mit der Polizei, werden sie beide für zehn Jahre inhaftiert.

An diesen drei Ausgängen sieht man sehr deutlich, dass die eigene Entscheidung allein nicht das Ergebnis bestimmt, sondern dass es auch stark auf die Entscheidungen der anderen Akteure ankommt. Dies ist ein Aspekt, den man bei der Schwarmintelligenz wiederfinden kann (s. Kapitel 2.2.2). Andere Askepte der Spieltheorie berühren das Thema dieser Arbeit nicht und sollen hier nicht weiter erläutert werden.

2.2. Künstliches Leben

Artificial Life (A-Life) ist ein Teilbereich der KI. Er beschreibt die Forschung an von Menschen erstellten Systemen, die grundlegende Eigenschaften des Lebens beinhalten. Christopher G. Langton, der Initiator der ersten Konferenz über Artificial Life, die 1987 in Los Alamos statt fand, definierte diesen Forschungszweig so:

> *„A field of study devoted to understanding life by attempting to abstract the fundamental dynamical principles underlying biological phenomena, and recreating these dynamics in other physical media, such as computers, making them accessible to new kinds of experimental manipulation and testing.“* [Langton, 1992]

Es gibt zwei große *Ziele* im Forschungsbereich A-Life.

1. Die Erklärung und Untersuchung biologischer Zusammenhänge mit Hilfe von Simulationen des natürlichen Lebens.

2. Die Lösung von konkreten Anwenderproblemen durch die Anwendung biologischer Konzepte.

In den ersten Bereich fallen auch die Versuche, künstliche Systeme zu erschaffen, die als *lebend* bezeichnet werden können. Jedoch gibt es mit dem Begriff *Leben* ähnliche Probleme wie mit dem Begriff *Intelligenz* bei der KI. Es stellt sich als sehr schwer heraus, eine Definition für *Leben* zu finden und bisher gibt es keine, von allen akzeptierte Definition von dem was Leben ist. Die wesentlichen Eigenschaften, die ein System besitzen muss, damit es als lebend bezeichnet werden kann, sind nach [Brockhaus, 2003] Energie-, Stoff- und Informationsaustausch, Wachstum, Fortpflanzung, Veränderung des Erbguts, und Reaktion auf Veränderungen der Umwelt.

Des Weiteren fehlt es an einem Gegenstück zu Alan Turings „Turing-Test“ für künstliches Leben, mit dessen Hilfe man evaluieren könnte, ob ein System als *lebend* bezeichnet werden kann oder nicht.

In [Kinnebrock, 1996] wurden Merkmale zusammengestellt, die ein System besitzen muss, damit es als künstliches Leben bezeichnet werden kann.

- Das System muss Informationen aufnehmen und verarbeiten können.
- Das System muss sich in einer Umwelt befinden und mit ihr interagieren können.
- Das System muss Verhaltensregeln besitzen, die die Interaktionen bestimmen.
- Diese Verhaltensregeln müssen ein dynamisches Verhalten erzeugen.

In dem zweiten Bereich, der Lösung von Anwendungsproblemen, gab es bisher einige Erfolge zu verbuchen. So sind zahlreiche, naturinspirierte Algorithmen zur Lösung von schwierigen Problemen entstanden. Beispiele hierfür sind die, später im Detail erläuterten, Ameisenkolonien und die Partikelschwarmoptimierung.

KL-Methoden unterscheiden sich grundlegend von denen aus der KI. Das Prinzip der KI ist es Systeme zu entwickeln, die wissen wie sie in allen Situationen reagieren sollen. Um dies zu erreichen, muss das zu lösende Problem zunächst gründlich analysiert werden, um anschließend ein Konzept zu entwerfen, welches alle Eventualitäten abdeckt. Dieser Ansatz wird auch als Top-Down Methode bezeichnet und ist charakteristisch für die klassische KI. Ausgehend von einem übergeordnetem Ziel wird versucht, die einzelnen Aufgaben in kleine Teile zu zerlegen, um diese Teilaufgaben dann effizient zu lösen. In der KL verfolgt man den umgekehrten Weg. Ausgangspunkt sind sehr einfache Verhaltensregeln eines Systems, die durch ihre Interaktionen ein globales Verhalten erzeugen. Dieses als emergent bezeichnete Globalverhalten enthält sehr häufig Elemente, die zur Entwicklungszeit nicht vorhergesehen waren.

Der Vor- und gleichzeitige Nachteil dieser Methode ist es, dass man nur sehr wenig Wissen über das zu lösende Problem besitzen muss und keine umfassende Analyse des Problems nötig ist. Allerdings ist es sehr schwer nachzuvollziehen, warum und wie diese Interaktionen zwischen den einzelnen Verhaltensregeln funktionieren. Dies macht es sehr schwierig vorher zu sagen, welche Regeln notwendig sind, um ein bestimmtes Gesamtverhalten zu erzielen. So gelingt dies häufig nur durch Ausprobieren verschiedener Regeln und anschließende Beobachtung, ob das auftretende Verhalten dem gewünschten entspricht. Auf der anderen Seite gelten KL-Methoden generell als flexibler als KI-Methoden. KI-Methoden stoßen im Allgemeinen sehr schnell an ihre Grenzen, sobald Situationen eintreten, welche während der Entwicklung nicht berücksichtigt wurden. KL-Methoden sind aufgrund ihres Bottom-Up Ansatzes resistenter gegen außergewöhnliche, unvorhergesehene Situationen. Dieser Vorteil wird jedoch meinst durch einen Verlust an *Intelligenz* gegenüber den KI-Methoden erkauft [Kinnebrock, 1996].

2.2.1. Evolutionäre Algorithmen

Die Informationen über die *Evolutionären Algorithmen (EAs)* wurden [Weicker, 2007] entnommen. Bei den EAs handelt es sich um eine probabilistische, populationsbasierende Methode für die Berechnung von Näherungslösungen nahezu beliebiger Optimierungsprobleme. Die Anfänge gehen bis in die 1950er Jahre zurück und wurden das erste Mal von Friedman 1956 eingesetzt um Schaltkreise zu optimieren. Allerdings wurden erst in den 1960er die Grundsteine für die Algorithmen gelegt, die bis heute das Forschungsfeld bestimmen.

Seitdem sind viele Varianten der evolutionären Algorithmen veröffentlicht worden. Zu diesen zählen unter anderen:

- Genetische Algorithmen
- Evolutionäres Programmieren
- Evolutionsstrategien

- Genetisches Programmieren

Im folgenden soll ein Überblick über das Prinzip der EAs gegeben werden, jedoch keine Beschreibung der konkreten Varianten. Eine Ausnahme bilden die Mutationsvarianten der 1/5-Regel und die selbstadaptive Schrittweitenanpassung der Evolutionsstrategien. Letztere wird in dieser Arbeit Verwendung finden.

Vorbild und Idee

Die EAs bedienen sich des biologischen Vorbilds der natürlichen Evolution, bei der die Gene eines Individuums verändert werden. Gene bestimmen die Ausprägungen von Merkmalen eines Lebewesens. Änderungen an diesen Merkmalen können die Überlebensfähigkeit eines Individuums beeinflussen.

Die Idee der EAs besteht darin, eine Menge von Lösungskandidaten (Individuuen), Population genannt, für ein Optimierungsproblem einer simulierten Evolution zu unterziehen und dabei im Laufe der Zeit immer besser angepasstere Lösungskandidaten hervorzubringen. Dabei werden die Gene meist durch Zahlen aus dem $\mathbb{N}$ oder $\mathbb{R}$ repräsentiert und die Selektion, welche in der Natur von der Umwelt vorgenommen wird, wird von einer Güte- oder Fitnessfunktion übernommen. Wie beim natürlichen Vorbild werden gute Lösungen ihre Eigenschaften mit einer größeren Wahrscheinlichkeit weitergeben als schlechte Lösungen.

Einsatzgebiet und Anforderungen

Das primäre Einsatzgebiet der EAs sind Optimierungsprobleme, die nicht in akzeptabler Zeit exakt und deterministisch berechnet werden können, also im Grunde genommen die Klasse der np-harten Probleme. Ein großer Vorteil von EAs liegt in den wenigen Anforderungen, die erfüllt sein müssen, damit sie angewendet werden können. Zum einen muss der Suchraum des Problems mit Hilfe eines Rechners darstellbar sein und zum anderen muss es eine Fitnessfunktion geben, so dass jedem Lösungskandidaten eindeutig eine Güte zugewiesen werden kann, die ein Maß für seine Qualität darstellt. Je höher der Wert, desto besser ist der Lösungskandidat.

Simulierte Evolution

Die Natur bedient sich bei der Anpassung von Lebewesen an die Umwelt mehrerer Evolutionsfaktoren. Die wichtigsten hier vorgestellten und verwendeten sind:

- Mutation
- Rekombination
- Umweltselektion

Nachdem die Initialisierung der Population und deren Bewertung erfolgt ist, beginnt der *Evolutionszyklus*, der aus sechs Schritten besteht (die Definitionen der Evolutionsfaktoren und Parameter folgt im Abschnitt 2.2.1):

1. *Terminierungsbedingung prüfen:* Zu Beginn eines neuen Zyklus wird geprüft, ob die Bedingungen für den Abbruch des Algorithmus erfüllt sind oder nicht. Die häufigsten Bedingungen sind das Erreichen einer gewissen Fitness, keine Verbesserung der Fitness in den letzten x Zyklen oder die Überschreitung einer vorher festgelegten Anzahl an Iterationen. Ein oder mehrere Abbruchkriterien sind notwendig für EAs, da sie sonst nicht terminieren würden.

2. *Elternselektion:* Als nächstes werden die Paarungen von μ Eltern für die Rekombination von λ vielen Nachkommen ausgewählt. Die Selektionsmethode kann, je nach verwendetem Algorithmus, variieren. Die häufigsten Varianten sind die gleichverteilte Elternselektion und probabilistische Elternselektion auf Basis der Fitnesswerte. Dabei können, anders als in der Natur üblich, mehr als zwei Eltern für eine Rekombination ausgewählt werden.

3. *Rekombination:* Die von der Elternselektion ermittelten Paarungen werden nun mit einer gewissen Wahrscheinlichkeit einer Rekombination unterzogen, welche eine bestimmte Anzahl von Kindern erzeugt. Falls ein EA auf die Rekombination verzichtet, werden die Kinder mittels einfacher Duplizierung erzeugt. Am Ende der Rekombination und/oder Duplizierung sind λ viele Nachkommen entstanden.

4. *Mutation:* Im Anschluss findet die Mutation der λ vielen Nachkommen statt. Es gibt verschiedene Arten von Mutationen, von denen einige später genauer erklärt werden.

5. *Bewertung:* Die λ vielen Nachkommen werden nun mit Hilfe einer Fitnessfunktion bewertet.

6. *Umweltselektion:* Die Umweltselektion wählt nun, je nach Algorithmus, μ viele Individuen, entweder nur aus den λ Nachkommen oder den μ Eltern und λ Nachkommen, aus.

Definitionen und Parameter: Im Folgenden werden die bei EAs verwendeten Evolutionsfaktoren näher beschrieben. Der Fokus der Beschreibung liegt auf den von EAs verwendeten Varianten und nicht auf den biologischen Vorbildern. Zudem werden weitere Definitionen und Parameter vorgestellt.

Parameter: Hier werden einige der Parametereinstellungen und Variablen der EAs bzw. deren Methoden vorgestellt:

- $\mu \in \mathbb{N}$: Größe der Elterngeneration
- $\lambda \in \mathbb{N}$: Größe der Nachkommengeneration
- σ: Standardabweichung der Mutationsschrittweite
- Ω: Phänotyptischer Suchraum
- G: Genotypischer Suchraum

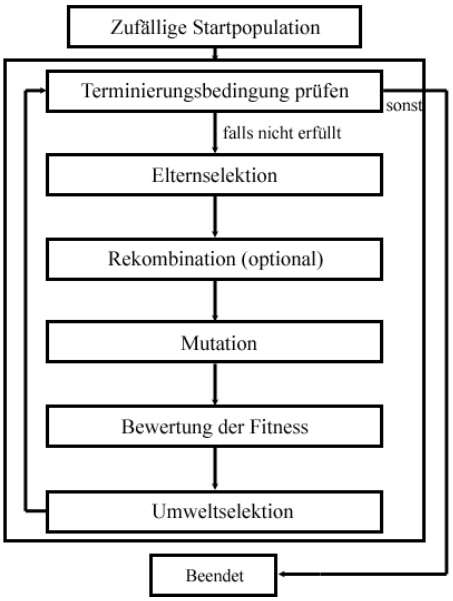


Abbildung 2.1.: Evolutionszyklus

- id: Identische Funktion
- PR[*]: Auswahlwahrscheinlichkeit
- $\geq$: Besser- oder Gleichrelation bezüglich der Güte
- $A^{(i)}$: i-tes Individuum einer Population

Optimierungsproblem: Wie schon zu Anfang erwähnt, können mit Hilfe von EAs nahezu beliebige Optimierungsprobleme gelöst werden. Da die Anforderungen sehr unterschiedlich ausfallen können, soll hier eine sehr allgemeine Definition von Optimierungsproblemen gegeben werden.

Definition 1 *Ein Optimierungsproblem (Ω, f, $\geq$) ist gegeben durch einen Suchraum Ω, eine Bewertungsfunktion f: $\Omega \to \mathbb{R}$, die jedem Lösungskandidaten einen Gütewert zuweist, sowie eine Vergleichsrelation $\geq \in \{\leq, \geq\}$. Dann ist die Menge der globalen Optima $H \subseteq \Omega$ definiert als*

$$H = \{x \in \Omega | \forall x' \in \Omega : f(x) \geq f(x')\} \tag{2.1}$$

Dekodierungsfunktion: Um sich nicht um die verschiedenen Darstellungsarten von Optimierungsproblemen und Lösungskandidaten sorgen zu müssen, benutzt man die Dekodierungsfunktion *dec*, welche die Struktur des Suchraums Ω, dem Phänotyp, von der Darstellung, dem Genotyp G, der Lösungskandidaten trennt.

Definition 2 *Eine Dekodierungsfunktion dec: $G \to \Omega$ ist eine Abbildung vom Genotyp G auf den Phänotyp bzw. Suchraum Ω.*

In vielen Fällen wird jedoch keine Kodierung des Problems vorgenommen, so dass $G = \Omega$ gilt und damit auch $dec = id$. Im Folgenden wird, sofern nicht anders angekündigt, keine Kodierung des Problems verwendet.

Individuum: Ein Individuum A besteht aus mehreren Informationen. Zum einen aus seinem Genotyp A.G $\in$ G , also der Position von A innerhalb des Suchraumes Ω (mit G = Ω). Zum anderen aus einer Menge von Zusatzinformationen A.S $\in$ Z, dem Raum aller möglichen Zusatzinformationen. Zusatzinformationen sind beispielsweise Parametereinstellungen für die Evolutionsfaktoren und können sich, so wie der Genotyp A.G, über die Zeit durch Mutation und/oder Rekombination ändern. Zuletzt wird noch die Güte oder Fitness von A als A.F gespeichert.

Definition 3 *Ein Individuum A ist ein Tupel (A.G, A.S, A.F) bestehend aus dem eigentlich Lösungskandidaten, dem Genotyp $A.G \in G$, den optionalen Zusatzinformationen $A.S \in Z$ und der Güte $A.F = f(dec(A.G)) \in \mathbb{R}$*

Operatoren: EAs verwenden zwei Arten von Operatoren, die auf Individuen (Lösungskandidaten) einer Population angewendet werden: Mutation und Rekombination.

Mutation: Die Mutation hat die Aufgabe an einem Individuum A eher kleine Veränderungen vorzunehmen. Dabei werden die einzelnen Komponenten von A.G mit einer gewissen Wahrscheinlichkeit verändert[1].

Es gibt verschiedene Möglichkeiten die Mutation in EAs durchzuführen. So kann man zum einen die Mutationshäufigkeit, zum Beispiel 20% der Population in jedem Zyklus, festlegen und anschließend die entsprechende Anzahl an zu mutierenden Individuen auswählen. Auf diese Art und Weise bleibt die Anzahl der Mutationen stets die Gleiche. Eine andere Methode ist es, jedes Individuum einer Population mit einer bestimmten Wahrscheinlichkeit mutieren zu lassen, so dass die Anzahl der Mutationen von Zyklus zu Zyklus variieren kann.

Der wichtigste Parameter der Mutation ist die *Schrittweite* σ. Sie bestimmt die Konvergenzgeschwindigkeit und, falls keine Rekombination verwendet wird, die Fähigkeit lokale Optima zu verlassen. Als Basis werden in der Regel gaußsche Mutationen verwendet, da mit dieser Methode sowohl kleine als auch vereinzelt größere Änderungen auftreten können. Generell gilt: Falls die Suche sich noch nicht den Optima genähert hat, sind hohe Schrittweiten vorteilhaft, damit lokale Optima schnell verlassen werden und sich die Individuen besseren Regionen nähern können. Befindet sich die Suche allerdings schon in der Nähe der Optima, sind kleinere Schrittweiten für die Feinabstimmung die bessere Wahl. A priori ist es sehr schwer einen geeigneten Wert für die Schrittweite anzugeben. Um diesem Problem zu begegnen, wurden verschiedene Methoden zur adaptiven Schrittweitenanpassung entwickelt. Die einfachste Variante sieht eine konstante Abnahme der Schrittweite um den Faktor α pro Generation vor.

$$\sigma' \leftarrow \sigma \cdot \alpha, \text{ mit } 0 < \alpha < 1 \tag{2.2}$$

[1] Beim Menschen beträgt die Wahrscheinlichkeit etwa 10^{10}. Berücksichtigt man die große Anzahl an Genen und deren Bausteine, beträgt die Wahrscheinlichkeit für eine Mutation bei einer Zellteilung etwa 10%.

Bei dieser Variante geht man davon aus, dass sich zu Anfang des EAs die Suche noch weit von den Optima entfernt abspielt und sich diesen im Laufe der Zeit immer weiter annähert. Gleichzeitig wird die Schrittweite σ immer kleiner. Es findet also eine Verschiebung von einer eher zufälligen zu einer lokalen Suche statt. Jedoch kann es bei dieser Methode schnell zu Problemen kommen, falls die Suche nicht so verläuft wie oben angenommen wurde. Zum Beispiel könnte sich die Suche langsamer als erwartet den Optima nähern. Dies hätte zur Folge, dass die Optimierung innerhalb von lokalen Optima gefangen werden könnte, da die Schrittweite nicht mehr ausreicht um diese zu verlassen.

Einen Ausweg aus diesem Problem bietet die *adaptive Schrittweitenanpassung nach der 1/5-Erfolgsregel* nach [Rechenberg, 1973], die bei den Evolutionsstrategien (ES) eingesetzt wird. Bei dieser wird die Schrittweite abhängig vom Verhältnis von erfolgreichen zu nicht erfolgreichen Mutationen innerhalb eines bestimmten Zeitraums verändert. Eine Mutation wird als erfolgreich bezeichnet, falls sie eine Verbesserung der Güte bewirkt hat. Es wird davon ausgegangen, dass um so mehr erfolgreiche Mutationen auftreten, je weiter entfernt von den Optima die Suche stattfindet. Um die aktuelle Situation der Suche festzustellen, wird in regelmäßigen Abständen die Erfolgsrate

$$p_s = \frac{\text{Anzahl an erfolgreichen Mutationen}}{\text{Anzahl aller Mutationen}} \tag{2.3}$$

ermittelt und mit einem Schwellwert Θ verglichen:

$$p_s > \Theta : \sigma' \leftarrow \sigma \cdot \alpha \tag{2.4}$$

$$p_s < \Theta : \sigma' \leftarrow \frac{\sigma}{\alpha} \tag{2.5}$$

$$p_s = \Theta : \sigma' \leftarrow \sigma \tag{2.6}$$

Auch wenn diese Variante eine Verbesserung gegenüber der festen Schrittweitenanpassung darstellt, so kann es, je nach Optimierungsproblem, zur vorzeitigen Konvergenz kommen.

Eine weitere, bessere Variante ist die *selbstadaptive Schrittweitenanpassung*. Hier werden alle Individuen um den Parameter σ erweitert, welcher in A.S, also als Zusatzinformation der Individuen, gespeichert wird. Dieser Parameter wird dann in jeder Generation ebenfalls einer (gaußschen) Mutation unterzogen. Die Idee dahinter ist, dass gute Individuen auch gute Strategieparameter besitzen und in späteren Generationen fortbestehen. Hat ein Individuum eine schlechte Schrittweite, wird dieses (bzw. dessen Kinder) früher oder später nicht mehr in die nächste Generation übernommen werden. In vielen Fällen liefert diese selbstadaptive, gaußsche Mutation gute Ergebnisse. Weitere Varianten werden in [Weicker, 2007] vorgestellt, finden in dieser Arbeit jedoch keine Verwendung, da ihre Implementierung zu aufwendig im Vergleich zu ihrem Nutzen wäre.

Abbildung 2.2 zeigt ein Beispiel für eine Mutation von einem Lösungskandidaten A bzw. von dessen *Gene*. Jede Zahl stellt ein *Gen* dar und besitzt eine Mutationswahrscheinlichkeit von 30%. Die Gene an den Stellen sechs und neun werden einer Mutation unterzogen, die anderen bleiben unverändert.

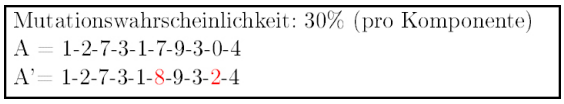


Abbildung 2.2.: Beispiel einer Mutation

Rekombination: Die Rekombination ist dafür zuständig λ viele Kind-Individuen zu erzeugen und sorgt, im Gegensatz zur Mutation, in der Regel für recht große Veränderungen. Allerdings ist die Größe der Veränderung stark von der Diversität, also der Vielfalt an verschiedenen Individuen innerhalb der Population, abhängig. Je unterschiedlicher die Individuen sind, desto größer können die Änderungen ausfallen.

Da eine Rekombination nur zu einer gewissen Wahrscheinlichkeit eintritt und es EA-Varianten gibt, welche keine Rekombination benutzen, zum Beispiel die *(1+1)- ES*, müssen die fehlenden Kind-Individuen durch Duplizierung von vorhandenen Elter- Individuen ergänzt werden. Die Rekombination benötigt mindestens zwei verschiedene *Elter-Individuen* $E_1, \ldots, E_k$ um aus ihnen ein oder mehrere neue *Kind-Individuen* $C_1, \ldots, C_l$ zu generieren, welche dann aus Teilen der Eigenschaften der Eltern zusammengesetzt werden. Im Folgenden wird davon ausgegangen, dass an einer Rekombination zwei Eltern E_1 und E_2 beteiligt sind und diese zwei Kinder C_1 und C_2 erzeugen. Wie die Selektion der Eltern durchgeführt wird, wird im Abschnitt *Elternselektion* (s.u.) erläutert.

Falls eine Rekombination eintritt, muss die Position der Schnittstelle ermittelt werden, an der die *Gene* der Eltern E_1 und E_2 auseinander geschnitten werden sollen. In der Regel liegt die Zahl der Schnittstellen bei *Anzahl der beteiligten Eltern - 1*. Durch das Auseinanderschneiden entstehen aus den Eltern E_1 und E_2 insgesamt vier Teilstücke $E_1.1$, $E_1.2$ und $E_2.1$, $E_2.2$. Anschließend werden die Teilstücke von E_1 und E_2 untereinander kombiniert, so dass zwei neue Kind-Individuen C_1 und C_2 entstehen, welche zum Teil aus den Genen von E_1 und zum Teil aus den Genen von E_2 bestehen. Abbildung 2.3 zeigt ein Beispiel für eine Rekombination zweier Individuen E_1 und E_2.

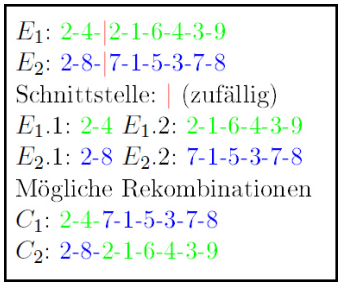


Abbildung 2.3.: Beispiel einer Rekombination

Dabei ist darauf zu achten, dass nur passende Teilstücke miteinander kombiniert werden. So könnte in dem Beispiel (s. o) $E_1.1$ nicht mit $E_2.1$ kombiniert werden, da das daraus resultierende Individuum keine gültige Lösung darstellen würde. Eine gültige Lösung muss in diesem Falle genau acht Gene besitzen.

Allerdings ist diese Art der Rekombination nicht für alle Optimierungsprobleme geeignet. So benötigen Probleme, bei denen die Lösungskandidaten aus Permutationen über

eine Menge von Elementen bestehen, andere Mechanismen der Rekombination, die jedoch hier nicht weiter behandelt werden, da dies über den Rahmen dieser Arbeit hinausgehen würde.

Auch wenn durch die Rekombination große Änderungen vorkommen können, so ist dabei zu beachten, dass sie keine neuen Lösungen hervorbringen kann. Sie kombiniert lediglich die schon vorhandenen (Teil-)Lösungen miteinander. Um wirklich neue Lösungen zu erzeugen, benötigt man die Mutation.

Definition 4 *Für einen durch den Genotyp G kodiertes Optimierungsproblem und die Zusatzinformationen Z, wird ein Mutationsoperator durch die Abbildung*

$$Mut^{\xi} : G \times Z \to G \times Z, \tag{2.7}$$

definiert, wobei $\xi \in \Xi$ einen Zustand des Zufallszahlengenerators darstellt.

Definition 5 *Analog wird ein Rekombinationsoperator mit $r \geq 2$ Eltern und $s \geq 1$ Kindern ($r, s \in \mathbb{R}$) durch die Abbildung*

$$Rek^{\xi} : (G \times Z)^r \to (G \times Z)^s \tag{2.8}$$

definiert.

Selektion Bei der *Elternselektion* sollte jedes Individuum einer Elterngeneration eine Chance haben für den nächsten Schritt innerhalb des Evolutionszyklus, der Rekombination, ausgewählt zu werden. Dies kann man auf mehrere Weisen erreichen. Die einfachste ist es die Selektion nach dem Zufallsprinzip vorzunehmen, d.h. jedes Individuum hat, unabhängig von seinen Eigenschaften, die gleiche Chance von $1/\mu$ als Elter ausgewählt zu werden. Eine andere Variante ist es die Wahrscheinlichkeit an die Fitness des Individuums zu koppeln. Je höher der Fitnesswert eines Individuums $A^{(i)} \in P$, desto höher die Wahrscheinlichkeit $PR(A^{(i)})$, dass es als Elter ausgewählt wird:

$$PR(A^{(i)}) = \frac{A^{(i)}.F}{\sum_{k=1}^{\mu} A^{(k)}.F}, \text{ mit } \leq i \leq \mu \tag{2.9}$$

Bei der ersten Variante, der gleichverteilten Selektion, wird die Diversität der Population stärker gefördert, bei der zweiten verläuft die Suche innerhalb des Suchraumes weniger zufällig, denn bei einer Selektion, die von der Fitness abhängt, werden die Gene der besseren Individuen häufiger wiederverwendet als bei der gleichverteilten Selektion.

Über längere Zeit kann dies die Diversität der Population verringern, da durch Mutation und Rekombination häufig recht ähnliche Individuen entstehen, welche ähnliche Fitnesswerte aufweisen. So werden vermeintlich schlechte Individuen nach und nach verdrängt und die Suche konzentriert sich auf einen kleinen Ausschnitt des Suchraumes. Bei Optimierungsproblemen mit vielen lokalen Optima kann dies schnell zu einem Problem werden, da es für die folgenden Generationen schwer werden wird, den eingegrenzten Suchraum durch die Rekombination wieder zu verlassen. In einem solchen Fall, kann nur noch die Mutation einen Ausweg bieten, allerdings nur wenn die Schrittweite σ ausreichend groß ist.

Die *Umweltselektion* wird, wie der Name schon sagt, in der Natur durch die Umwelt durchgeführt. Sie findet heraus, welche Individuen einer Population die am besten angepassten und damit die überlebensfähigsten sind. Je *fitter*[2] ein Individuum ist, desto höher ist die Wahrscheinlichkeit, dass sich die Gene von diesem auch in Zukunft durchsetzen. In der Natur gibt es dabei noch andere Formen der Selektion, die hier aber nicht weiter betrachtet werden sollen.

Bei den EAs übernimmt die *Fitnessfunktion f* die Aufgabe der Umweltselektion. Zu jedem Lösungskandidaten wird ein Fitnesswert berechnet, welcher bestimmt, wie gut das entsprechende Individuum ist. Die einfachste Form der Selektion wählt die μ-besten Individuen der λ Nachkommen aus und übernimmt diese in die nächste Generation. Je nach Algorithmus werden nur aus den λ Nachkommen oder aus den λ Nachkommen und μ Eltern die Individuen für die nächste Generation ausgewählt. Dieser wird am Anfang des Optimierungsprozesses festgelegt und in der Regel nicht mehr verändert.

Definition 6 *Ein Selektionsoperator wird auf eine Population*

$$P = \left\langle A^{(i)}, \ldots, A^{(r)} \right\rangle \tag{2.10}$$

angewandt:

$$Sel^{\xi} : (G \times Z \times \mathbb{R})^r \to (G \times Z \times \mathbb{R})^s \tag{2.11}$$

$$\left\langle A^{(i)} \right\rangle_{1 \le i \le r} \to \left\langle A^{(IS^{\xi}(c_1,\ldots,c_r)_k)} \right\rangle_{1=k=s} \tag{2.12}$$

Die dabei zugrunde gelegte Indexselektion IS hat die Form

$$IS^{\xi} : \mathbb{R}^r \to \{1, ..., r\}^s . \tag{2.13}$$

Schema von EAs

Definition 7 *Ein generischer evolutionärer Algorithmus zu einem Optimierungsproblem* $(\Omega, f, ' >')$ *ist ein 8-Tupel (G, dec, Mut, Rek,* IS_{Eltern}*,* IS_{Umwelt}*,* μ*,* λ*). Dabei bezeichnet* μ *die Anzahl der Individuen in der Elternpopulation und* λ *die Anzahl der erzeugten Kinder pro Generation. Ferner gilt:*

$$Rek : (G \times Z)^k \to (G \times Z)^{k'} \tag{2.14}$$

$$IS_{Eltern} : \mathbb{R}^{\mu} \to (1, ..., \mu)^{\frac{k}{k'} \cdot \lambda}, \text{ mit } \frac{k}{k'} \cdot \lambda \in \mathbb{N} \tag{2.15}$$

$$IS_{Umwelt} : \mathbb{R}^{\mu+\lambda} \to (1, ..., \mu + \lambda)^{\mu} \tag{2.16}$$

[2] fitter im Sinne der Anpassung an die Umwelt

2.2.2. Schwarmintelligenz

Im Rahmen dieser Arbeit soll die *Schwarmintelligenz (SI)* als Basis genutzt werden, um die Simulation von glaubwürdigen Charakteren/Lebenwesen zu bewerkstelligen. Die SI ist ein Forschungsgebiet der künstlichen Intelligenz, bei der staatenbildende Insekten als Modellvorlage dienen um Algorithmen zu entwerfen, welche zur Lösung komplexer Probleme eingesetzt werden können. Der Ausdruck SI wurde 1989 von Gerardo Beni und Jing Wang im Rahmen der Robotikforschung geprägt [Gerardo Beni, 1989]. Dabei handelt es sich um einen populationsbasierten Ansatz bei dem viele relativ einfache und homogene Agenten zum Einsatz kommen. Anders als bei der klassischen KI liegt der Schwerpunkt jedoch nicht auf der Intelligenz des Einzelnen, sondern vielmehr auf dem kooperativen Verhalten der Agenten. Ein einzelner Agent besitzt nur begrenzte, lokale Informationen über seine Umwelt und über das Problem selbst. Dadurch trägt jeder Agent zwar zur Lösung eines Problems bei, jedoch ohne einen Überblick darüber zu haben, wie die Lösung zustande gekommen ist. Die Regeln, nach denen die Agenten funktionieren, sind meist sehr simpel. Trotz des Fehlens einer zentralen Kontrolle kann ein sehr komplexes Verhalten erzielt werden, welches weit mehr ist als die Summe der einzelnen Aktionen. Ein auf diese Weise entstehendes Verhalten wird auch als *emergent* bezeichnet.

SI-Algorithmen können durchaus als Greedy- und Approximations-Algorithmen bezeichnet werden, denn die nächste Aktion hängt nur von dem gegenwärtigen Zustand ab, ohne dabei zu beachten, welche Folgen diese Entscheidung für die Zukunft haben könnte. Außerdem gibt es keine Garantie, dass die optimale Lösung gefunden wird.

Zwei sehr bekannte Beispiele für Schwarmintelligenz sind die Ant Colony Optimization [Dorigo, 1992] und die Particle Swarm Optimization [J. Kennedy, 1995], welche im Folgenden genauer erklärt werden.

Ant Colony Optimization (ACO)

ACO-Algorithmen konzentrieren sich auf die Lösung des *Kürzeste-Wege-Problems* und aller anderen Probleme, die sich darauf reduzieren lassen. Dafür hat man das Verhalten von Ameisen bei der Nahrungssuche beobachtet. Obwohl die Sichtweite von Ameisen äußerst beschränkt ist, finden sie üblicherweise in kurzer Zeit den nahezu optimalen Weg zwischen ihrem Nest und einem Ziel, wie beispielsweise einer Futterquelle. Die folgenden Informationen wurden [Dorigo et al., 2006] entnommen.

Natürliches Vorbild: Der Erfolg der Ameisen begründet sich auf *Stigmergie* [3], also den Veränderungen der Umwelt, die sie während ihrer Suche an ihrem Weg vornehmen. Über diese Veränderungen können sie indirekt miteinander kommunizieren. Im Falle der Ameisen besteht die Veränderung aus dem Platzieren von Pheromonen. Trifft eine Ameise auf eine solche Pheromonspur, folgt sie ihr mit einer gewissen Wahrscheinlichkeit in Abhängigkeit von der Pheromon-Konzentration. Die Konzentration hängt dabei von zwei Faktoren ab: Erstens von der Häufigkeit mit der die Pheromone aufgetragen wurden und zweitens von dem Alter der Pheromone.

[3] Der Begriff Stigmergie wurde 1959 vom französischen Biologen Pierre-Paul Grassé eingeführt [Grassé, 1959].

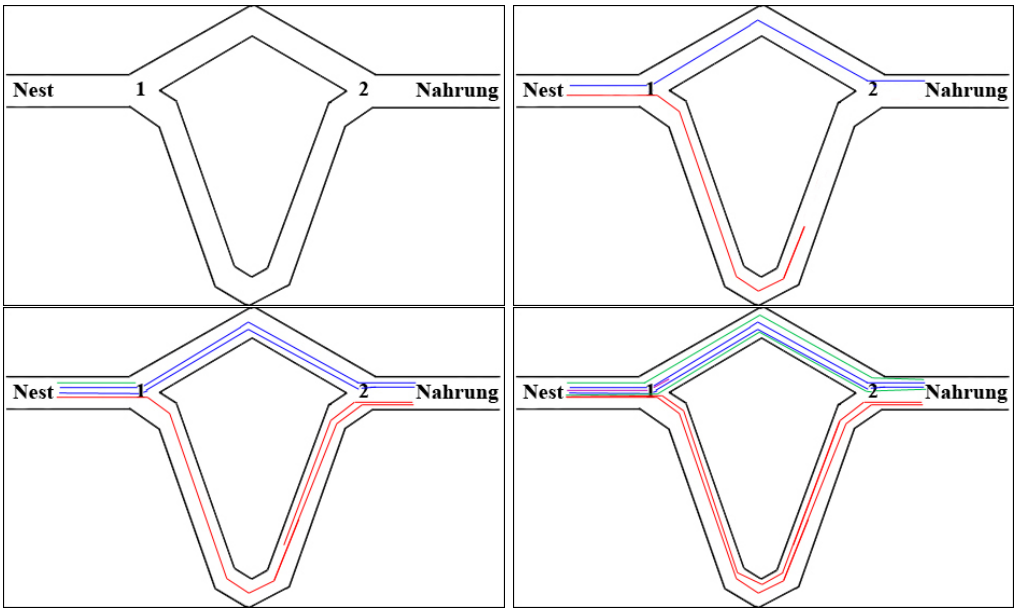


Abbildung 2.4.: Im Verlauf der Zeit steigt die Konzentration auf der kürzeren Strecke schneller an als auf der längeren, da in der gleichen Zeit die Wege häufiger abgelaufen werden. In der Zeit, die eine Ameise braucht, um den langen Weg zweimal abzulaufen, kann eine Ameisen den kürzeren Weg fünfmal ablaufen.

Je älter die Spur ist, desto schwächer wird sie, da die Pheromone mit der Zeit verdunsten. Das Besondere an den verdunstenden Pheromonspuren ist: Die kürzeste Strecke von A nach B hat implizit auch die höchste, maximale Pheromon-Konzentration und damit die höchste, maximale Wahrscheinlichkeit, von Ameisen verfolgt zu werden. Dies liegt darin begründet, dass die Pheromone einer langen Strecke weniger häufig erneuert werden können als die von einer kürzeren. Ist ein Weg doppelt so lang wie eine kürzere Alternative, ist die Menge an hinterlassenen Pheromonen über einen betrachteten Zeitraum zwar die gleiche, sie verteilt sich jedoch auf eine längere Strecke und weist daher eine geringere Konzentration auf. Je kürzer eine Strecke ist, desto höher kann die Pheromon-Konzentration auf ihr werden und desto schneller kann die Konzentration ansteigen (Abbildung 2.4).

Algorithmus: Überträgt man dieses Verfahren auf einen formalen Algorithmus, benötigt man zunächst einen ungerichteten *Graphen G=(V,E)*, der die Umgebung abbildet (Abbildung 2.5). Die Menge von Knoten V stellt die relevanten Entscheidungspunkte dar, an denen sich die virtuellen Ameisen entscheiden müssen, welchem Weg sie fortan folgen möchten. Die Menge von Kanten E verbindet jeweils zwei Knoten miteinander und beinhaltet die jeweiligen Kosten (zum Beispiel benötigte Zeit oder Länge) $\eta_{i,j}$ und Pheromon-Konzentration $\tau_{i,j}$ zwischen den beiden Knoten i und j.

- Initialisierung (Graphen bilden, Parameter setzen, Pheromonspuren initialisieren)
- Solange (Abbruchkriterium nicht erfüllt)

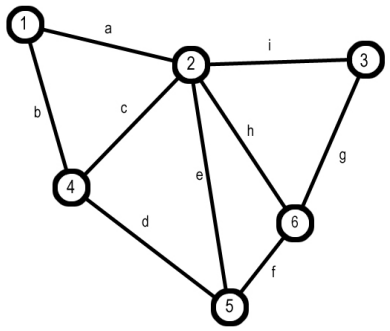


Abbildung 2.5.: Beispiel für einen Graphen

1. Konstruiere n viele Lösungen (n = Anzahl der Ameisen)
2. Optionale Aktionen
3. Pheromonaktualisierung

Nach der Initialisierung läuft die Simulation in einer Schleife ab, bei der in jedem Durchgang jeder Ameisen-Agent eine gültige Lösung (einen kompletten Weg von A nach B) konstruiert. Dabei entscheidet der Ameisenagent an jedem Knoten probabilistisch, welchen Weg er als nächstes wählt. Je höher die Pheromon-Konzentration, desto höher die Wahrscheinlichkeit, dass dieser Weg gewählt wird. Nachdem alle Ameisen eine Lösung berechnet haben, findet die Pheromonaktualisierung statt. Je nach verwendeter ACO-Variante werden dann die Pheromonspuren aktualisiert. Im einfachsten Fall, wird zuerst die Verdunstung durchgeführt und danach eine neue Pheromonspur auf der besten, in diesem Durchlauf gefundenen, Lösung aufgetragen. Im Folgenden werden die einzelnen Schritte detaillierter betrachtet.

Initialisierung: Bei der Initialisierung werden die Startwerte der Pheromon-Konzentration gesetzt oder andere, problemspezifische Parametereinstellungen vorgenommen. Je nach verwendeter Variante können die Startwerte bei Null oder einem bestimmten Wert beginnen.

Konstruktion einer Lösung: Jede Lösung startet mit einer leeren Menge von Knoten s^P. Je nachdem, was für ein Problem gelöst werden soll (zum Beispiel Travelling Salesman Problem (TSP) oder kürzeste Wege), wird nun schrittweise eine gültige Lösung erstellt, indem bei jedem Schritt ein Knoten der Lösung hinzugefügt wird. Die Wahl des Knotens hängt dabei von der bisherigen Teillösung s^P, den Nachbarn des zuletzt hinzugefügten Knotens, der Pheromon-Konzentration und den heuristischen Variablen der Kanten zwischen dem letzten Knoten und seinen Nachbarn ab. Der Startknoten wird, abhängig vom Problem, entweder vorgegeben (Shortest Path) oder kann zufällig gewählt werden (TSP). Die Wahrscheinlichkeit p, dass die Kante $c_{i,j}$ gewählt wird, berechnet sich bei der bekanntesten Form der ACO-Algorithmen wie folgt:

$$p(c_{i,j}|s^P) = \frac{\tau_{i,j}^{\alpha} \cdot \eta_{i,j}^{\beta}}{\sum_{c_{i,l} \in N(s^P)} \tau_{i,l}^{\alpha} \cdot \eta_{i,l}^{\beta}}, \forall c_{i,j} \in N(s^P) \tag{2.17}$$

Die Parameter:

- s^P: Bisherige Teillösung, der zuletzt aufgenommene Knoten stellt die aktuelle Position dar
- $N(s^P)$: Menge der Nachbarknoten der aktuellen Position
- $\tau_{i,j}$: Pheromon-Konzentration zwischen den Knoten i und j
- $\eta_{i,j}$: Heuristische Variable auf Basis der Kosten der Kante zwischen i und j (zum Beispiel $1/d_{i,j}$ (Länge))
- α, β: Gewichtungsfaktoren $\in$]0,∞], die die relative Gewichtung der Pheromon-Konzentration und der heuristischen Variable angeben

Optionale Aktionen: Abhängig von dem zu lösenden Problem kann es vorkommen, dass zwischen dem Konstruieren der Lösungen und der Pheromonaktualisierung spezielle, nachbearbeitende Schritte erforderlich sind. Am häufigsten wird an dieser Stelle eine lokale Suche auf die gefundenen Lösungen angewendet.

Pheromonaktualisierung: Die Pheromonaktualisierung ist ein sehr wichtiger Schritt bei den ACO-Algorithmen. Ihr Ziel ist es dafür zu sorgen, dass gute Lösungen einen hohen und schlechte Lösungen einen niedrigen Wert erhalten. Im Allgemeinen wird dies ähnlich wie in der Natur bewerkstelligt. Zum einen verdunsten die Pheromone mit der Zeit und nehmen an Intensität ab, zum anderen trägt jede Ameise neue Pheromone auf alle von ihr besuchten Kanten auf. Jedoch unterscheiden sich auch die Pheromonaktualisierungen von ACO zu ACO. Die ursprüngliche Formel für die Aktualisierung lautet:

$$\tau_{i,j} \leftarrow (1-\rho) \cdot \tau_{i,j} + \sum_{k=1}^{m} \Delta\tau_{i,j}^{k} \tag{2.18}$$

Die Parameter:

- $\tau_{i,j}$: Pheromon-Konzentration auf der Kante zwischen Knoten i und j
- ρ: $\in$]0,1] ist die Verdunstungsrate
- m: Anzahl der Ameisen
- $\Delta\tau_{i,j}^{k}$: Die Menge an Pheromonen, die aufgetragen wurde von der *k-ten* Ameise
- $\Delta\tau_{i,j}^{k} = 0$ falls Kante nicht besucht wurde von Ameise k, sonst $1/L_k$, mit L_k die Länge der gesamten Tour der Ameise

Particle Swarm Optimization (PSO)

Ein weiteres, bekanntes Beispiel für die SI ist die PSO. Als Vorlage dient das Verhalten von Vögeln und Fischen, welche sich als große Schwärme bewegen können.

Natürliches Vorbild: Schwärme erwecken den Anschein, als handle es sich es um ein einziges, großes Objekt und nicht um eine einfache Ansammlung aus vielen einzelnen Individuen. Diese Illusion kann durch wenige und einfache Regeln, an die sich alle beteiligten Individuen innerhalb des Schwarms halten müssen, aufgebaut werden. Die Regeln sollen hier anhand eines Fischschwarms vorgestellt werden. Für andere Schwärme gestalten sich diese jedoch meistens recht ähnlich. Die Abbildung 2.6 zeigt die drei grundlegenden Regeln von Fischschwärmen:

1. *Ausrichtung:* Jeder Fisch passt seine Richtung und Geschwindigkeit an die Richtung und Geschwindigkeit seiner direkten Nachbarn an.

2. *Schwarmzentrierung:* Jeder Fisch versucht, sich dem Zentrum des Schwarms zu nähern.

3. *Kollisionsvermeidung:* Jeder Fisch hält zu seinen Nachbarn einen Mindestabstand ein.

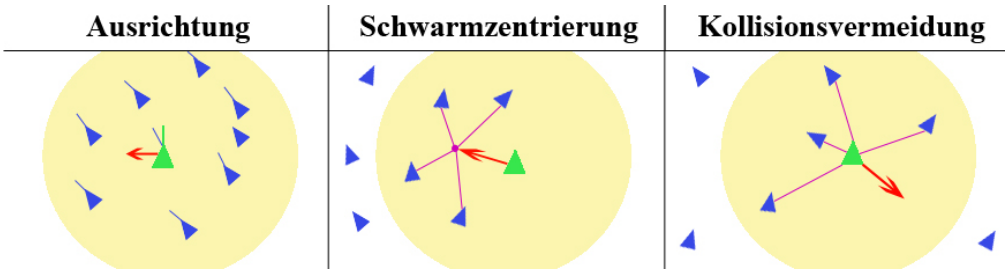


Abbildung 2.6.: Die drei grundlegenden Regeln von Schwärmen

Damit die Fische auch in der Lage sind Hindernissen auszuweichen und auf Futterquellen zu reagieren, sind noch zusätzliche drei Regeln erforderlich.

1. Der vorderste Fisch steuert die nächste Nahrungsquelle an.

2. Der vorderste Fisch bewegt sich mit großem Abstand an Hindernissen vorbei.

3. Jeder Fisch verlangsamt seine Geschwindigkeit und nährt sich weiter dem Zentrum des Schwarms, falls vor ihm Hindernisse auftreten.

Im Folgenden soll erläutert werden, wie dieses Verhalten von Vögel- und Fischschwärmen eingesetzt wird um Optimierungsprobleme zu lösen.

Partikelschwarmoptimierung: Die Informationen über die PSO sind [J. KENNEDY, 2001] entnommen worden, es sei denn, es wird explizit eine andere Quelle angegeben. Die Partikelschwarmoptimierung wird eingesetzt, um nahezu optimale Lösungen für eine Funktion f zu finden. Bei f kann es sich um eine beliebige Funktion handeln, welche einen d-dimensionalen Eingabevektor j aus dem Wertebereich $\mathbb{R}^d$ als Eingabe akzeptiert:

$$f : \mathbb{R}^d \to \mathbb{R}^x \tag{2.19}$$

Zwar ist der Such- und Zielraum im Allgemeinen $\mathbb{R}^d$ bzw. $\mathbb{R}^1$, es können jedoch auch sehr viel allgemeinere Such- und Zielräume benutzt werden, ohne dass die generellen Prinzipien der Partikelschwarmoptimierung geändert werden müssten [CLERC, 2006]. Häufig werden zusätzlich die Wertebereiche der Eingabevektoren j auf $[X_{min}, X_{max}]^d$ beschränkt, so dass die Suche nur innerhalb eines bestimmten Unterraumes stattfindet. Dieser Unterraum stellt den *sinnvollen* Suchbereich dar und ist von dem zu lösenden Problem abhängig.

Damit die Qualität einer Lösung bewertet werden kann, benötigt man eine *Fitnessfunktion fit*:

$$fit : \mathbb{R}^d \to \mathbb{R}^1, \tag{2.20}$$

die jeden Eingabevektor j, bzw. jede Position x des Suchraumes, auf einen eindeutigen Wert aus $\mathbb{R}^1$ abbildet. Je höher der Fitnesswert, desto besser die gefundene Lösung. Bei Maximierungsprobleme kann beispielsweise als Fitnessfunktion $fit(x) = f(x)$, bei Minimierungsproblemen $fit(x) = 1/f(x)$, genutzt werden. Ist die Ausgabe der Funktion f kein einzelner Wert, muss eine eigene Fitnessfunktion erstellt werden.

Jedes *Partikel* i des Schwarms der Größe n wird zum Zeitpunkt t durch mehrere Informationen beschrieben:

- x_i(t) = (x_i1, x_i2, ..., $x_i d$): die aktuelle Position innerhalb des d-dimensionalen Suchraumes
- p_i(t) = (p_i1, p_i2, ..., $p_i d$): die beste, selbst besuchte Position
- v_i(t) = (v_i1, v_i2, ..., $v_i d$): den aktuellen d-dimensionalen Geschwindigkeitsvektor

Für gewöhnlich werden zu Beginn der PSO Startposition $x(t = 0)$ und Geschwindigkeitsvektor $v(t = 0)$ aller Partikel zufällig initialisiert. Ist a priori-Wissen über das zu bearbeitende Problem vorhanden, können Startpositionen und Geschwindigkeitsvektoren entsprechend gewählt werden, um den Optimierungsprozess zu beschleunigen.

Die Optimierung selber ist ein *iterativer Prozess*. Die allgemeine Vorgehensweise sieht für jedes Partikel i drei Schritte vor, die solange in einer Schleife ausgeführt werden, bis ein vom Benutzer festgelegtes Abbruchkriterium (zum Beispiel das Erreichen eines bestimmten Fitnesswertes oder nach einer festgelegten Anzahl von Iterationen) erfüllt worden ist. Die drei Schritte sind:

1. *Evaluieren*: Berechnung der Fitness der aktuellen Position x. Speichere x als beste, selbst besuchte Position p, falls $fit(x) > fit(p)$.
2. *Vergleichen*: Bestimmung der Position g des Nachbarpartikels mit dem höchsten Fitnesswert

3. *Imitieren*: Anpassung des Geschwindigkeitsvektors v und anschließende Veränderung der eigenen Position x.

Wie bei dem natürlichen Vorbild „fliegen“ die Partikel während der Optimierung innerhalb des Suchraumes umher, wobei sie bei jedem Schritt neue Lösungen finden. Die Abbildung 2.8 zeigt diese Bewegung beispielhaft. Die neue Position x_i des Partikels i berechnet sich aus seiner vorherigen Position $x_i(t-1)$ und dem neuem Geschwindigkeitsvektor $v_i(t)$:

$$x_i(t) = x_i(t-1) + v_i(t) \tag{2.21}$$

Der neue *Geschwindigkeitsvektor* $v_i(t)$ des Partikels i berechnet sich aus drei Bestandteilen (Abbildung 2.7):

- $v_i(t-1)$: dem vorherigen Geschwindigkeitsvektor
- $p_i - x_i(t)$: einem Vektor in Richtung der bisher besten, selbst gefundenen Lösung (individuelle Suche)
- $g_i - x_i(t)$: einem Vektor in Richtung des Nachbarpartikels mit der höchsten Fitness (kollektive Suche)

Diese drei Bestandteile werden noch durch die Gewichtungen c_1, c_2 und c_3 modifiziert. Dabei ist c_1 für die Gewichtung der Eigenbewegung, c_2 für die Gewichtung der individuellen Suche und c_3 für die Gewichtung der kollektiven Suche verantwortlich. Eine detaillierte Beschreibung dieser Gewichtungen findet weiter unten statt:

$$v_i(t) = (c_1 \cdot v_i(t-1))) + (c_2 \cdot (p_i - x_i(t))) + (c_3 \cdot (g_i - x_i(t))) \tag{2.22}$$

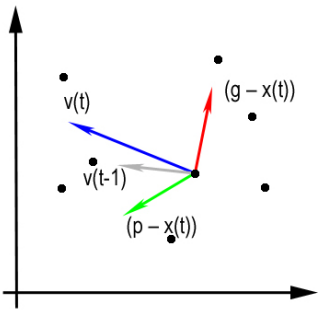


Abbildung 2.7.: Die drei Vektorkomponenten: blau = tatsächliche Bewegung, rot = in Richtung des besten Nachbarn, grün = in Richtung der besten eigenen Position, grau = vorherige Bewegung

Parameter: Die richtige Wahl dieser Parameter ist entscheidend für die Qualität der von der PSO gefundenen Lösungen. Wissen über das Problem *kann* dabei sehr hilfreich sein und die benötigte Zeit für die Optimierung stark verkürzen. Im Gegenzug können schlecht gewählte Parameter zu schlechten Ergebnissen oder zu einer langsamen Konvergenz führen. Grund genug um die Parameter im Detail zu betrachten.

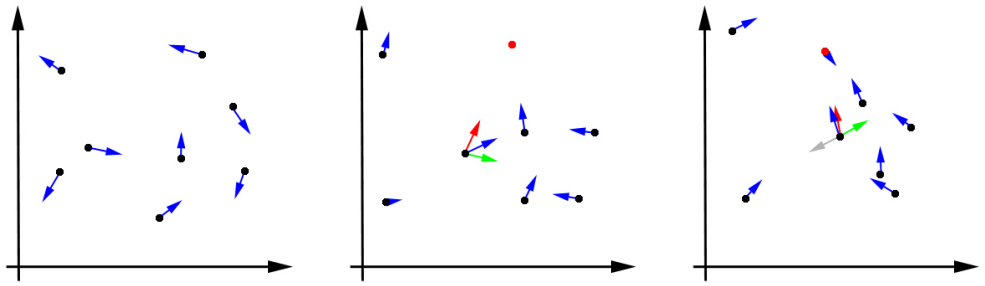


Abbildung 2.8.: Die Bewegungen der Partikel nach der Initialisierung mit zufälligen Geschwindigkeitsvektoren und der 1. Iteration bzw. der 2. Iteration

Selbstvertrauen (in die eigene Bewegung) $c_1 \in]0,1[$: Stellt die Präferenz der Partikel ihrer alten Bewegung zu folgen dar. Hierüber lässt sich die Konvergenzgeschwindigkeit der PSO steuern. Je kleiner c_1, desto schneller konvergiert der Algorithmus. Allerdings erhöht sich mit kleiner werdender c_1 auch das Risiko einer vorzeitigen Konvergenz, da lokale Optima wegen der immer kleiner werdenden Geschwindigkeiten nicht mehr verlassen werden können. Ist c_1 hingegen zu groß, verlangsamt dies die Konvergenz. Der Algorithmus benötigt dann mehr Zeit, behält jedoch länger die Fähigkeit, lokale Optima zu verlassen.

Es ist wichtig, dass $c_1 < 1$ gewählt wird, da für Werte $c_1 > 1$ sich keine Konvergenz einstellen und es zu einer sog. *Explosion* kommen könnte. Bei der Explosion handelt es sich um das Phänomen, dass die Geschwindigkeiten der Partikel stetig anwachsen. Dadurch wird zum einen eine Konvergenz der Position verhindert und zum anderen der interessierende Suchraum verlassen. Allerdings fördern Werte größer 1 die Erkundung des Suchraumes. Falls man sich dazu entscheidet $c_1 > 1$ zu wählen, wird ein weiterer Parameter v_{max} benötigt, um dem oben angesprochenen Problem der Explosion entgegen zu wirken. v_{max} begrenzt die maximale Geschwindigkeit $v \in [-v_{max}, +v_{max}]$ der Partikel, so dass sie nicht unkontrolliert ansteigen kann. v_{max} kann für jede Dimension unterschiedlich ausfallen und sollte einen Wert, der der halben Reichweite des *sinnvollen*[4] Suchraumes der jeweiligen Dimension entspricht, nicht überschreiten. Ein größerer Wert würde dazu führen, dass der sinnvolle Suchraum zu häufig verlassen wird.

Geschwindigkeitskoeffizienten c_2, c_3: Die Geschwindigkeitskoeffizienten c_2 und c_3 bestehen aus d vielen, gleichverteilten Zufallszahlen aus dem Intervall $[0, c_{max}]$ $c_{max} \in \mathbb{R}$ und steuern die Balance zwischen der individuellen und der kollektiven Suche. Eine individuellere Suche beschleunigt die Konvergenz, verbleibt aber häufiger in lokalen Optima. Eine kollektivere Suche konvergiert langsamer, ist dafür robuster. Es sind d verschiedene Zufallszahlen, da jede Komponente der d-dimensionalen Vektoren $(p - x)$ und $(g - x)$ mit einer eigenen Zufallszahl modifiziert werden sollte. Ein häufig gemachter Fehler in

[4] Mit *sinnvoll* ist im Sinne des Problems sinnvoll gemeint. Wenn zum Beispiel der Startzeitpunkt einer Aktion optimiert werden soll, sollte der Suchraum auf den Wertebereich von 0 bis 24 Stunden eingegrenzt werden.

der Implementierung ist die Verwendung von nur einer einzigen Zufallszahl für alle d Dimensionen, was zu schlechteren Ergebnissen führen kann. Die zufällige Gewichtung spielt eine wichtige Rolle, denn für viele Probleme ist nicht bekannt, ob eine kollektive oder eine individuelle Suche schneller zu besseren Ergebnissen führt. So fällt der Einfluss beider Komponenten mal stärker, mal schwächer ins Gewicht.

Schwarmgröße n: Die Schwarmgröße n beeinflusst die Zeit, die die PSO benötigt, um eine gute Lösung zu finden und wird zu Beginn der Optimierung festgelegt. Wird sie zu groß gewählt, werden zwar in weniger Iterationen bessere Ergebnisse gefunden, jedoch ist die Anzahl der Funktionsevaluationen, also das Produkt aus Iterationen und Schwarmgröße, das entscheidende Kriterium für die Geschwindigkeit der PSO und nicht die Anzahl der Iterationen. Wird die Population hingegen zu klein gewählt, kann dies die benötigte Zeit ebenfalls erhöhen oder sogar dazu führen, dass nur sehr schlechte Lösungen gefunden werden.

Topologie der Nachbarschaft: Die Topologie der Nachbarschaft ist ein weiterer Parameter, der frei gewählt werden kann. Die Nachbarschaft wird dabei durch zwei Faktoren bestimmt:

- Die Form (Ring, Stern, Cluster)
- Die Größe der Nachbarschaft k

Nach [Clerc, 2006] gilt für die Größe k im Allgemeinen: Je weniger lokale Optima die Funktion f besitzt, desto größer sollte die Nachbarschaft gewählt werden, denn weist eine Funktion viele lokale Optima auf, ist es sinnvoll, wenn nur wenige Nachbarpartikel in diese Richtung gelenkt werden. Die restlichen Partikel können so besser andere Teile des Suchraumes erkunden. Wie bei den anderen Parametern auch, gibt es keine „beste“ Nachbarschaftsform und Größe. Häufig verwendete Nachbarschaften sind jedoch:

- Die k-nächste Nachbarn im Suchraum, k zwischen 3 und 5
- Jeder Partikel ist Nachbar von jedem anderen Partikel (globale Nachbarschaft $k = n$)
- Ringstruktur als Nachbarschaft (a-b-c-d-e-a)

Die drei verschiedenen Nachbarschaften werden schematisch in Abbilding 2.9 dargestellt.

Konvergenz Die Analyse des Konvergenzverhaltens einer PSO ist ein schwieriges Unterfangen. Daher beschränken sich die meisten Bücher und Artikel über die PSO auf die Analyse des Konvergenzverhaltens von nur zwei Partikeln. Es gibt jedoch Methoden, wie eine PSO zu einer Konvergenz gezwungen werden kann. Eine häufig verwendete ist es, die Gewichtung der eigenen Bewegung c_1 mit der Zeit zu verringern, so dass die Bewegung mit fortschreitender Zeit zum Stillstand kommt. An dieser Stelle soll jedoch nicht weiter auf die Analyse des Konvergenzverhaltens eingegangen werden, da eine Untersuchung der Konvergenz für diese Arbeit nicht relevant ist. In Kapitel 4.2.1 wird erläutert, dass

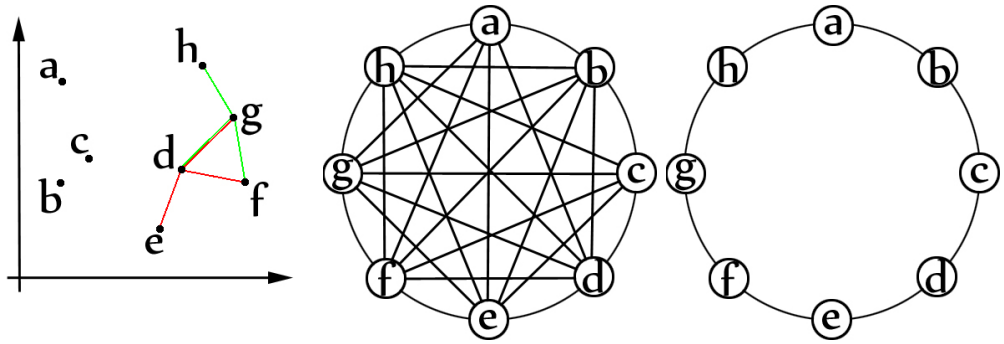


Abbildung 2.9.: KNN mit (k=3), Globale Nachbarschaft (k=n) und Ringstruktur (k=2)

Parameter	**Zweck**	**Wertebereich**	**Empirische Empfehlung**	**Empfohlener Wert**
c_1	Selbstvertrauen	$\mathbb{R}$	$\in]0,1[$	0.7
c_{max}	Vertrauen in andere	$\mathbb{R}$	ca 1.5	1.43
n	Schwarmgröße	$\mathbb{N}$	20 - 40	20
k	Nachbarschaftsgröße	$\mathbb{N}$	3 - 5 oder n	3
v_{max}	Geschwindigkeitsbegrenzung	$\mathbb{R}$	$(X_{max}\text{-}X_{min})/2$ (nur nötig falls $c_{max} > 1$)	$(X_{max}\text{-}X_{min})/2$

Tabelle 2.1.: Empirisch ermittelte Werte nach [Clerc, 2006], die häufig zu guten Ergebnissen führen

im Rahmen dieser Arbeit keine PSO für die Optimierung benutzt wird, sondern Partikelschwärme als Grundlage der Simulation benutzt werden. Weitere Informationen zum Konvergenzverhalten einer PSO sind [Maurice Clerc, 2002] und [Trelea, 2003] zu entnehmen.

Oszillation: Ein weiteres Problem ist die Oszillation. Dabei handelt es sich um das Phänomen, dass Partikel sich mit hoher Geschwindigkeit um ein Optimum im Suchraum herum bewegen ohne sich ihm anzunähern, so dass keine Konvergenz eintritt. Dies ist vergleichbar mit dem Kreisen des Mondes um die Erde.

Durch die Einführung der maximalen Geschwindigkeit v_{max} kann die Oszillation bekämpft werden, so fern ein passender Wert für v_{max} gefunden wird. Wie die anderen Parameter auch, hängt der optimale Wert vom Problem ab und setzt Wissen darüber voraus. Zu große Werte für v_{max} lassen den Parameter unwirksam werden, zu kleine Werte verhindern evtl. das Verlassen eines lokalen Optimums.

Vorteile: Die Vorteile der PSO liegen besonders in der Einfachheit, im geringen Speicherplatzbedarf, der geringen Rechenlast und der schnellen Konvergenz in der Nähe von

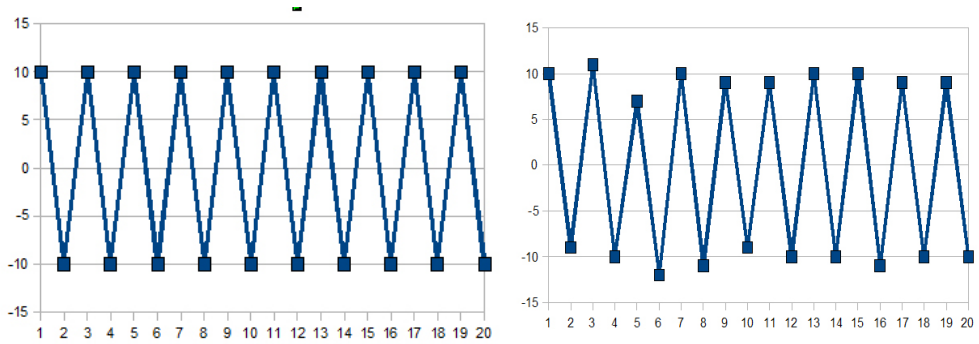


Abbildung 2.10.: Gleichmäßige Oszillation (links) und ungleichmäßige Oszillation (rechts)

Optima. Zudem kann die Konvergenzgeschwindigkeit mit der von evolutionären Algorithmen verglichen werden [Angeline, 1998]. Die PSO lässt sich sehr leicht auf alle möglichen Optimierungsprobleme anwenden, bei denen die Lösungskandidaten mit Hilfe einer Fitnessfunktion bewertet werden können. (siehe EAs - Definition: Optimierungsproblem)

Nachteile: Es gibt eine Reihe von Nachteilen, die die PSO mit sich bringt. Zum Beispiel ist es sehr schwierig die Parametereinstellungen richtig zu wählen, denn diese sind sehr häufig von dem konkreten Problem abhängig und können einen großen Einfluss auf die Qualität der gefundenen Lösungen und/oder die Geschwindigkeit, mit der diese gefunden werden, haben.

Des Weiteren sollte beachtet werden, dass gute Lösungen meistens nur gefunden werden, wenn sie sich innerhalb der konvexen Hülle, die von den Partikel aufgespannt wird, befinden. Durch die ständige Bewegung in Richtung der Nachbarn ist es für ein Partikel am Rande des Schwarms sehr schwierig sich von dem Schwarm weg, also aus ihrer konvexen Hülle hinaus, zu bewegen. Die Eigenbewegung und die zufällige Startbewegung sind die einzigen beiden Komponenten, die einen Partikel dazu veranlassen könnten, die konvexe Hülle zu verlassen. Jedoch wird die Eigenbewegung bei jedem Schritt wieder in Richtung eines Nachbarn abgelenkt und damit in Richtung der konvexen Hülle. Die Bewegung in Richtung der bisher besten Postion und die Bewegung in Richtung des besten Nachbarpartikels können hingegen nicht dazu beitragen die Hülle zu verlassen (Abbilding 2.11).

Dieses Problem wurde innerhalb dieser Arbeit durch theoretische Überlegungen entdeckt und basiert auf keiner Analyse des Problems.

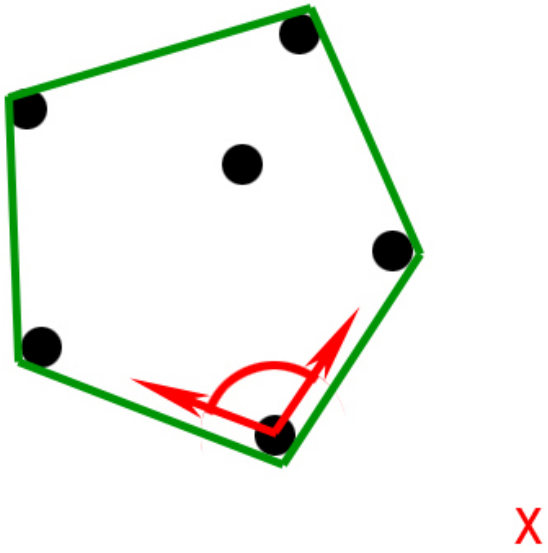


Abbildung 2.11.: Nur am Anfang sind die Chancen für das Verlassen der konvexen Hülle einigermaßen gut. In späteren Phasen wird dies sehr schwierig.

3. Künstliche Intelligenz und Videospiele

KI hat schon früh Einzug in Videospiele gefunden. Zu den ersten Videospielen mit KI gehören Hunt the Wumpus (1972) von Gregory Yob und Star Trek (1972) von Don Daglow[1]. Auch das erste, weltweit populäre Videospiel Pong (1972) von Atari wurde nach seiner Veröffentlichung mit einem Computergegner versehen [Kent, 2001]. Allerdings hatte es die KI damals noch recht einfach. Die Spielwelt von Pong war sehr simpel und die KI musste nur eine von drei möglichen Aktionen auswählen:

- Bewege den Balken nach unten,
- nach oben,
- oder gar nicht.

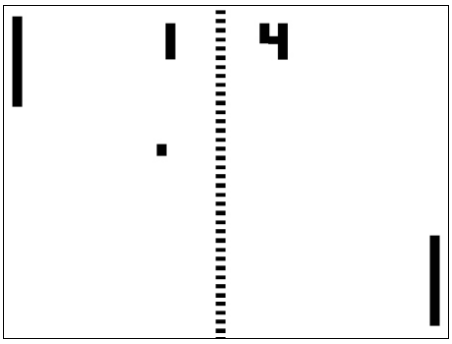


Abbildung 3.1.: Spielwelt von Pong

Seitdem hat sich sehr viel im Bereich der Videospiele getan, nicht nur im Bereich der Grafik und der Spielarten, sondern auch im Bereich der Spiele-KI. Mittlerweile sind die Entscheidungen, die die Spiele-KI treffen muss, längst nicht mehr so simpel wie bei Pong. Die immer komplexer werdenden Spielwelten stellen immer neue Herausforderungen an die Spiele-KI und erweitern ihren Handlungsraum beträchtlich. Die Spiele-KI muss nun aus einer Fülle von Aktionen und nahezu unendlich vielen Kombinationsmöglichkeiten, die beste Entscheidungen treffen. Dies bedeutet jedoch nicht, dass die Spiele-KI immer die effizienteste Entscheidung treffen muss, um zum Beispiel den Spieler zu besiegen. Dies ist nicht das Ziel von Videospielen. Videospiele dienen in erster Linie der Unterhaltung und sollen den Spieler erfreuen, was eine viel schwierigere Aufgabe darstellt. Denn es wäre in vielen Situationen ein Leichtes für eine Spiele-KI den Spieler zu übertreffen. Besonders in Spielen, bei denen es auf Geschick und Präzision ankommt (zum Beispiel

[1] `http://en.allexperts.com/e/g/ga/game_artificial_intelligence.htm`, 23.8.09

First-Person-Shooter (FPS)), wäre es einem menschlichen Spieler unmöglich eine KI zu schlagen.

Seit PONG gab es im Bereich der Spiele-KI zwar beträchtliche Weiterentwicklungen, aber verglichen mit den Verbesserungen in anderen Gebieten, wie der Grafik in Bezug auf eine realistische Darstellung, besteht bei der Spiele-KI noch Nachholbedarf. Diese Aussage trifft jedoch nicht auf alle Bereiche der KI zu. Denn die KI von heute umfasst ein weitläufiges Aufgabenspektrum, welches unterschiedlich gut abgedeckt wird. Einige der Aufgaben wurden bereits zufriedenstellend gelöst, andere noch nicht. Besonders die schon bereits angesprochene Simulation von Lebewesen bedarf deutlicher Verbesserungen.

Im Folgenden werden die Einsatzgebiete der KI und die entsprechenden Anforderungen erläutert, um einen Überblick über die Vielfalt der Aufgaben der KI in Spielen zu liefern. Des Weiteren werden einige neue Konzepte im Bereich der KI vorgestellt. Dabei wird unterschieden zwischen der akademischen Forschung und den in der Praxis tatsächlich eingesetzten Verfahren. Anschließend werden kurz die Gründe für die Unterschiede zwischen Theorie und Praxis genannt.

3.1. Aufgaben der KI in Spielen

Die KI muss im Grunde genommen in der Lage sein alle Entscheidungen, die sonst ein menschlicher Spieler treffen würde, eigenständig zu treffen. Je nach Spiel können die Aufgaben der KI, bzw. ihre Wichtigkeit, variieren. Es gibt jedoch einige Aufgaben die in fast allen Spiele gelöst werden müssen, unabhängig davon, ob es sich nun um ein Actionspiel wie CRYSIS (2007) von Crytec oder ein Sportspiel wie FIFA 06 (2005) von Electronic Arts handelt. Zu diesen Aufgaben gehören:

- Wegfindung
- Terrainanalyse
- Entscheidungsfindung
- Künstliches Leben

Im Folgenden werden diese einzelnen Aufgaben kurz erläutert und ein Überblick darüber gegeben, wie diese in der Theorie und/oder Praxis gelöst werden können. Anschließend wird auf den bisherigen Einsatz von Schwarmintelligenz in Videospielen eingegangen, auch wenn dieser nicht zu den Bereichen zählt, die sehr häufig verwendet werden.

3.1.1. Wegfindung

Bei der Wegfindung soll ein Weg von Punkt A nach Punkt B gefunden und dabei Hindernisse (statische und dynamische) berücksichtigt werden. Der Algorithmus darf nicht zu viele Ressourcen verbrauchen, da er sehr häufig angewendet wird. Es gibt kaum ein Spiel, in dem die Wegfindung nicht zum Einsatz kommt. Das macht diese Aufgabe zu einer der Wichtigsten im Bereich der Spiele-KI.

Üblicherweise wird die Umgebung als Graph dargestellt. Hierzu wird die Umgebung in ein gleichmäßiges Raster unterteilt, dessen Elemente je einen Knoten des Graphen

darstellen. Die Form der Rasterung kann unterschiedlich ausfallen und theoretisch ist jede Form denkbar. Die am häufigsten verwendeten Formen sind Dreiecke, Quadrate, Hexagone und Polygone (für Navigation Meshes, siehe Abbildung 3.3).

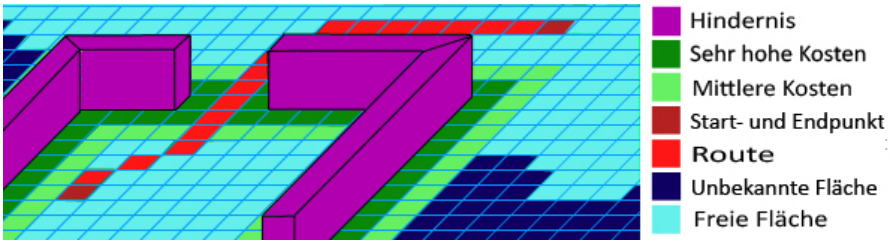


Abbildung 3.2.: Einteilung einer Spielumgebung in Quadrate

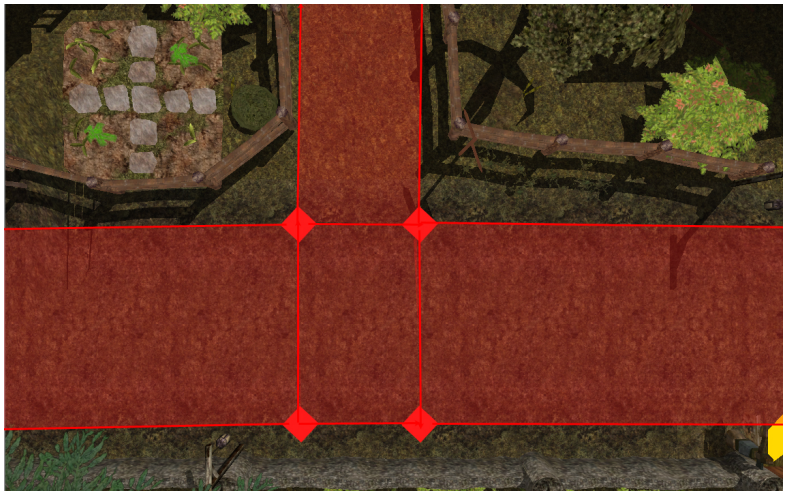


Abbildung 3.3.: Einteilung einer Spielumgebung in ein *Navigation Mesh*

Egal, welche Form gewählt wird, diese Rasterung kann immer als Graph mit Knoten, auch Wegpunkte genannt, und Kanten dargestellt werden. Dabei wird jedem Element des Rasters ein Knoten zugewiesen und über Kanten mit allen Knoten verbunden, die das jeweilige Element umgeben haben. Man unterscheidet bei der Wegfindung zwischen der globalen und der lokalen Wegfindung. Bei ersterer geht es darum den gesamten Weg von A nach B anhand der schon erwähnten Wegpunkte zu planen. Bei der lokalen Wegfindung geht es darum, dynamisch einen Weg zu finden, um von Wegpunkt zu Wegpunkt zu gelangen. Diese Unterteilung wird benutzt, um auf Hindernisse zu reagieren, die während der Offline-Planung nicht vorhanden waren.

Methoden aus der Theorie und Praxis

Der in Videospielen am häufigsten verwendet Algorithmus ist $A*$ (gesprochen: A Star oder A Stern). Generell kann jedoch jeder Algorithmus verwendet werden, der einen Pfad zwischen zwei Knoten findet wie zum Beispiel der Dijkstra-Algorithmus. $A*$ versucht mit

einer Schätzung der restlichen Entfernung immer zuerst die Knoten zu untersuchen, die wahrscheinlich am schnellsten zum Ziel t führen. Dazu bekommt jeder an die Startposition s angrenzende Knoten v einen Wert F zugewiesen, welcher sich aus der Distanz G von s nach v und der geschätzten Distanz H von v nach t zusammensetzt. Die Schätzung H darf in keinem Fall die tatsächlichen Kosten überschreiten. Daher wird im einfachsten Fall für die Schätzung der Distanz H eine Gerade benutzt. Der Knoten mit kleinstem f wird als nächster untersucht [Stuart J. Russell, 2003b].

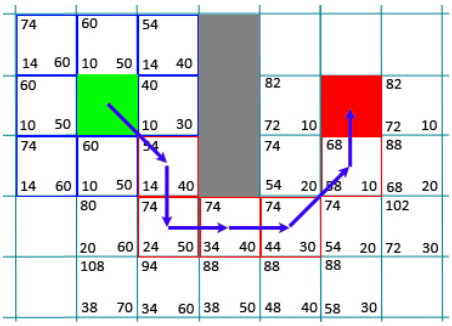


Abbildung 3.4.: Die Kosten F werden links oben, G links unten und H rechts unten angezeigt.

Ein Problem von $A*$ sind die anderen NSCs, welche sich in der Welt bewegen. Versuchen viele NSCs die gleiche Route zu nehmen, kann es zu Engpässen kommen. Je nach Implementierung können diese Hindernisse anders behandelt werden und es kann zu unerwünschten Verhaltensweisen kommen, wie das Auswählen eines viel längeren alternativen Weges, obwohl die anderen NSCs den Weg nur kurzfristig versperren. Eine Abhilfe für dieses Problem soll das *Cooperative Pathfinding* [Silver, 2006] schaffen. Dabei teilen sich naheliegende NSCs ihre Absichten bezüglich ihrer Wege mit, so dass sie diese berücksichtigen können.

Beim *Risk-Adverse Pathfinding Using Influence Maps* [Paanakker, 2006], einer eher für Strategiespiele ausgelegte Methode, werden *Influence Maps*, welche in der Terrainanalyse erläutert werden, benutzt, um bei der Wegfindung feindliche und freundliche Regionen oder Einheiten zu berücksichtigen.

Die *Dynamische Aktualisierung von Navigationsgraphen durch Polygonunterteilung* [Paul Marden, 2008] visiert speziell das Problem von dynamischen Hindernissen an. In modernen Spielen werden immer häufiger Physik-Effekte genutzt, die auch die Umgebung verändern können. So können zum Beispiel Gebäude und Brücken einstürzen oder große Hindernisse wie Felsen oder Bäume einen Weg versperren. Häufig sind Navigationsgraphen statisch und können sich nicht, oder nur unter hohen Kosten, an Veränderungen anpassen. Diese Technik benutzt eine andere Form der Speicherung von Navigationsdaten und erlaubt eine schnelle Aktualisierung durch geschicktes Clipping. Das Ergebnis ist ein Graph der wie gewöhnlich von $A*$ verwendet werden kann.

3.1.2. Terrainanalyse

Bei der Terrainanalyse geht es darum, bestimmte Positionen innerhalb der Spielumgebung nach bestimmten Kriterien zu bewerten. Dabei kann dies entweder statisch (offline) oder dynamisch (online) geschehen. Oft werden für diesen Zweck *Influence Maps* eingesetzt. In Strategiespielen geht es meistens um eine taktische Bewertung der aktuellen und/oder zukünftigen Position auf Basis der aktuellen und schon früher gesammelten Informationen. Die verwendeten Informationen können zum Beispiel Einheitenart- und stärke des Gegners oder günstige strategische Positionen beinhalten. So könnte eine erhöhte geographische Position besser zu verteidigen sein und gleichzeitig eine bessere Sicht ermöglichen, um den heranrückenden Feind frühzeitig zu entdecken. Die Terrainanalyse stellt in der Regel Informationen für andere Aufgaben wie die Weg- und/oder Entscheidungsfindung bereit.

Methoden aus der Theorie und Praxis

Influence Maps basieren auf einer Rasterisierung der Spielumgebung in viele Zellen, in welchen Informationen über das entsprechende Gebiet gespeichert werden. Je nach Spiel werden unterschiedliche Informationen gespeichert. In einem Echtzeit Strategiespiel (engl. Real-time strategie, kurz RTS), einem häufigen Anwendungsgebiet der Influence Maps, werden zum Beispiel Informationen über die Anzahl an feindlichen und freundlichen Einheiten, ihre Stärke, Ressourcen und strategische Wichtigkeit gespeichert. Bei der Implementierung hat man die Wahl, ob man für jeden Spieler eine eigene Influence Map anlegt oder eine allgemeine. Aus Leistungsgründen ist es ratsam nur eine Influence Map zu benutzten, da sie häufig aktualisiert werden muss. In diesem Fall jedoch verwendet die Spiele-KI für ihre Berechnungen Informationen, die sie normalerweise nicht besitzen würde. Der Artikel [Tozour, 2001] enthält weitere Details zum Thema Influence Maps. Spiele die Influence Maps benutzten sind zum Beispiel Age of Empires (1997) von Microsoft-Game-Studiosund und Warcraft III (2002) von Blizzard Entertainment [2].

In dem Artikel „RTS Terrain Analysis: An Image-Processing Approach“ [Julio Obelleiro, 2008] wird ein anderer Ansatz verfolgt. Mit Hilfe einer einfachen Bildverarbeitung werden Informationen über die aktuelle Situation der Umwelt gewonnen und die Terrainanalyse mit der Wegfindung verbunden. Die Analyse besteht aus den Schritten Bildbearbeitung, Transformation, Wegfindung und Datenextration.

3.1.3. Entscheidungsfindung

Bei der Entscheidungsfindung geht es um den Bereich, der am ehesten als „wirkliche“ KI bezeichnet werden kann. In einem Spiel werden Spieler, sowohl die menschlichen als auch die von der KI gesteuerten, vor viele Entscheidungen gestellt. Mit Hilfe der ihnen zur Verfügung stehenden Informationen müssen sie die beste Entscheidung treffen. Je nach Spielegenre müssen unterschiedliche Arten von Entscheidungen getroffen werden.

Beispielhafte Entscheidungen in einem RTS sind:

- Wie sollen die Ressourcen auf Militär und Entwicklung verteilt werden?

[2] `http://www.plm.eecs.uni-kassel.de/ki-in-spielen-dateien/Ki_in_Spielen_Terrain_Analyse_Taktiken_Formationen_und_Kooperation_von_Agenten_Folien.pdf`, 23.8.09

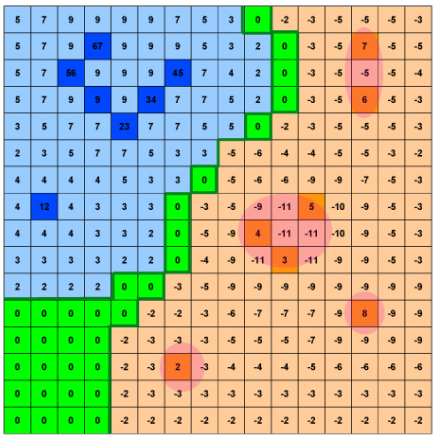


Abbildung 3.5.: Beispiel für eine Influence Map

- Wann und wo soll der Gegner angegriffen werden?
- Welche Positionen sind strategisch Vorteilhaft oder wichtig?
- Welche Einheiten sollen produziert werden?
- Wie erhalte ich Informationen über den Gegner?

Dabei sollten die Entscheidungen, die von der KI getroffen werden, möglichst flexibel und menschenähnlich sein. Außerdem sollte die KI in der Lage sein, unterschiedliche Spielweisen (offensiv, defensiv oder eine Mischung aus beidem) zu beherrschen.

Methoden aus der Theorie und Praxis

Entscheidungsbäume sind eine einfach Möglichkeit, eine KI in einem Spiel Entscheidungen treffen zu lassen. Dabei stellen Knoten Fragestellungen dar, an denen sich der NSC entscheiden muss, welche Aktion er wählen will. Häufig müssen mehrere Aspekte berücksichtigt werden, bevor es zu einer Aktion kommt. Ein einfaches Beispiel soll die Funktionsweise verdeutlichen:

Die Abbildung 3.6 zeigt einen Entscheidungsbaum für einen NSC, der einen Schuss hört. Sobald dieses Ereignis eintritt, wird der Baum aufgerufen und die einzelnen Knoten werden abgearbeitet. Je nachdem wie die aktuelle Situation aussieht, werden andere Aktionen ausgeführt. Der Vorteil von Entscheidungsbäumen ist, dass sie einfach zu verstehen und zu implementieren sind und sehr wenig Ressourcen benötigen. Jedoch können sie für komplexere Verhaltesweisen sehr schnell unübersichtlich und damit schwer zu verwalten werden. Außerdem sind sie sehr unflexibel, denn es können nur solche Situationen oder Zustände behandelt werden, die der Programmierer berücksichtigt hat.

Finite(Fuzzy) State Machines (FSMs) erfreuen sich bei Spielen großer Beliebtheit und werden häufig eingesetzt. Bei FSMs handelt es sich um einen Automaten, der verschiedene Situationen und Aktionen kennt. Zu jedem Zeitpunkt befindet sich der Agent in einen bestimmten Zustand. Abhängig von diesem Zustand führen die zur Verfügung stehenden Aktionen zu anderen Zuständen. Abbildung 3.7 zeigt ein Beispiel für eine FSM.

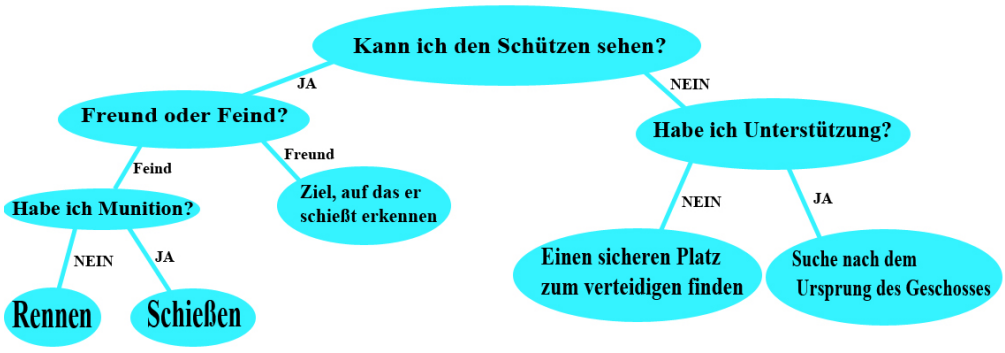


Abbildung 3.6.: Ein einfacher Entscheidungsbaum

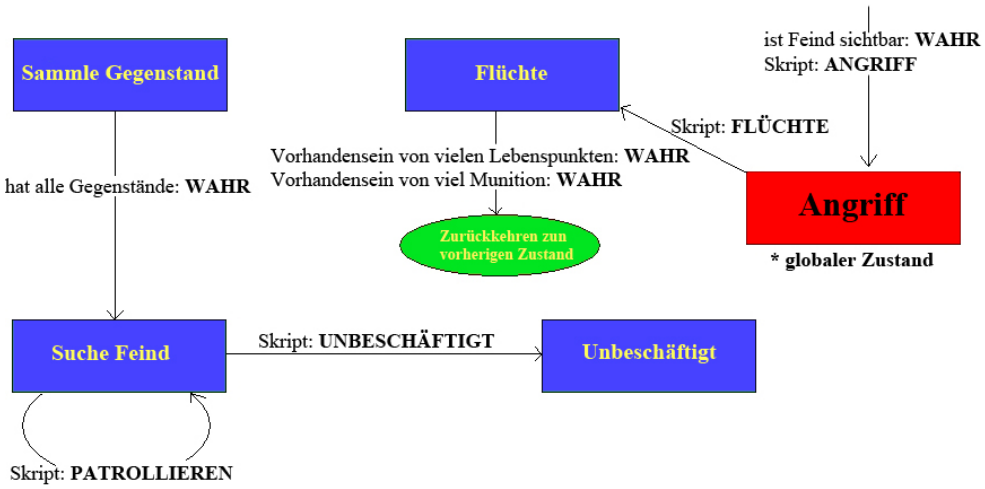


Abbildung 3.7.: Ein Beispiel für eine Finite State Machine

Der Vorteil von FSMs ist, dass sie für einfache Entscheidungen sehr schnell zu implementieren und, im kleinen Rahmen, überschaubar sind. Der Nachteil ist, dass sie bei komplexer werdenden Verhaltensweisen sehr schnell unübersichtlich werden und es so zu schwer erkennbaren Fehlern kommen kann. FSMs werden in allen Genres eingesetzt. Bekannte Beispiele sind QUAKE (1996) von id Software, WARCRAFT II (1995) von Blizzard Entertainment und HALO 2 (2004) von Microsoft Game Studios[3].

Bei *Hierarchical Task Networks* (HTNs) handelt es sich um einen Top-Down-Ansatz, bei dem die NSCs ein High-Level Ziel verfolgen. Dieses Ziel wird dann in mehrere *Teilziele* unterteilt und diese ggf. weiter in *Teilteilziele*. Der Vorteil von HTNs ist die Flexibilität bei der Erfüllung von Zielen. Jedes Ziel hat Vorbedingungen, die erfüllt sein müssen, damit es erreicht werden kann. So muss zum Beispiel ein NSC Essen besitzen, bevor er seinen Hunger stillen kann. Das Teilziel „Essen besorgen“ besitzt ebenfalls Vorbedingungen, die erfüllt werden müssen, usw., bis die Aktionen so atomar geworden sind, dass sie

[3] `http://lyonesse.stanford.edu/~ali/li-constr-gameagent-video-aiide09.pdf`, 23.8.09

direkt ausgeführt werden können [Stuart J. Russell, 2003c].

Viele Spiele verwenden *Skripte*, welche das Verhalten von NSCs im Vorfeld festlegen und von den Programmierern per Hand implementiert wurden. Prinzipiell handelt es sich dabei um fest vorgeschriebene Sequenzen von Aktionen, die durch bestimmte Ereignisse ausgelöst werden. Skripten fehlt es in der Regel an Flexibilität, dafür sind sie aber im Gegenzug recht einfach zu realisieren und erfreuen sich großer Beliebtheit. Der Aufwand wird jedoch schnell sehr hoch, wenn man versucht etwas komplexere Verhaltensweisen zu erzielen. Ein weiterer Vorteil von Skripten ist das hohe Maß an Kontrolle über das Verhalten der NSCs. Gerade in Spielen, die eine Geschichte erzählen, gibt es viele Stellen, an denen sich ein NSC so verhalten soll, wie es der Programmierer geplant hat.

Andere Ansätze sind zum Beispiel *Goal based Action Planning*, *Goal Trees* und *Decision Tree/Finite State Machine Hybride* welche in [Rabin, 2006] und [Rabin, 2008] beschrieben werden.

3.1.4. Künstliches Leben

Hierbei geht es um die Simulation von natürlichen Lebewesen bzw. deren Verhaltensweisen. Ziel ist nicht, den NSC so klug, sondern so realistisch wie möglich wirken zu lassen. Denn je nach simulierten Lebewesen wäre es auch nicht gewünscht, wenn diese extrem intelligent wären. Im Grunde genommen handelt es sich hierbei auch um die Aufgabe der Entscheidungsfindung. Wahrscheinlich ist A-Life die schwerste und am schlechtesten umgesetzte Disziplin der genannten Einsatzgebiete der KI. Bis auf wenige Ausnahmen, wie zum Beispiel TES 4: Oblivion (2006) und Die Sims Reihe (2000 -) von Electronic Arts, gibt es wenig Spiele, von denen man behaupten könnte, sie lösten diese Aufgabe gut. In Kapitel 4 wird noch einmal detaillierter auf das Thema A-Life eingegangen.

Methoden aus der Theorie und Praxis

Mit Hilfe eines *Schedulers* wird in den Spiel TES 4: Oblivion ein 24 Stunden Tagesablauf für alle NSCs erstellt. Sie jagen, essen, schlafen und gehen ihrem Job nach. Dynamische Ereignisse können den Tagesabluf modifzieren, Aktionen unterbrechen, die dann später fortgeführt werden. Das Verfahren ist eher einfach, der Effekt jedoch sehr komplex und glaubwürdig.

Eine weitere, erfolgreiche Technik wird in der Sims Reihe eingesetzt und nennt sich *Smart Terran* und *Smart Objects*. Die KI steckt in den Objekten und der Umgebung und nicht in den NSCs. Jeder NSC hat bestimmte Bedürfnisse wie Hunger, Komfort, Hygiene, Spass usw. Die Sims selber wissen nicht, wie sie diese Bedürfnisse erfüllen können. Es sind die Objekte innerhalb der näheren Umgebung, die den NSCs berichten, wozu sie benutzt werden können. So informiert der Kühlschrank die NSCs darüber, dass es in ihm etwas zu essen gibt und ein Bett signalisiert, dass der NSC sich hier erholen kann. Der NSC wählt anschließend nur die Aktion aus, welche die größte Stimmungsverbesserung bewirkt. Die Animationen für die Benutzung eines Objekts stecken ebenfalls in den Objekten selber. Das ermöglicht den Entwicklern auf einfache Weise neue Gegenstände einzubauen, ohne dass es vorher Animationen dafür geben muss.

3.1.5. Bisherige Einsatzgebiete von SI in Spielen

Der Einzug der Schwarmintelligenz in Spielen ist kein Novum an sich. Sie wird bereits erfolgreich in Spielen, besonders Strategiespielen, eingesetzt. Jedoch wird diese nicht im Rahmen von A-life verwendet, sondern lediglich zur Steuerung von Bewegungen vieler Einheiten, die sich in einem Schwarm bewegen sollen. Dazu bedient man sich der Partikelschwärme, da sich diese am besten für diese Aufgabe eignen, ist doch das natürliche Vorbild von diesen genau das, was man versucht nachzubilden. Abseits der Spielindustrie kommen die Partikelschwärme auch in Filmen zum Einsatz, wie zum Beispiel in der „Der Herr der Ringe“[4] (2001) von Peter Jackson, um die Bewegungen der vielen Akteure in den großen Schlachten zu steuern.

Flocking: Als Flocking bezeichnet man eine Technik für die Simulation von natürlichen Verhaltensweisen bei der Bewegung von vielen Individuen innerhalb einer Gruppe. Diese Technik stellt eine Alternative zu der Berechnung aller individuellen Pfade dar, was einen großen Aufwand, schwere Wart/- und Editierbarkeit zur Folge hätte. Im Grunde genommen funktioniert Flocking nach den Regeln wie sie in Kapitel 2.2.2 in dem Beispiel der natürlichen Fisch- und Vogelschwärme als Vorbild für die PSO erklärt wurde. Flocking ist jedoch keine PSO, da keine Optima o.ä. berechnet werden und zählt eher zu den Partikelsimulationen.

Flocking wurde in vielen Spielen erfolgreich eingesetzt wie zum Beispiel in Half-Life (1998) von Valve Corporation, Unreal (1998) von Epic Games, Theme Hospital (1997) von Electronic Arts und Enemy Nations (1997) von Windward Studios [Davison, 2005].

3.2. Forschungsfeld: Videospiele

In den letzten Jahren sind Videospiele häufiger Gegenstand akademischer Forschungen geworden. Lange galt dieser Bereich als weniger seriös. Allerdings hat man erkannt, dass es in Videospielen viele und sehr schwierige Probleme zu lösen gilt, die durchaus als „normale“ Anwendungsprobleme angesehen werden können. Besonders das Forschungsgebiet der Robotik hat viele Gemeinsamkeiten mit der Entwicklung von Spiele-KIs. Die Voraussetzungen und Eigenschaften beider Bereiche überschneiden sich recht häufig.

Es gibt jedoch in der Forschung zwei grundsätzlich verschiedene Arten von Spielen bzw. Spiele-KIs. Im Folgenden werden die beiden Spielarten als *abstrakte* und *realitätsnahe* Spiele bezeichnet.

3.2.1. Abstrakte Spiele

Die in Kapitel 2.1 vorgestellte Spieltheorie befasst sich mit *abstrakten Spielen* wie Schach, Go, Dame, Mühle, Vier Gewinnt und Tic Tac Toe, die alle auch von Computer gespielt werden können. Diese Spiele unterscheiden sich von den heutzutage üblichen Spielen enorm, denn die Spielwelten bei den abstrakten Spielen sind stark eingeschränkt. In

[4] `http://www.inf.fu-berlin.de/lehre/SS09/KI/folien/vl10.pdf`, s. 18, 17.8.09

[Stuart J. Russell, 2008a] werden verschiedene Eigenschaften von *Arbeitsumgebungen* vorgestellt. Zu diesen zählen:

- *Vollständig oder teilweise beobachtbar:* Bei vollständig beobachtbaren Umgebungen kennt jeder Agent den genauen Zustand der aktuellen Situation, bei nur teilweise beobachtbaren nicht. Der Agent muss Annahmen über unbekannte Zustände treffen.
- *Deterministisch oder stochastisch:* Eine Umgebung ist deterministisch, wenn von vornherein klar ist, welche Aktionen zu welchen Effekten führen. Andernfalls ist sie stochastisch.
- *Statisch oder dynamisch*: Bei statischen Umgebungen können nur die Aktionen von Spielern zu Änderungen der Umgebungen führen. Kann sich die Umgebung über die Zeit, ohne Einfluss der Spieler, verändern, bezeichnet man sie als dynamisch.
- *Diskret oder stetig:* Die Eigenschaft diskret oder stetig kann auf die Umgebung, die Zeit oder die Aktionen angewendet werden. Zum Beispiel besitzt Schach ein diskretes Spielfeld, eine diskrete, endliche Menge von Aktionen und Zuständen. Änderungen der Zustände geschehen sprunghaft, es gibt keine Zwischenschritte bei dem Umsetzen einer Spielfigur. Stetige Umgebungen hingegen unterliegen kontinuierlichen Veränderungen.
- *Einzelagent oder Multiagent:* Eine Umgebung kann von nur einem oder von mehreren Agenten beeinflusst werden. Dabei kann man noch zwischen kooperativen und konkurrierenden Umgebungen unterscheiden.

Spiele wie zum Beispiel Schach oder Go besitzen folgende Eigenschaften:

- Vollständige Informationen
- Deterministisch
- Abwechselnde Reihenfolge der Spieler
- Spieleranzahl = 2
- Nullsummenspiel: Ein Spieler gewinnt(+1), der andere verliert(-1) oder das Spiel endet unentscheiden (± 0) für beide Spieler)
- diskret
- statisch (falls es kein Zeitlimit gibt)

Diese Eigenschaften treffen zum Beispiel auf Schach und Go zu, jedoch nicht auf andere Spiele wie Poker oder Backgammon. Bei Poker besitzt jeder Agent nicht die vollständigen Informationen, während bei Backgammon eine Zufallskomponente, der Würfel, enthalten ist. Dennoch werden diese Spiele zu den *abstrakten Spielen* gezählt, die Gegenstand der Spieltheorie sind. Zwei weitere Eigenschaften, die die meisten Spiele

dieser Art besitzen, sind, dass sie statisch und diskret sind. Beispielsweise sind die Felder eines Schachbrettes diskret und es gibt keine Zwischenschritte bei den Aktionen, die Veränderung der Position einer Spielfigur erfolgt sprunghaft.

Die Spiele-KI in Schachprogrammen kann auch nur in begrenztem Maß als intelligent bezeichnet werden. Denn meistens werden nur durch die zur Verfügung stehende Rechenkraft erschöpfende Suchen durchgeführt, um den Zug zu berechnen, welcher die höchste Gewinnchance aufweist. Zugegebenermaßen werden diese Berechnungen auf sehr intelligente Weise beschleunigt, trotzdem ist das angewendete Prinzip die *Brute-Force-Methode*. Wie in Kapitel 2.1 schon erwähnt, ist IBM selbst der Meinung, dass „Deep Blue" eigentlich keine Intelligenz besitzt, sondern den amtierenden Schachweltmeister nur mit Hilfe seiner Rechenkraft schlagen konnte. Da abstrakte Spiele nicht Gegenstand dieser Arbeit sind, wird hier nicht weiter auf sie eingegangen.

3.2.2. Realitätsnahe Spiele

Realitätsnahe Videospiele sind sehr viel komplexer und offener als die Spiele, die in der Spieltheorie untersucht werden. Nimmt man ein typisches Rollenspiel als Vergleich, so ist die Anzahl an verschiedenen Zuständen der Spielwelt nicht im Entferntesten überschaubar, auch nicht für einen Computer. Zudem gibt es auch viele Entscheidungen, welche nicht mit Hilfe einer erschöpfenden Suche getroffen werden können, jedenfalls nicht in der zur Verfügung stehenden Zeit, denn in allen nicht auf Runden basierenden Spielen ist diese stark eingeschränkt. Zu den Eigenschaften der *realitätsnahen* Spiele zählen:

- Stochastisch: Es kann nicht genau vorhergesagt werden, welchen Effekt eine Aktion haben wird.
- Stetig
- Dynamisch
- Spieler agieren zeitgleich[5].
- Spieleranzahl ist variabel: Nicht nur menschliche Spieler nehmen an dem Spiel teil, sondern auch NSCs[6].
- Kein Nullsummenspiel: Der Sieg eines Spielers muss nicht die Niederlage eines anderen Spieler zur Folge haben. Es können kooperative und konkurrierende Spieler aufeinander treffen.
- Teilweise beobachtbar: Die meisten Spieler besitzen nicht alle Informationen.

Nach Russel und Norvig ist diese Konstellation von Eigenschaften einer der schwierigsten Fälle, die auftreten können [Stuart J. Russell, 2008a].

Da eine erschöpfende Suche für die Entscheidungsfindung nicht angewendet werden kann, müssen an andere Techniken entwickelt werden um die Aktionen der Agenten zu

[5] Ausnahmen sind zum Beispiel Runden basierende Spiele.

[6] Bei den Massive Multiplayer Online Roleplaying Games wie World of Warcraft (2004) von Blizzard Entertainmentspielen mehrere Tausend (menschliche) Spieler miteinander und mehrere Tausend NSCs.

bestimmen. In Kapitel 3.1 wurde auf einige Ansätze und bestehende Techniken bereits eingegangen und mit dieser Arbeit soll untersucht werden, ob und wie sich die Schwarmintelligenz dafür eignet.

3.3. Geschichte der Spiele-KI

Die ersten KIs in Videospielen sind in den 1970er aufgetaucht. Alle Spiele vor dieser Zeit boten keine Spiele-KI und konnten nur im Mehrspielermodus von menschlichen Spielern gespielt werden. Seit den Anfängen hat sich sehr viel getan im Bereich der Spiele-KI. Besonders die immer größer werdende Rechenkapazität begünstigte die Entwicklung der KI. Denn wenn es um diese ging, musste die KI immer Einschnitte hinnehmen, da die Leistung hauptsächlich der Grafik gewidmet wurde bzw. immer noch wird.

Hier eine unvollständige Liste von Spielen, die bestimmte KI Techniken als erstes, oder erfolgreich eingesetzt haben.

- 1974 - Qwak! von Team17: Eines der ersten Videospiele mit einer KI
- 1980 - PacMan von Namco: PacMan benutzt vier unterschiedliche KI Typen. Typ 1 bewegt sich direkt auf den Spieler zu. Typ 2 versucht den Spieler abzufangen, also sich an die zukünftige Position des Spielers zu bewegen. Typ 3 flieht vor den Spieler und Typ 4 ignoriert ihn.
- 1989 - SimCity von Maxis: Benutzt zelluläre Automaten und Influence Maps um zu ermitteln, wann und wie die Gebäude sich entwickeln.
- 1992 - Dune II von Westwood Studios: Benutzte als eines der ersten Spiele FSMs[7].
- 1996 - Tamigotchi von Aki Maita: A-Life-Simulation eines Lebewesens, das Befürnisse hat die von dem Spieler erfüllt werden müssen. Battlecruiser 3000AD, Creatures: Einsatz von Neuronalen Netzen für die Entscheidungsfindung und genetischen Algorithmen.
- 1998 - Half-Life von Valve Corporation: Kooperatives Verhalten der Computergegner, die Taktiken wie Einkreisen beherrschen.
- 2000 - Die Sims von Electronic Arts. A-Life-Simulation, die sog. Smart Terrain und Smart Objects verwendet.
- 2001 - Black & White von Lionhead Studios: Maschinelles Lernen als Spielprinzip. Der Spieler trainiert eine Kreatur mittels Belohnung/Bestrafung. Zum Einsatz kommen Neuronale Netze und Entscheidungsbäume.
- 2005 - FEAR von Monolith Productions: Nutzt ein Planungssystem ähnlich wie STRIPS[8]. Die AI Gegner arbeiten im Team und beherrschen einige Taktiken[9].

[7] `http://www.actiontrip.com/features/briefhistoryofvideogameai.phtml`, 23.8.09

[8] Für Informationen über Stanford Research Institute Problem Solver (STRIPS) von Richard Fikes und Nils Nilsson s. [Stuart J. Russell, 2008b]

[9] `http://web.media.mit.edu/~jorkin/gdc2006_orkin_jeff_fear.pdf`, 23.8.09

- 2006 - TES 4: Oblivion von Bethesda Softworks: Gelungene A-Life Umsetzung. NSCs nutzen Scheduler für die Planung ihres Tagesablaufes. Ereignisse können Tagesablauf ändern.

- 2007 - S.T.A.L.K.E.R von GSC Game World: NSCs nutzen GOAP[10] [Orkin, 2003] für die Aktions- und Motivational Graphs [Julien Devade, 2003] für die Aufgabenwahl[11].

- 2008 - Spore von Maxis: Benutzt Flüssigkeits- und Partikelsimulationen, Flocking und Verhaltens-bäume [12].

Auf der Internetseite *AI Game Dev*[13] gibt es viele weitere Informationen über die Geschichte der Spiele-KI und die neusten Entwicklungen. In dem Artikel „Top 10 Most Influential AI Games" werden die, aus Sicht des Autors, wichtigsten Spiele bezüglich ihrer Spiele-KI vorgestellt [Champandard, 2007].

3.4. Die Kluft zwischen der akademischen Forschung und der Spieleindustrie und ihre Gründe

In vielen Artikeln und Büchern zu Spiele-KI in Spielen liest man, dass die Spiele-KI in den letzten Jahren immer wichtiger geworden sei. Trotzdem gibt es kaum Spiele, die diesen *Trend* auch wirklich umsetzen. Während das akademische Interesse an Spielen in den letzten Jahren stark angestiegen ist, ist das Interesse der Spielindustrie an den Fortschritten der Forscher nur sehr begrenzt. In dem Artikel „We're Not Listening: An Open Letter to Academic Game Researchers" von John Hopson (2006). John Hopson ist ein Spieleforscher bei *Microsoft Game Studios* und wirkte bei bekannten Spielen wie Halo und Age of Empires mit[14]. In diesem Artikel werden die Gründe für diese große Lücke zwischen der Forschung und der Praxis durchleuchtet. Zu den wichtigsten zählen:

- *Return On Investment:* Videospiele werden produziert, um Gewinn zu erziehen und jede Stunde Arbeit, die in einem Produkt steckt, erhöht die Kosten. Für die Industrie ist es wichtig zu sehen, welche Anforderungen (besonders die benötigte Zeit) und welche Auswirkungen eine neue Technik auf ihr Projekt hat. Der Hintergrund, warum diese Technik funktioniert, ist meistens nicht von Interesse. Jedoch wird gerade auf diesen Aspekt oft sehr ausführlich eingegangen.

- *Die richtige Sprache sprechen:* Die Sprache in akademischen Artikeln und Vorträgen ist für Nicht-Akademiker nur sehr schwer zu verstehen. Die meisten wollen oder können sich die Fähigkeit, diese Sprache zu verstehen auch nicht aneignen. Daher ist es ratsam, die Artikel und Vorträge in einer verständlichen Sprache zu präsentieren.

[10] Goal Oriented Action Planning
[11] `http://aigamedev.com/open/interviews/stalker-alife/`, 23.8.09
[12] `http://dankline.wordpress.com/2008/10/30/aiide-2008-day-2-live-blogging/`, 23.8.09
[13] `http://aigamedev.com`, 23.8.09
[14] `http://gamasutra.com/features/20061110/hopson_01.shtml`, 31.8.09

- *Modularität und benötigte Zeit:* Ein kritischer Aspekt ist die benötigte Zeit und Personal für die Umsetzung einer Idee. Abhängig von dem Entwicklungsstand des Spiels, ist es oft nicht möglich größere Änderungen vorzunehmen. Während der frühen Phasen der Entwicklung sind die Konzepte einfacher umzusetzen, sie muss jedoch schnell und einfach in einem kleinem Rahmen zu testen sein, denn nur die wenigsten Entwickler können es sich leisten viel Zeit in eine Technik zu investieren, die möglicherweise nicht funktioniert oder sich nicht eignet.

- *How To:* Aus der akademischen Arbeit muss hervorgehen, wie die erklärte Technik umzusetzen ist. Es reicht nicht, ein Problem zu beschreiben, es muss auch gelöst werden. Die Konsequenzen oder Lösungsvorschläge aus entdeckten Erkenntnissen müssen präzise formuliert und die Schlussfolgerungen nicht dem Leser überlassen werden.

- *Beweise:* Es gibt viele Leute, die viele (gute) Ideen haben, wie man Videospiele verbessern kann. Daher ist es wichtig, dass bei neuen Ideen immer anhand von Beispielen gezeigt wird, wie sie funktionieren.

- *Fokus auf das Richtige:* Viele Akademiker forschen an Techniken, die für die Industrie nicht so relevant sind, wie sie glauben. Egal wie gut oder schlecht ihre Arbeit ist, wenn sich die Industrie für diesen Bereich nicht interessiert, wird die Arbeit nicht wahrgenommen. Ein einfacher Weg, wie die Entwicklung auf die wichtigen Bereiche gelenkt wird, ist die Spieleindustrie zu fragen, für welche Probleme sie eine Lösung braucht.

Andere Gründe dafür liegen in der Komplexität moderner Spiele und in dem von den Entwicklern unterschätzten Aufwand, der für eine gute Spiele-KI nötig wäre[15]. Oft fällt erst am Ende der Entwicklung auf, dass es noch gar keine richtige Spiele-KI gibt und diese muss dann in den letzten Monaten vor der Veröffentlichung schnell entwickelt und implementiert werden. Probleme, die auf Grund der schon vorhandenen Spiel-Engine[16] entstehen, können meistens nicht mehr behoben werden und müssen dann „irgendwie“ gelöst werden, was meistens zu eher uneleganten Lösungen führt, zumal diejenigen, die die Spiele-KI programmieren, nicht selten das erste mal mit dieser Aufgabe konfrontiert werden. Unter den Spiele-KI Programmierern ist man sich einig: Um eine erfolgreiche Spiele-KI zu entwickeln, muss diese an der Entwicklung der Spiel-Engine beteiligt sein.

Nach Frank Gwosdz, Mitbegründer der Firma Artificial, welche seit 2005 Spiele-KI-Software für Spielestudios entwickelt, ist ein weiteres, schwerwiegendes Problem das Fehlen von einheitlichen Tools für die Entwicklung der Spiele-KI eines konkreten Spiels[17].

Nicht zuletzt ist das Fehlen von guter Vermarktbarkeit ein Grund für das schlechte KI Niveau. Sicherlich sollte das Versprechen des Herstellers, eine innovative KI entwickelt zu haben, welche den Spielspaß durchaus anheben kann, Kunden anlocken. Jedoch kann

[15] `http://www.tecchannel.de/webtechnik/entwicklung/1744817/warum_kuenstliche_intelligenz_ki_in_spielen_stagniert/`, 20.10.09

[16] Die Spiel-Engine beschreibt den Teil des Spielprogramms, welcher für die Steuerung des internen Ablaufs verantwortlich ist, also das Basisprogramm des Spiels.

[17] `http://www.tecchannel.de/webtechnik/entwicklung/1744817/warum_kuenstliche_intelligenz_ki_in_spielen_stagniert/`, 20.10.09

die tatsächliche Umsetzung und Qualität erst während des Spielens festgestellt werden und das auch nicht innerhalb der ersten Minuten. Aspekte wie die Grafik eines Spiels können eindrucksvoll mit Hilfe von Bildern auf der Verpackung und Videos dargestellt werden und ziehen damit um ein vielfaches mehr Kunden an als die KI.

4. Schwarmintelligenz in Videospielen

Dieses Kapitel beschäftigt sich mit den Anforderungen und der Umsetzung einer A-Life-Simulation mit Hilfe von SI-Techniken. Zunächst werden die Anforderungen, die an eine A-Life-Simulation gestellt werden, erläutert und anschließend die Idee dargestellt, wie diese Anforderungen erfüllt werden sollen.

4.1. Schwarmintelligenz und A-Life

In diesem Abschnitt soll auf die Besonderheiten und Probleme hingewiesen werden, die bei einer A-Life-Simulation auftreten und darauf, wie diese gelöst werden können.

4.1.1. Anforderungen

Ziel einer A-Life-Simulation ist es, eine realistische bzw. glaubhafte Simulation von Lebewesen zu erreichen. Um zu evaluieren, ob das Ziel erreicht wurde, benötigt man einen *Maßstab*, nach dem man den Grad der Erfüllung bewerten kann. An dieser Stelle sei erwähnt, dass es nicht das Ziel dieser Arbeit ist, eine KI zu entwickeln, die diese Anforderung erfüllt, sondern zu untersuchen, in wie weit die SI bei der Annäherung an dieses Ziel behilflich sein kann.

Im Folgenden soll die Idee, wie dieser Maßstab mit Hilfe von Fitnessfunktionen erstellt werden kann, beschrieben werden.

Fitnessfunktionen

Der Maßstab nach dem eine A-Life-Simulation bewertet wird, ist der Grad an Realismus. Die Bewertung des Realismus kann jedoch nur von einem Menschen durchgeführt werden, da sie das subjektive Empfinden des Betrachters widerspiegelt. Dieses subjektive Empfinden führt zu unterschiedlichen Bewertungen.

Dieser Umstand macht eine automatische Bewertung und anschließende Verbesserung sehr schwierig, da die gängigen Optimierungsverfahren eine berechenbare Funktion voraussetzen, die hier nicht gegeben ist. In diesem Falle benötigte man eine Fitnessfunktion für den Grad an Realismus.

Da im Rahmen dieser Arbeit kein Weg gefunden wurde, eine Fitnessfunktion für Realismus zu erstellen, wurde eine Alternative entwickelt, um die A-Life-Simulation zu verbessern. Diese Alternative zur Bewertung des Gesamtverhaltens sieht es vor, das Gesamtverhalten nicht mit Hilfe einer einzelnen Funktion zu bewerten, sondern das Gesamtverhalten in viele, einzelne Verhaltensmuster aufzuspalten und eine Bewertung dieser kleineren Verhaltensmuster durchzuführen. Die Idee dahinter ist die, dass wenn die einzelnen Teile des Gesamtverhaltens realistisch sind, dann auch die Summe der einzelnen Verhaltensmuster realistisch ist.

Zunächst muss überlegt werden, welche verschiedenen NSC-Gruppen simuliert werden sollen. Im Rahmen dieser Arbeit werden fünf verschiedene Gruppen simuliert: *Bauern*, *Wachen*, *Goblins*, *Raub-* und *Beutetiere*. Jeder Gruppe müssen anschließend Verhaltensmuster zugewiesen werden und jedes Verhaltensmuster benötigt eine Fitnessfunktion, mit welcher das Erfüllen dieser Verhaltensmuster gemessen werden kann. Dabei sollen die jeweiligen Fitnessfunktionen der Verhaltensmuster hohe Werte erzeugen, falls das Verhaltensmuster zufriedenstellend erfüllt wurde und niedrige Werte erzeugen, falls das Verhaltensmuster nicht zufriedenstellend erfüllt wurde.

Es folgt eine Auflistung der Verhaltensmuster der in dieser Arbeit verwendeten NSC-Gruppen:

- *Bauern*
  - Gefahren meiden (Raubtiere, Goblins)
  - Optimierung des Tagesablaufs (Fitness des Einzelnen)
  - Bedürfnisse erfüllen (Nahrung, Wasser, Schlaf)
- *Wachen*
  - Bauern beschützen
  - Jagen (Goblins)
  - Bedürfnisse erfüllen (Nahrung, Wasser, Schlaf)
- *Goblins*
  - Gefahren meiden (Wachen)
  - Jagen (Beutetiere, Bauern)
  - Bedürfnisse erfüllen (Nahrung, Wasser, Schlaf)
- *Beutetiere*
  - Gefahren meiden (Raubtiere, Goblins, Wachen, Bauern)
  - Bedürfnisse erfüllen (Nahrung, Wasser, Schlaf)
- *Raubtiere*
  - Gefahren meiden (Goblins, Wachen)
  - Jagen (Beutetiere)
  - Bedürfnisse erfüllen (Nahrung, Wasser, Schlaf)

Im Folgenden werden die Aspekte, die bei der Bewertung der Verhaltensmuster eine Rolle spielen, vorgestellt:

- *Gefahren meiden*
  - Erlittene Verletzungen
  - Distanz zu Gefahrenzonen
  - Aktivierungsdauer des Fluchtverhaltens, Distanz zu Feinden

- *Optimierung des Tagesablaufs*
  - Aktivierungszeit der Aktionsregeln, die den Tagesablauf steuern im Verhältnis zur maximalen Aktivierungszeit
- *Bedürfnisse*
  - ausreichend erfüllt
  - nicht ausreichend erfüllt
- *Beschützen*
  - Anzahl der Aktivierungen des Fluchtverhaltens der zu beschützenden Gruppe
  - Anzahl der Verletzungen, die der Gruppe Bauern zugefügt wurde
  - Eindringen von feindlichen Agenten in die zu bewachende Zone
- *Jagen*
  - Dauer der Jagdzeit im Verhältnis zur Menge der erlegten Beute
  - Anzahl der Beute im Verhältnis zu den zu jagenden Agenten.

Nachdem die Fitnessfunktionen der einzelnen Verhaltensmuster aufgestellt worden sind, kann versucht werden, diese wieder mit Hilfe von passenden Verknüpfungen und Gewichtung zu einer großen Funktion zusammenzuführen. Damit würde man erreichen den Grad der Erfüllung für die aufgeführten Verhaltensmuster in eine einzelne Funktion zu überführen. Angenommen, diese Verhaltensmuster spiegeln das Verhalten wieder, welches der Entwickler als realistisches Gesamtverhalten ansieht, so könnte man so eine Bewertung für den Grad an Realismus aus Sicht des Entwicklers erreichen.

Allerdings gibt es bei diesem Versuch, die Fitnessfunktionen zusammenzuführen, einige Probleme. Im Rahmen dieser Arbeit ist es nicht gelungen, diese Probleme, die bei der Verknüpfung von den Fitnessfunktionen auftreten, zu lösen. Das größte Problem bei der Zusammenführung der Fitnessfunktionen ist ihre Normalisierung.

Eine Alternative zur Verknüpfung der einzelnen Fitnessfunktionen der Verhaltensmuster ist ihre separate Betrachtung. Im Folgenden werden zunächst das Problem der Normalisierung und dann die Probleme, die bei den möglichen Verknüpfungen auftreten, erläutert.

Normalisierung: Eine Normalisierung muss nur stattfinden, wenn die einzelnen Fitnessfunktionen später wieder zusammengeführt werden sollen. Jedoch ist dies eine sehr schwierige Aufgabe, da zu Beginn nicht klar ist, welche Werte für die einzelnen Funktionen gute oder schlechte Werte darstellen. Je nach Beschaffenheit der Funktionen können die Werte auch von Fitnessfunktion zu Fitnessfunktion sehr unterschiedlich ausfallen, so dass bei der einen ein Funktionswert von *10* einen guten Wert darstellen würde, bei der anderen aber erst ein Wert ab *10.000* als gut angesehen werden könnte. Eine Normalisierung könnte also nur anhand der bisher gefundenen Werte für die jeweiligen Fitnessfunktionen erfolgen. Angenommen, nach einigen Testläufen würden Werte zwischen *10* und *100* ermittelt, so könnte man das Maximum *100* als Basis für die Normalisierung verwenden. Ein Wert von *50* würde dann als durchschnittlich guter Wert eingestuft werden.

Allerdings gäbe es keine Garantie, dass der Wert von *100* sich im Bereich der optimalen Lösungen aufhalten würde. Es wäre durchaus möglich, dass ein Wert von *1.000* zu den optimalen Lösungen zählen würde, wegen schlechter Parameter dieser aber nicht annähernd erreicht wurde. Ist das tatsächliche Maximum also *1000* statt *100*, so wäre der aktuelle Wert von *50* als schlecht einzustufen. Wenn dann ein höherer Maximalwert gefunden würde, müsste dieser als neue Basis bestimmt werden. Jedoch würde das bedeuten, dass die vorangegangenen Berechnungen mit einem Wert von *100* als Basis zu falschen Ergebnissen geführt hätte, denn der Wert von *50* wurde zuvor als durchschnittlich angesehen und nicht als sehr schlecht. Alle Entscheidungen, die also auf der Einstufung dieser Werte beruhten, wären von falschen Annahmen ausgegangen und demnach nicht korrekt.

Verknüpfungen: Es liegt also maßgeblich an der Art der Verknüpfungen der Fitnessfunktionen untereinander, ob eine Normalisierung, in welcher Form auch immer, benötigt wird oder nicht. Verfahren, die ohne eine Normalisierung auskommen, sind wegen der oben aufgezeigten Problematik zu bevorzugen. Mögliche Verknüpfungen sind folgende:

- *UND (Multiplikation)*
  Eine UND-Verknüpfung bevorzugt die Agenten, welche alle Verhaltensmuster mittelmäßig erfüllen. Besonders Ausreißer nach unten haben einen starken Einfluss auf die Gesamtfitness, während Ausreißer nach oben keinen adequaten Einfluss haben. Ein kleines Zahlenbeispiel verdeutlicht dies: *(0,1 * 0,9 = 0,09) < (0,5 * 0,5 = 0,25)*

  Eine Normalisierung der einzelnen Fitnesswerte ist unerlässlich für dieses Verfahren, da sonst Verhaltensmuster, die strukturell bedingt hohe Werte erzeugen, andere Verhaltensmuster dominieren würden. Nach der Normalisierung könnten die einzelnen Faktoren je nach ihrer Wichtigkeit mit Gewichtung versehen werden.

- *ODER (Addition)*
  Bei einer Addition der einzelnen Fintnessfunktionen ist eine Normalisierung ebenso wichtig wie bei der Multiplikation. Die Gründe hierfür sind die gleichen. Der Unterschied zur Multiplikation besteht darin, dass das extreme Erfüllen von Verhaltensmustern nicht zu schlechteren Werten führt als die durchschnittliche Erfüllung aller Funktionen. Wird ein Muster gar nicht und ein anderes voll erfüllt, würde dies zu einer mittelmäßigen Bewertung führen.

- *Min, Max*
  Benutzt man das Minimum aller Funktionen, kommt es nur auf die niedrigsten Werte an. Anders als bei der Multiplikation oder Addition, kann kein Ausgleich durch die überdurchschnittlich hohe Erfüllung anderer Funktionen erzielt werden. Wie bei den Methoden zuvor, ist eine Normalisierung unverzichtbar. Im Rahmen dieser Arbeit werden diese Arten der Verknüpfungen nicht weiter betrachtet.

- *Separate Betrachtung*
  Eine weitere Möglichkeit wäre es, jede Fitnessfunktion getrennt zu betrachten. Der Vorteil wäre, dass auf die schwer zu realisierende Normalisierung verzichtet werden kann. Allerdings wird dieser Vorteil dadurch erkauft, dass es nicht nur einen einzelnen Wert für die Bewertung des Verhaltens gibt, sondern mehrere. Da die

Anzahl der Verhaltensmuster sich jedoch in einem Rahmen von 1 bis 10 bewegt, sollte dies keine Probleme verursachen. Zusätzlich würde der Beobachter erkennen können, welches Verhaltensmuster gut und welche weniger gut erfüllt werden. Eine Gewichtung der verschiedenen Funktionen entfällt ebenfalls. Es ist jedoch zu beachten, dass ohne Normalisierung nicht sichergestellt werden kann, ob nun ein Verhaltensmuster gut oder schlecht erfüllt wurde, sondern nur, ob die Veränderungen der Regeln zu besseren oder schlechteren Ergebnissen geführt haben. Jedoch ist zu beachten, dass sich die Regeln der einzelnen Verhaltensmuster gegenseitig beeinflussen können. Abbildung 4.1 zeigt dieses Phänomen. Optimiert man nur eine Fitnessfunktion zur gleichen Zeit, kann ausgeschlossen werden, dass die veränderten Parameter einer anderen Funktion die gemessene Veränderung hervorgerufen haben.

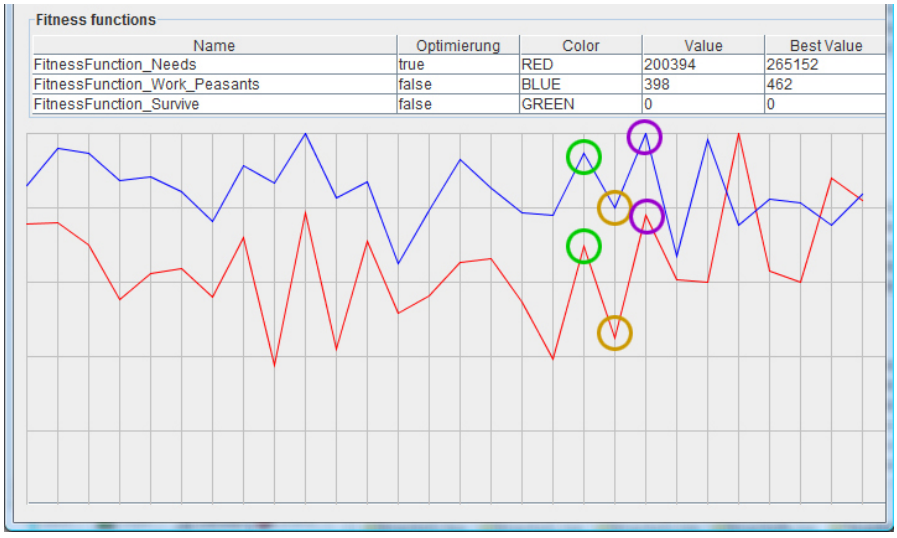


Abbildung 4.1.: Die markierten Stellen zeigen die sich gegenseitig beeinflussende Wirkung. Die rote Kurve wird einer Optimierung unterzogen, die blaue Kurve nicht. Trotzdem folgt die blaue Kurve dem Verlauf der roten Kurve.

In dieser Arbeit wurden zwei verschiedene Ansätze entwickelt, welche das Ziel haben eine oder mehrere Fitnessfunktionen zu erstellen um den Grad an Realismus zu messen. Der erste Ansatz benutzt dabei eine UND-Verknüpfungen der einzelnen Fitnessfunktionen, der zweite Ansatz verwendet keine Verknüpfung und betrachtet die Verhaltensmuster separat. Eine detaillierte Beschreibung der Ansätze wird in Kapitel 4.3.2 gegeben.

Nachdem die Anforderungen und das Aufstellen eines Maßstabes zu derer Erfüllung erläutert wurden, wird nun die Idee vorgestellt, wie die Simulation aufgebaut werden soll.

4.2. Grundidee: SI als Spiele-KI

Es gibt zwei verschiedene Ansätze für den Aufbau der Simulation, zwischen denen gewählt werden kann. Der erste behandelt jeden NSC in der Simulation als eigenständigen

Partikel, welcher nach bestimmten Regeln funktioniert, ähnlich wie die Fische in dem Beispiel zur PSO, und stellt im eigentlichen Sinne keine PSO, sondern einen Partikelschwarm dar.

Die zweite Möglichkeit ist die Bestimmung des kognitiven Prozesses eines NSCs mit Hilfe einer PSO. Zwar würden die Agenten auf diese Weise SI nutzen, jedoch entspräche dieser Ansatz nicht dem Prinzip der Schwarmintelligenz, bei der die einzelnen Agenten nach einfachen Regeln funktionieren, sondern würde sie weitaus komplexer machen. Außerdem stellt die PSO ein Optimierungsverfahren dar und man benötigte zunächst eine Eingabe und eine Fitnessfunktion, die optimiert werden soll. Zudem wäre der Austausch des Optimierungsverfahren keine wesentliche Änderung und sorgt in der Regel nur für veränderte Laufzeiten, aber nicht für grundlegend anderen Lösungen. Daher wird der erste Ansatz gewählt.

Zunächst soll aber erst genauer auf den Unterschied zwischen Partikelschwärmen (PS) und der Partikelschwarmoptimierung (PSO) eingegangen werden.

4.2.1. Partikelschwärme und Partikelschwarmoptimierung

Bei den Partikelschwärmen geht es um die Simulation von einer großen Anzahl von mehr oder weniger homogenen Partikeln, die sich untereinander beeinflussen und mit ihrer Umwelt interagieren.

Im Grunde genommen sind es einfache, regelbasierte Systeme. Beispiele für Partikel wären zum Beispiel die Fische eines Fischschwarms (siehe Kapitel 2.2.2) oder eine Menge von Wassermolekülen in einer Flüssigkeitssimulation. Bei diesen Simulationen ist das Ziel eine originalgetreue Abbildung der Realität, um zum Beispiel Experimente mit Flüssigkeiten mit Hilfe von Computern zu simulieren, damit keine echten Teststände aufgebaut werden müssen.

Partikelschwärme haben kein klar definiertes Programmende, wie es bei der Berechnung eines Wertes, zum Beispiel dem Abstand zweier Punkte, der Fall ist. Allenfalls könnte man den Stillstand aller Partikel als Ende der Simulation auffassen, falls es keine äußeren Einflüsse gäbe, welche die Partikel wieder in Bewegung versetzen könnten. Meistens möchte man jedoch das Verhalten der Partikel über einen bestimmten Zeitraum in bestimmten Situationen beobachten. Für diesen Zweck wählt man einen Startzustand und einen gewissen Zeitraum aus, die als Start und Endpunkt der Simulation dienen.

Die PSO hingegen ist eine besondere Form eines Partikelschwarms mit speziellen Regeln, welche ein bestimmtes Ziel verfolgt. Sie wird eingesetzt um Optima einer Funktion zu bestimmen. Meistens handelt es sich um unbekannte oder sehr komplexe Funktionen, für die eine exakte Berechnung der Optima zu aufwendig oder unmöglich wäre. Da die Optima nicht bekannt sind, kann auch nicht beurteilt werden, ob die bisher gefundenen Lösungen globale oder nur lokale Optima darstellen; ein allgemeines Problem von Approximationsverfahren. Demnach terminieren PSOs ebenfalls nicht, es sei denn es werden ein oder mehrere künstliche Abbruchkriterien hinzugefügt. Im Allgemeinen handelt es sich dabei, um das Überschreiten einer gewissen Rechenzeit oder des Schwellwerts der besten, bisher gefundenen Lösung.

In dieser Arbeit wird für die Simulation von Lebewesen keine PSO, sondern es werden Partikelschwärme eingesetzt. Das Besondere gegenüber normalen regelbasierten Systemen werden die verwendeten Regeln sein. Es sollen möglichst einfache Regeln verwendet

werden, welche durch die Interaktion untereinander ein emergentes Verhalten hervorrufen, mit dem o.g. Ziel der realistischen Simulation. Im Grunde genommen müssen Regeln ähnlich denen der anderen SI-Verfahren entwickelt werden. Diese Regeln sollen allerdings ein anderes Verhalten hervorrufen, als die Optimierung von Funktionen.

Im Folgenden wird das in dieser Arbeit verwendete regelbasierte System vorgestellt. Dabei wird zuerst die Originalversion von Matt Gilgenbach und Travis McIntosh [Matt Gilgenbach, 2006] vorgestellt und anschließend auf die Änderungen des Systems im Rahmen dieser Arbeit eingegangen.

4.3. Regelmanager-System

Als Grundlage für die Simulation dient ein einfaches Regelsystem. Der Aufbau ähnelt dem einer PS und ist somit als Basis sehr gut geeignet. Ursprünglich wurde dieses System von den Autoren entwickelt, um schon vorher definierte Verhaltensweisen für mehrere NSCs wiederzuverwenden und so den Aufwand für die Entwicklung der Spiele-KI zu reduzieren. Das System ist für eher reaktives Verhalten ausgelegt und eignet sich laut Autor nicht dafür komplexe Verhaltensweisen (High-Level Verhalten) zu implementieren. Diese Eigenschaft kann als *Sicherung* dienen, damit kein High-Level Verhalten schon vorher festgelegt wird, sondern eine Folge der Interaktionen der Verhaltensregeln ist.

Das Grundsystem besteht aus zwei Hauptkomponenten, dem *Regelmanager* und den *Verhaltensregeln*. Jede Kreatur besitzt einen *Regelmanager*, der die verschiedenen Verhaltensweisen eines NSCs überwacht und verwaltet. Er unterhält jeweils eine Liste für Verhaltensregeln und exklusive Verhaltensregeln, welche zusammen das Gesamtverhalten des NSCs bestimmen. In regelmäßigen Zeitabständen (zum Beispiel alle 2 Sekunden) werden die Verhaltensregeln von dem Regelmanager auf ihre Ausführbarkeit geprüft und ggf. ausgeführt.

Eine *Verhaltensregel* besitzt Anweisungen in Form von Aktionen und Zuständen, die für das jeweilige Verhalten, wie zum Beispiel das Verfolgen eines anderen NSCs, notwendig sind. Eine Aktion beschreibt eine physische Handlung eines NSCs wie zum Beispiel „Laufen“ oder „Angreifen“. Meistens handelt es sich bei den Aktionen um sog. atomare Aktionen, also jene, welche unmittelbar durch den NSC ausgeführt werden könnten, wie etwa das „Bewegen von A nach B“ oder das „Angreifen eines Zieles“ in unmittelbarer Nähe. Zwischen den einzelnen Verhaltensregeln bestehen (fast) keine Abhängigkeiten, eine wichtige Eigenschaft um die Modularität zu gewährleisten. Es ist jedoch nicht immer möglich, völlige Unabhängigkeit zu erreichen.

Dieser Vorteil ist jedoch zugleich auch einer der Nachteile dieses Systems, denn ohne Abhängigkeiten wird es sehr schwer, komplexere Verhaltensmuster, welche aus einer Abfolge von mehreren Aktionen bestehen können, zu realisieren. Dazu jedoch später mehr.

Exklusive Verhaltensregeln unterliegen einer weiteren Einschränkung, so dass nur eine einzige von ihnen zur gleichen Zeit aktiv sein kann. Meistens handelt es sich dabei um Aktionen, die mit Animationen verbunden sind, wie das „Bewegen von A nach B“ oder „Angreifen“, welche es nicht zulassen, dass parallel andere exklusive Aktionen/Animationen ausgeführt werden. Sind mehrere exklusive Verhaltensregeln ausführbar, muss anhand ihrer Priorität entschieden werden, welches Verhalten den Vorzug erhalten soll. Im einfachsten Falle entscheidet die Reihenfolge der Verhaltensregeln in der Liste des Regelma-

nagers über die Prioritäten.

Im Folgenden werden der Aufbau und die Methoden einer Verhaltensregel vorgestellt. Eine Verhaltensregel besteht aus vier Methoden: *Runnable*, *Update*, *Exit* und *Enter*. In der *Runnable-Methode* wird überprüft, ob alle Bedingungen erfüllt worden sind, damit diese Regel angewendet werden soll. Anders als bei beispielsweise der PSO, bei der jede Regel immer angewendet wird, gibt es im Bereich der Spiele-KI viele Regeln, die nur in einigen Fällen, unter besonderen Bedingungen ausgeführt werden sollen. Falls eine Regel ausführbar ist, wird anschließend ihre *Update-Methode* aufgerufen. Unterscheidt sich die zuletzt ausgeführte Regel von der aktuellen, wird zunächst die *Exit-Methode* der alten Regel und anschließend die *Enter-Methode* der neuen Regel ausgeführt. In diesen Methoden können Initialisierungen und Aufräumarbeiten vorgenommen werden.

4.3.1. Modifiziertes Regelmanager-System

Wie schon vorher erwähnt, ist das gerade vorgestellte System mit dem Aufbau von Partikelschwärmen durchaus vergleichbar. Bei dem natürlichen Beispiel zur Partikelschwarmoptimierung wurde gezeigt, dass es bei der SI darauf ankommt, dass alle Individuen des Schwarms einen Satz von Regeln befolgen. Damit das Ganze aber nicht zu einem gewöhnlichen, regelbasierten System wird, müssen die Regeln, oder einige von ihnen, Informationen über ihre Nachbarn verarbeiten oder weitergeben.

Das in dieser Arbeit verwendete, modifizierte Regelmanager-System sieht ein paar Änderungen vor. Zum einen müssen die in Kapitel 4.3 als *Verhaltensregeln* beschriebenen Regeln nicht immer Verhaltensregeln sein, in dem Sinne, dass sie ein Verhalten hervorrufen. Sie können auch andere Aufgaben übernehmen, wie die Änderung von internen Zuständen eines NSCs. Im folgenden wird von *Regeln* und *Aktionsregeln* gesprochen. Diese ersetzen die Verhaltensregeln (jetzt: Regeln) und die exklusiven Verhaltensregeln (jetzt: Aktionsregeln).

Die Regeln spielen in dieser Arbeit eine zentrale Rolle. Denn Ziel dieser Arbeit ist es, nicht nur ein paar funktionierende Regelsätze für den Einsatz von SI als A-Life-Simulation zu geben, sondern auch ein Werkzeug zu entwickeln, mit dessen Hilfe neue Regeln einfacher erschaffen und bestehende verbessert werden können. Im Rahmen der Diplomarbeit wurden zwei verschiedene Herangehensweisen erarbeitet um eine Regelverbesserung zu erreichen. Wie dies im Detail geschieht wird im Folgenden erklärt werden.

4.3.2. Die Regeln

Zunächst erfolgt eine Unterteilung der verschiedenen Regeln nach zwei Eigenschaften. Zum einen, ob es sich um Regeln handelt, die eine physische Handlung zur Folge haben oder nicht und zum anderen, ob die Regeln nur einen einzelnen NSC betreffen oder mehrere. Anschließend wird der Aufbau der Regeln erläutert, gefolgt von den Verfahren, die dazu entwickelt wurden, diese Regeln halb-automatisch zu verbessern.

Normale Regeln und Aktionsregeln: Normale Regeln sind nie mit physischen Aktionen des NSCs verbunden und werden immer vor den Aktionsregeln ausgeführt. Zu diesen Regeln zählt zum Beispiel das Aufkommen von Hunger oder Müdigkeit. Aktionsregeln führen hingegen immer zu physischen Aktionen und unterliegen den Einschränkungen,

dass sie nur als ausführbar gelten, wenn der entsprechende NSC nicht schläft und noch keine andere Aktionsregel aktiviert wurde. Die Reihenfolge, in der die Regeln in die Liste des Regelmanager eingetragen wurden, legt auch gleichzeitig die Priorität der Regeln fest.

Individuelle und kollektive Regeln: Alle Regeln können in *individuelle* und *kollektive Regeln*, je nach verwendeten Informationen, unterteilt werden. Kollektive Regeln benutzen Informationen, die sie über anderen NSCs gesammelt haben um Aktionen auszuführen, die sie selbst oder andere NSCs betreffen. Individuelle Regeln, benutzten hingegen nur Informationen über den eigenen NSC und die Effekte der Regeln betreffen keine anderen NSCs.

	Individuell	*Kollektiv*
Aktionsregel	Hunger/Durst stillen, Arbeiten	Fluch vor Feinden, Jagen, Schwarmbewegungen
Regel	Abnahme von Sättigung	Austausch von Informationen unter NSCs (zum Beispiel Gefahrenzonen, Parametereinstellungen[1])

Tabelle 4.1.: Beispiele für kollektive und individuelle Regeln

Für das gewünschte, emergente Verhalten sind maßgeblich die kollektive Regeln von Interesse. Jedoch kann auf die individuellen Regeln nicht verzichtet werden, wenn man das Ziel einer glaubwürdigen Simulation von Lebewesen verfolgt.

Aufbau: Wie bei der ursprünglichen Form besitzen die Regeln ebenfalls *Vorbedingungen* die erfüllt werden müssen, damit sie angewendet werden. Allerdings werden die Regeln jetzt ebenfalls eine Liste von *Effekten* besitzen, in denen die Aktionen definiert sind. So ist es möglich, dass eine Regel gleichzeitig mehrere Effekte auslösen kann. Ein Beispiel wäre das schleichende Jagen eines Raubtieres. Diese Eigenschaft macht die Regeln modularer, als es in der originalen Version der Fall war.

Die Regeln selber sind recht allgemein gehalten, so muss nicht für jede NSC Gruppe eine eigene Regel aufgestellt werden. Damit sich die Verhaltensweisen trotzdem von NSC-Gruppe zu NSC-Gruppe oder auch innerhalb einer NSC-Gruppe unterscheiden können, besitzen viele Regeln, bzw. ihre Effekte und Vorbedingungen, *Parametereinstellungen*, um das Verhalten an den NSC und seine Umgebung anzupassen. *Parametereinstellungen* stellen Variablen dar, die Vorbedingungen und Effekte in der Stärke ihrer Ausprägung beeinflussen können. Beispielsweise kann über einen Parameter gesteuert werden, ab welcher Distanz die Vorbedingung *Kreatur gesichtet* als erfüllt gelten soll. So kann auf weit entfernte feindliche NSCs anders reagiert werden als auf sehr nahe.

Im Anschluss folgt eine Definition der Regeln, Vorbedingungen, Effekte und Aktionen.

Definition 8 *Eine Regel r ist ein Tupel (P, E_{Enter}, E_{Exit}, E_{Update}). Dabei stellt P die Menge an Vorbedingungen $p_1, ..., p_n$ dar, die erfüllt werden müssen, damit die Regel r ausführbar ist. Die Mengen $E_{Enter} = \{a_1, ..., a_m\} \subseteq \Phi$, $E_{Exit} = \{b_1, ..., b_i\} \subseteq \Phi$ und*

$E_{Update} = \{c_1, ..., c_k\} \subseteq \Phi$ stellen Teilmengen von Φ, der Menge aller möglichen Effekte dar, welche bei den jeweiligen Ereignissen Enter, Exit und Update ausgeführt werden sollen.

Das Ereignis Enter der Regel r tritt zum Zeitpunkt t auf, falls zum Zeitpunkt t-1 mindestens eine Vorbedingung $p \in P$ von r nicht erfüllt worden ist und zum Zeitpunkt t alle Vorbedingungen $p \in P$ erfüllt sind.

Das Ereignis Exit der Regel r tritt zum Zeitpunkt t auf, falls zum Zeitpunkt t-1 alle Vorbedingungen $p \in P$ der Regel r erfüllt worden sind und zum Zeitpunkt t mindestens eine Vorbedingung $p \in P$ nicht erfüllt ist.

Ein Ereignis Update der Regel r tritt zum Zeitpunkt t auf, falls zum Zeitpunkt t alle Vorbedingungen $p \in P$ erfüllt sind. Bei den Aktionsregeln gilt als zusätzliche Bedingung, dass zum Zeitpunkt t noch keine andere Aktionsregel a aufgeführt worden sein darf.

Eine Vorbedingung p kann den Wert *wahr* oder *falsch* annehmen. Welchen Wert p zum Zeitpunkt t angenommen hat, hängt ab von den zu untersuchenden Zuständen des NSC und/oder seiner Umwelt zum Zeitpunkt t. Ein Beispiel für eine Vorbedingung, ist die Vorbedingung *Ausgeruht*. Hier wird überprüft ob ein NSC unter dem Effekt *Erschöpfung*[2] leidet oder nicht.

Ein Effekt e stellt eine Sammlung von Aktionen und Zustandsveränderungen dar, die den NSC und/oder seine Umwelt betreffen können. Meistens handelt es sich dabei um Aktionen, die ein NSC ausführt. Ein Beispiel für einen Effekt ist das Aktivieren des „Schleichenmodus“, welcher in der „Jagen“ Regel der Raubtiere verwendet wird. Dieser Effekt sorgt für eine geringere Bewegungsgeschwindigkeit des NSCs, verkleinert aber dafür die Chance, von einem anderen NSC entdeckt zu werden.

Eine Aktion a ist eine simulierte, physische Tätigkeit eines NSCs. Eine Aktion a heißt atomar, wenn sie von dem zugehörigen Agenten direkt zum Zeitpunkt t ausgeführt werden kann, ohne dass andere Aktionen a' vorher ausgeführt werden müssen. Ein NSC kann zum Zeitpunkt t immer nur eine einzige Aktion ausführen. Ein Beispiel für eine atomare Aktion wäre das „Bewegen von Punkt A nach B“. Dies kann direkt von dem NSC durchgeführt werden. Eine nicht atomare Aktion ist das „Angreifen“ oder „Aufheben von Item-Objekten“. Um diese Aktion auszuführen, muss der NSC erst bis auf eine gewisse Entfernung an das Objekt herantreten, um diese Aktion auszuführen. Es wird also zuerst die atomare Aktion „Bewegen“ ausgeführt.

In Kapitel 5.3.1 werden konkrete Beispiele der Regeln mit ihren Vorbedingungen, Effekten und Parametern vorgestellt.

Datenaufnahme

Damit eine Verbesserung der Regeln erfolgen kann, müssen Informationen über diese und über das Verhalten der NSCs gesammelt werden, welche die Regeln verwenden.

Die *Datenaufnahme* sammelt Informationen über das Verhalten eines NSCs innerhalb eines bestimmten Zeitraumes, dem *Lernintervall*. Ein *Lernintervall* ist der Zeitraum, in dem die aktuellen Parametereinstellungen der Effekte, sowie die Vorbedingungen und Regeln getestet werden. So kann überprüft werden, welche Auswirkungen die veränderten

[2] Effekte werden in Kapitel 5.2 beschrieben.

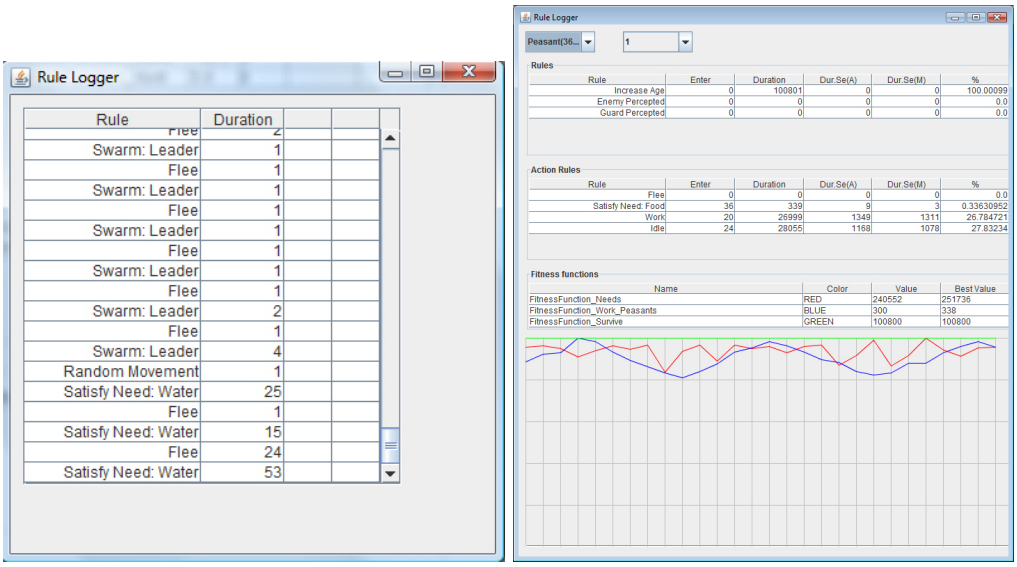


Abbildung 4.2.: Beispiele für einen RuleLogger (links) und RuleLogfiles (rechts).

Parametereinstellungen und Regeln auf die Aktivitäten des NSCs haben. Sofern nicht anders angekündigt, beträgt der Zeitraum eine simulierte Woche.

Jeder NSC besitzt für die *Datenaufnahme* einen eigenen *Logger* (Abbildung 4.2, links). Jeder Logger besitzt eine Liste, in der in chronologischer Reihenfolge die ausgeführten Aktionsregeln zuzüglich ihrer Ausführungsdauer festgehalten werden. Des Weiteren besitzt der Logger eine Liste von *RuleLogfiles* (Abbildung 4.2, rechts), sortiert nach den *Lernintervallen*. Für jedes Intervall wird für jede Regel eine *RuleLogfile* angelegt. In den *RuleLogfiles* werden nur Informationen über eine bestimmte Regel angelegt so wie folgende Daten:

- Anzahl der Regelaktivierungen
- Anzahl der Updates gemessen über das Intervall
- Durchschnittliche Dauer einer Aktivierung (Arithmetisches Mittel)
- Durchschnittliche Dauer einer Aktivierung (Median)
- Maximale Dauer einer Aktivierung
- Minimale Dauer einer Aktivierung
- Prozentualer Zeitanteil über das Intervall

Mit Hilfe dieser Informationen sollen Probleme und nicht gewünschte Effekte aufgedeckt werden, damit der Benutzer die betroffenen Regeln anpassen kann. Zu den negativen Effekten zählen:

- *Niemals aktivierte Regeln erkennen:* Durch einfaches Überprüfen der Anzahl der Aktivierungen können Regeln, die nie aktiviert wurden, schnell erkannt werden.

Je nach Situation kann der Benutzter sich dann die Vorbedingungen noch einmal näher ansehen und ggf. Änderungen vornehmen.

- *Auffällige Aktivierungsdauer erkennen:* Durch das Betrachten der Aktivierungsdauer einer Regel kann festgestellt werden, ob sich die Regel auffällig verhält oder nicht. So können sehr hohe Maximalwerte oder der abweichende Durchschnittswerte von Median und arithmetischem Mittel Aufschluss über mögliche Probleme liefern.

Nachdem die Datenaufnahme erläutert wurde, soll erklärkt werden wie mit Hilfe dieser Informationen die Regeln verbessert werden sollen, mit dem Ziel eine glaubhaftere Simulation der ausgewählten NSC-Gruppen zu erreichen. Innerhalb dieser Arbeit wurden zwei Ansätze entwickelt, die im Folgenden erläutert werden.

Ansatz 1: Zwei-Ebenen Modell

Bei dem *Zwei-Ebenen Modell* wird die Regelverbesserung in zwei unterschiedliche Optimierungsaufgaben geteilt und getrennt betrachtet. Im Folgenden wird die Verbesserung von bestehenden Regeln bzw. der Parametereinstellungen als *Interne Verbesserung* bezeichnet werden. Ziel der *internen Verbesserung* ist es, die jeweilige Fitnessfunktion der NSCs zu optimieren. Das *Zwei-Ebenen Modell* benötigt eine einzige Fitnessfunktion, beispielsweise die Ausführungszeit der sog. *primären Regel*. Jeder NSC kann eine *primäre Regel* besitzen, die das oberste Ziel des NSCs angibt und eine *Aktionsregel* sein muss. Alle anderen Aktionsregeln halten den NSC davon ab, diese primäre Regel auszuführen und dienen in der Regel der Erhaltung des NSCs („Essen“, „Schlafen“, „Fluchtverhalten“). Ein Beispiel für eine primäre Regel der NSC Gruppe Bauer wäre das Arbeiten auf dem Feld.

Die zweite *externe Verbesserung* konzentriert sich darauf, das Gesamtverhalten aller NSC-Gruppen, also der gesamten Simulation, zu optimieren. Ziel ist es, die Regeln der NSCs so anzupassen, dass ihr Verhalten als realistisch angesehen werden kann. Dabei ist die zu optimierende Funktion nicht ohne weiteres automatisch zu berechnen, da die Bewertung, ob ein Verhalten realistisch ist oder nicht, die subjektive Meinung des Betrachters darstellt.

Ebene 1 - Interne Verbesserung: In der Ebene 1 geht es darum, bestehende Regeln mit Hilfe der richtigen Parametereinstellungen zu verbessern. Das zu optimierende Kriterium ist der *Fitnesswert* des einzelnen NSCs. Dieser Fitnesswert wird bestimmt durch die Fitnessfunktion, die für jede einzelne NSC-Gruppe im Vorfeld festgelegt werden muss. Die Wahl der Fitnessfunktion beeinflusst das Verhalten der NSCs und hat später einen nicht unerheblichen Einfluss auf die Bewertung der gesamten Simulation. Die Idee dahinter ist, dass, wenn ein NSC gute Fitnesswerte erzielt, bzw. seine Fitness im Laufe der Zeit verbessert, sein Verhalten als intelligent angesehen werden kann. Die Optimierung gewährleistet auch eine Anpassung der Regeln bei sich ändernden Bedingungen. So könnte zum Beispiel ein Ergebnis der Anpassung sein, dass die Bauern im Winter früher mit der Arbeit aufhören, da es früher dunkel wird.

Innerhalb dieser Arbeit werden für die NSC-Gruppen Fitnessfunktionen benutzt, welche folgende Kriterien beinhalten:

- *Bauern*: Eigenes Überleben, Menge der in der Stadthalle abgegebenen Feldfrüchte, Erfüllung der Bedürfnisse
- *Wachen*: Eigenes Überleben, Überleben anderer Wachen und der Bauern, Erfüllung der Bedürfnisse
- *Goblins*: Eigenes Überleben, Anzahl an gefangenen Bauern, Erfüllung der Bedürfnisse
- *Beutetiere*: Eigenes Überleben, Erfüllung der Bedürfnisse
- *Raubtiere*: Eigenes Überleben, Anzahl gejagter Beutetiere, Erfüllung der Bedürfnisse

Die Suche nach den richtigen Fitnessfunktionen ist ein langwieriger und entscheidender Prozess. Die oben aufgeführten Fitnessfunktionen der einzelnen NSC Gruppen dienen vorrangig zu Testzwecken und sind wahrscheinlich nicht gut dafür geeignet, um eine wirklich glaubhafte Simulation herbeizuführen. Mit diesen Beispielen soll lediglich das Prinzip der Fitnessfunktionen erklärt werden. In Kapitel 4.3.2 werden die Schwierigkeiten, die bei der Erstellung der Fitnessfunktionen auftreten, anhand der Fitnessfunktion für das *Überleben-Verhaltensmuster* dargestellt.

Im Folgenden wird das Optimierungsverfahren erläutert, welches für die interne Verbesserung benutzt werden wird.

Lernverfahren: Für jede Kreatur wird ein *(1+1)-ES* mit selbst adaptiver Schrittweite verwendet, um die Parametereinstellungen der Regeleffekte und Vorbedingungen zu optimieren. Ausgehend von einem Startwert des Parameters, wird dieser einer Mutation unterworfen und anschließend über ein Lernintervall hinweg getestet. Zusätzlich kommt es von Zeit zu Zeit zu einer Übernahme der Parametereinstellungen, bzw. Teilen davon, durch andere NSCs nach dem Paradigma der Partikelschwarmoptimierung. Mit einer gewissen Wahrscheinlichkeit tauschen NSCs einer Gruppe bei ihrem Aufeinandertreffen ihre Parametereinstellungen aus. Dabei werden die Parametereinstellungen des schlechteren NSCs denen des besseren angeglichen.

Ebene 2 - Lernen von Regeln (Externe Verbesserung): Nachdem die Daten über die Regeln gesammelt wurden, müssen diese ausgewertet werden. Dieses Problem stellt sich jedoch als außerordentlich schwierig heraus, denn die Bewertung der Qualität der Regeln hinsichtlich ihrer Sinnhaftigkeit ist vom Betrachter abhängig und erfolgt auf subjektiver Basis. Es wird nicht selten der Fall eintreten, dass sich zwei menschliche Betrachter nicht einig werden, ob es sich bei dem gezeigten Verhalten nun um ein sinnvolles oder nicht sinnvolles handelt. Wie also soll dann ein Programm diese Entscheidung treffen können? Es gibt zwei unterschiedliche Ansätze, wie dieses Problem gelöst werden kann: Die erste Variante basiert auf beobachtetem Lernen, die zweite Variante basiert darauf, nicht sinnvolles Verhalten zu erkennen.

Beobachtetes Lernen: Kann die Berechnung des Realismus nicht automatisiert werden, muss die Beurteilung des Verhaltens durch einen menschlichen Betrachter erfolgen. Hierbei ist das größte Problem die benötigte Zeit. Ein Beobachter müsste jeden Durchlauf ansehen und anschließend bewerten, ob sich die NSCs gut oder schlecht verhalten haben. Zusätzlich könnte angegeben werden, wie gut oder wie schlecht das Verhalten war. Je nachdem wie weit sich das gezeigte Verhalten vom gewünschten abweicht, können die Schrittweiten der EA angepasst werden. Bei guten Verhaltensweisen können sie reduziert oder sogar auf Null gesetzt werden, bei schlechten Verhaltensweisen können größere Änderungen vorgenommen werden. Es können auch Änderungen an den Regeln bzw. den Vorbedingungen und Effekten vorgenommen werden.

Nicht sinnvolles Verhalten erkennen: Die *zweite Variante* ist es, nicht zu bewerten, ob das Gesamtverhalten sinnvoll war, sondern zu versuchen nicht sinnvolles Verhalten zu eliminieren. Da es viel zu lange dauern würde, wenn sich ein Betrachter die gesamte Simulation anschauen müsste, da ja ganze Tage oder gar Wochen simuliert werden, muss die Beurteilung anhand einer Art Kennzahl stattfinden. Dabei beschränkt sich die Beurteilung der Verhaltensmuster darauf, ob ein Verhalten schlecht war oder nicht. Denn diese Aufgabe ist, im Gegensatz zu der Bewertung der Sinnhalftigkeit eines Verhaltens, zumindest teilweise realisierbar. Merkmale von schlechten Verhaltensweisen sind zum Beispiel oszillierende Aktionen mit kurzer Aktivierungsdauer oder schlechte Fitnesswerte (Abbildung 4.3). Der Simulator hat nun die Aufgabe, wichtige Informationen von den nicht wichtigen zu trennen und dem Benutzter zu präsentieren.

Rule	Duration
Flee	2
Swarm: Leader	1
Flee	1
Swarm: Leader	1
Flee	1
Swarm: Leader	1
Flee	1
Swarm: Leader	1
Flee	1
Swarm: Leader	2
Flee	1
Swarm: Leader	4
Random Movement	1
Satisfy Need: Water	25

Abbildung 4.3.: Ein Beispiel für oszillierende Aktionsregeln

Häufige Aktionssequenzen sollen dem Benutzer angezeigt werden, damit er anschließend entscheiden kann, ob es sich um ein schlechtes Verhalten handelt oder nicht. Die Beurteilung des Benutzers ist erforderlich, da nicht jede häufig auftretende Sequenz von Aktionen unerwünscht ist. So läuft der Prozess des *Erholens* doch stets gleich ab: Zunächst muss ein NSC einen geeigneten Ort aufsuchen, an dem er rasten kann um sich anschließend schlafen zu legen. Andere periodische Verhaltensweisen sind jedoch uner-

wünscht, wie das ständige Wechseln von Flucht- und anderen Verhaltensweisen. Ist ein beobachtetes Verhalten sinnvoll, kann möglicherweise die interne Verbesserung dieser Regel deaktiviert oder die Schrittweite der Mutation begrenzt werden. Im anderen Fall können die Parametereinstellungen verändert oder Vorbedingungen und/oder Effekte der Regeln hinzugefügt oder entfernt werden. Das Modifizieren der Vorbedingungen und Effekte hat sich jedoch als nicht praktikabel herausgestellt, da die Anzahl der sinnvollen Veränderungen sehr, sehr klein ist im Gegensatz zu allen Kombinationen an Vorbedingungen und Effekten. Eine Suche nach optimalen Vorbedingungen und Effekten würde viele hunderte, wenn nicht tausende, Iterationen benötigen. Da am Ende von jedem Durchgang eine Bewertung des Benutzers erfolgen muss, ist diese Methode nicht brauchbar, auch wenn die Anzahl der Beobachtungen sich im Vergleich zu der ersten Methode drastisch verkleinert hat.

Ein großes Problem bei der Beurteilung des Verhaltens anhand von wenigen Kennzahlen sind jedoch Verhaltensmuster, die nicht durch das Erkennen von häufigen Episoden gefunden werden konnten. Diese werden gänzlich außer Acht gelassen, es sei denn die Simulation wird von dem Beobachter durchgängig verfolgt.

Zusätzlich zum Erkennen von häufigen Aktionssequenzen ist eine Beobachtung der Fitnesswerte aus der ersten Optimierungsebene denkbar. Aufgrund der Zufallskomponente in der Simulation kann jedoch nicht genau bestimmt werden, was die Ursache für den schlechten Fitnesswert war, ohne die Ereignisse beobachtet zu haben. Eine mögliche Ursache wären zum Beispiel schlechte Parametereinstellungen oder aber auch die adauernde Verfolgung durch einen feindlichen NSC. Außerdem besteht weiterhin das in Kapitel 4.1.1 angesprochene Problem der Normalisierung der einzelnen Fitnessfunktionsteile.

Diese Probleme wurden als Anlass genommen ein anderes Verfahren zu entwickeln um die Verbesserung der Regeln zu unterstützen. Dieser Ansatz wird im Folgenden erklärt werden und beschäftigt sich mit der separaten Behandlung von Fitnessfunktionen.

Ansatz 2: Separate Fitnessfunktionen

Die automatische Verbesserung der Verhaltensweisen deckt ein großes Problem auf: Die Schwierigkeiten bei der Erstellung einer *einzelnen* Fitnessfunktion für die Bewertung des Realismus, genauer gesagt das Problem der Normalisierung der einzelnen Funktionsteile.

Mit dem Ansatz der separaten Fitnessfunkionen kann dieses Problem umgangen werden, indem gar keine Normalisierung benötigt wird. Jeder NSC ist mit einer kleinen Menge von Verhaltensmustern ausgestattet. Jedem Verhaltensmuster können Regeln und Aktionsregeln zugeordnet werden, die dieses Verhalten hervorrufen sollen. Ein Vorteil dieser Methode ist es, dass alle Regeln, die einen direkten Einfluss auf ein Verhalten haben, schnell identifiziert und bearbeitet werden können, falls ein Anlass dazu besteht. Jedoch können sich die einzelnen Verhaltensmuster auch gegenseitig beeinflussen. Besitzt ein NSC zum Beispiel das Verhaltensmuster *Arbeiten* und *Hunger stillen* kann es dazu kommen, dass die Regeln für den Hunger dazu führen, dass der NSC die ganze Zeit an einem Ort verweilt, an dem es etwas zu essen gibt und nicht von dort weggeht. Zweifellos ist dies die beste Methode immer gesättigt zu sein und der NSC erlangt für dieses Verhaltensmuster einen sehr hohen Wert. Der Nebeneffekt wäre jedoch, dass das andere Verhaltensmuster Arbeiten sehr schlechte Werte erhalten würde, da der NSC nicht mehr auf sein Feld geht um dort zu arbeiten.

Um diesem Problem entgegen zu wirken, können die Fitnessfunktionen einzeln optimiert werden, unter Beobachtung der anderen Fitnessfunktionen. Kommt es zu einer deutlichen Verschlechterung der Fitnesswerte von Funktionen, deren Regeln nicht optimiert werden, kann schnell identifiziert werden, welche Parametereinstellungen für die Verschlechterung der anderen Fitnessfunktionen verantwortlich ist.

Beispiel: Ein Beispiel soll verdeutlichen wie ein Optimierungsprozess ablaufen könnte. Eine neue NSC-Gruppe soll drei Fitnessfunktionen erhalten: *Bedürfnisse erfüllen*, *Feldarbeiten* und *Überleben*. Es wird davon ausgegangen, dass diese Regeln und deren Vorbedingungen und Effekte schon vorhanden sind und es nur darum geht, ihre Parameter zu optimieren. Der erste Schritt besteht darin, die Startwerte und den sinnvollen Suchraum der Parametereinstellungen zu initialisieren. Dies sollte mit Hilfe des Vorwissens des Benutzers geschehen, da a priori-Wissen die benötigte Zeit der Optimierung signifikant verkleinern kann.

Um auszuschließen, dass unglücklich gewählte Startwerte der Parametereinstellungen anderer, nicht beteiligter Regeln die Optimierung der einzelnen Fitnessfunktion stark beeinflussen, sollten mehrere, verschiedene Startwerte getestet werden, ohne dass eine Optimierung stattfindet. Ergeben sich keine nennenswerten Änderungen, kann die nächste Stufe der Optimierung beginnen.

Als nächstes werden einige Testdurchläufe gestartet, in denen jede Fitnessfunktion einzeln optimiert wird. Während die Optimierung einer Fitnessfunktion durchgeführt wird, kann beobachtet werden, ob und wie stark die veränderten Parametereinstellungen die anderen Fitnessfunktionen indirekt beeinflussen. Kommt es zu großen Veränderungen, können die veränderten Parameter angezeigt werden, die diese hervorgerufen haben. Je nachdem, ob es sich um eine positive oder negative Änderung handelt, können die Startwerte und/oder die Grenzen des Suchraumes entsprechend angepasst werden.

Im letzten Schritt werden alle Fitnessfunktionen mit den neuen Startwerten und Grenzen der Suchräume optimiert.

Um den Zufallsfaktor zu begrenzen, sollte die Simulation, wann immer es möglich ist, mit nur einer NSC-Gruppe gestartet werden. Fitnessfunktionen wie die *Feldarbeit* benötigen keine weitere NSC-Gruppe um zu funktionieren. Die Funktion *Überleben* hingegen kann nicht optimiert werden, wenn keine andere, feindliche NSC-Gruppe in der Simulation auftaucht.

Problem Fitnessfunktion: Der Erfolg dieser Optimierungsmethode hängt davon ab, wie gut die Fitnessfunktionen der einzelnen Verhaltensmuster die gewollten Verhaltensweisen fördern. Wenn dies nicht der Fall ist, werden nicht die gewünschten, sondern andere Verhaltensweisen trainiert. In erster Linie werden die Fitnessfunktionen mit Hilfe einer Analyse des gewünschten Verhaltensmuster erstellt. Dabei gilt es herauszufinden, welche Faktoren eine Rolle spielen und wie sie in die Fitnessfunktion eingehen sollen. Ob die so gefundene Fitnessfunktion wirklich die Verhaltensweisen fördert, die gewollt sind, kann nur durch das Beobachten der Simulation und/oder aller Fitnesswerte geschehen. Die Schwierigkeit, eine geeignete Fitnessfunktion zu erstellen, wird an Hand des Beispiels der Überleben-Funktion deutlich gemacht.

Beispielfunktion Überleben: Zunächst wird die Annahme getroffen, dass ein NSC nur durch physische Einwirkungen anderer NSCs sterben kann. Das Ableben durch Man-

gel an Nahrung, Wasser oder Schlaf wird bewusst außer Acht gelassen, da dies zu einer Überschneidung mit der Funktion *Bedürfnisse erfüllen* führen würde. Die beteiligte Vorbedingung ist das *Wahrnehmen eines Feindes* und die Effekte sind das *Fliehen* vor dem nächsten Feind und die *Aktivierung des Rennen-Modus*.

Als nächstes müssen die Bewertungskriterien erarbeitet werden. Als erstes und offensichtlichstes Kriterium kommt der Gesundheitszustand des NSCs zum tragen. Auf diese Weise wird ein Verhalten, welches es zulässt, dass der NSC verwundet wird, schlechter bewertet als ein Verhalten, welches dies verhindern kann. Allerdings sind die Abstufungen der Fitnessfunktion dann sehr gering, da in den meisten Fällen ein fliehender NSC nur verletzt wird, wenn er nicht mehr fliehen kann, weil er zum Beispiel in einer Ecke gefangen ist. Außerdem wird nicht unterschieden, ob ein NSC seinen Verfolgern nur knapp entkommt, oder immer ausreichend Sicherheitsabstand gehalten hat. Für das Ergebnis - *Überleben* - ist dies zwar nicht von Bedeutung, aber für den Grad an Realismus schon. Ziel dieser Funktion ist es nicht, primär für das Überleben zu sorgen, sondern viel mehr ein realistischen Überlebensinstinkt zu simulieren. Realistisch wäre es also, dass ein NSC zu seinen Feinden einen gewissen Abstand einhält und sie nicht zu nah an sich herankommen lässt. Der Abstand soll daher ebenfalls in die Funktion mit einfließen.

Problem: Um welchen Wert soll die Funktion verändert werden in der Zeit, in der kein Feind in Sichtweite gekommen ist? Vergibt man in dieser Zeit positive Werte, werden auf den ersten Blick Verhaltensweisen gefördert, die es vermeiden in gefährliche Situationen zu geraten. Jedoch ist dies gar nicht der Sinn dieser Fitnessfunktion. Der Sinn der Fitnessfunktion ist es, auf auftretende Gefahren richtig zu reagieren, denn es liegt häufig auch nicht in der Macht des einzelnen NSCs zu vermeiden, dass gefährliche Situationen auftreten, da dies von den Aktionen der anderen NSCs abhängt, die für ihn nicht vorhersehbar sind. Negative Werte können ebenfalls nicht vergeben werden, da dies wenig Sinn machen würde. Die verbleibende Option ist es, keine Veränderungen an dem Fitnesswert vorzunehmen.

Kommt ein Feind in Sichtweite, sollen die NSCs vor diesem fliehen. Wie oben erwähnt, soll der Abstand zum nächsten Feind in die Berechnung des Fitnesswertes einfließen. Es muss nun entschieden werden, in welcher Weise der Abstand in die Fitness eingehen soll. Erhält ein NSC positive Werte, proportional zu dem gehaltenen Abstand, so wird ein Verhalten gefördert, welches immer den maximal möglichen Abstand zu dem nächsten Feind hält. Dies bedeutet jedoch auch, dass es für den NSC als vorteilhaft erscheint, dauerhaft den Feind auf maximalem Abstand zu halten und ihn nicht außer Sichtweite zu lassen. Denn sobald der Feind den Wahrnehmungsbereich verlässt, bekommt der NSC keine weiteren Punkte für den Abstand.

Eine Lösung für dieses Problem ist es, die Zeit, in der ein NSC flieht, mit einzubeziehen. Man berechnet also die Fitness f mit $f = Entfernung/Fluchtzeit$. Dies fördert sowohl größere Entfernungen zum nächsten Feind als auch kürzere Fluchtzeiten. Damit ein NSC, welcher häufiger in Kontakt mit Feinden gerät, einen höheren Fitnesswert bekommt, als ein NSC der weniger Feindbegegnungen hat, muss dieser Wert über alle Fluchtaktivierungen gemittelt werden.

$$f = \frac{\text{Entfernung / Fluchtzeit}}{\text{Anzahl der Fluchtaktivierungen}} \tag{4.1}$$

5. Simulator und Editor

Damit die Regelsätze getestet werden können, bedarf es einer Testumgebung. Grundsätzlich könnte dafür ein Spiel benutzt werden, welches genug Möglichkeiten bietet, um das modifizierte Regelmanager-System zu implementieren, wie z.B. NEVERWINTER NIGHTS (NWN) . Jedoch haben Spiele als Testumgebung einen Nachteil. Die Geschwindigkeit der Simulation und damit das Testen der Regeln kann nur in Echtzeit ablaufen und nicht beschleunigt werden. Um diese Beschränkung zu umgehen, wurde im Rahmen dieser Arbeit ein Simulator entwickelt, welcher sehr stark an NWN angelehnt ist und die Möglichkeit bietet, die Spielgeschwindigkeit drastisch zu erhöhen. Je nach Anzahl der NSCs in der Simulation sind Zeitfaktoren von bis zu 175.000 facher Geschwindigkeit möglich[1]. Des Weiteren hat der Simulator den Vorteil, dass man nicht an die Eigenheiten eines bestimmten Spiels gebunden ist. Zum Beispiel gibt es im Falle von NWN viele Besonderheiten, die durch das Spiel bzw. die verwendete Programmiersprache NW-Script selbst zustande kommen, zu beachten.

Die Testumgebung besteht aus drei Komponenten:

- Editor
- Simulator
- Regeleditor

5.1. Editor

Das Hauptprogramm (Abbildung 5.1) bildet der Editor. Hier können neue Simulationsabläufe erstellt und gestartet werden. Für einen Simulationsablauf wird eine neue Umgebung (hier: Welt) erschaffen, die verschiedene Eigenschaften besitzt.

Feld: Create Simulation: Im Feld *Create Simulation* kann die Anzahl der NSCs der einzelnen NSC Gruppen eingestellt werden. Zu Beginn eines Lernintervalls werden so viele Kreaturen von den entsprechenden Typen erstellt, bis die eingetragene Anzahl erreicht ist. Dies soll sicherstellen, dass die gestorbenen Kreaturen ersetzt werden, damit die Simulation mit der gewünschten Anzahl an NSCs fortgesetzt werden kann. Die neu erstellten NSCs starten an speziellen, festgelegten Orten und nicht zufällig in der Umgebung.

Die Liste *Creature Types* zeigt alle zur Verfügung stehenden NSC-Gruppen an und bietet die Möglichkeit, die Regeln einer gesamten Gruppe vor dem Start der Simulation anzusehen und zu editieren.

[1] Verwendete Hardware: Intel Core 2 Duo T7100(2,1 GHz), 3 GB DDR2 RAM. Bei nur einem simulierten NSC. Allerdings nimmt die Geschwindigkeit im Laufe der Simulation stetig ab, was auf Optimierungsbedarf des Simulators hindeutet.

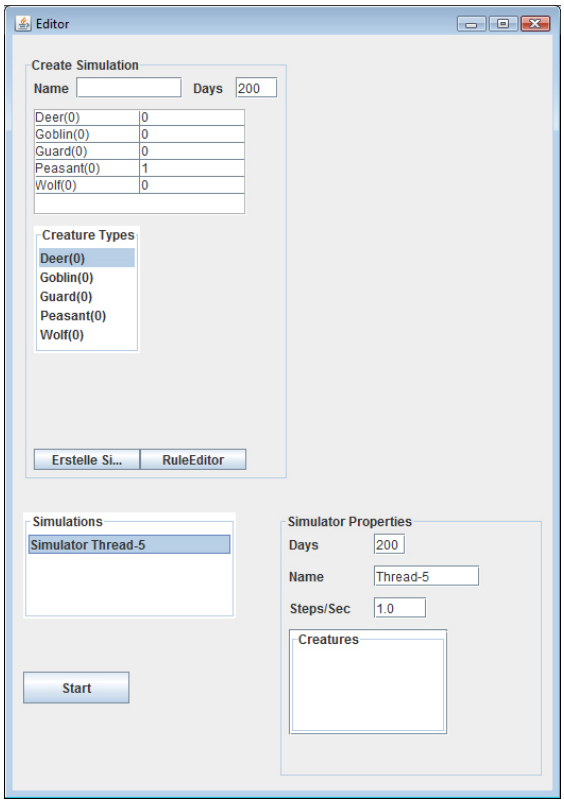


Abbildung 5.1.: Hauptprogramm: Editor

Ist der Simulator erstellt worden, wird er in der Liste *Simulations* aufgeführt und kann gestartet werden. Das Feld *Simulation Properties* zeigt noch einmal einige der Eigenschaften des Simulators an.

5.2. Simulator

Der Simulator bildet die Grundlage für das Testen der Regeln und Verhaltensmustern der NSCs. Er steuert die gesamte Simulation und übernimmt die Datenaufnahme.

Ist eine Simulation gestartet worden, erscheint dem Benutzter eine graphische Oberfläche, die dazu dient, den Verlauf der Simulation zu beobachten.

Auf der linken Seite befindet sich das Feld *Population*, welches eine Liste von Informationen über alle lebenden NSCs des Simulators anzeigt. Zu diesen Informationen gehören eine eindeutige ID, der Name des NSCs, die aktuelle Aktionsregel, die aktuelle Aktion und physische Eigenschaften wie Trefferpunkte, Rüstungsklasse und die Koordinaten des Standpunkts.

Unter der Liste von lebenden NSCs befindet sich eine Abbildung der Spielumgebung. Ein NSC wird durch einen farbigen Punkt[2] dargestellt. Dieser wird umgeben von seinem

[2] grün = Bauern, blau = Wachen, rot = Goblins, türkis = Beutetiere, schwarz = Raubtiere und magenta = der in der Populationsliste markierte NSC

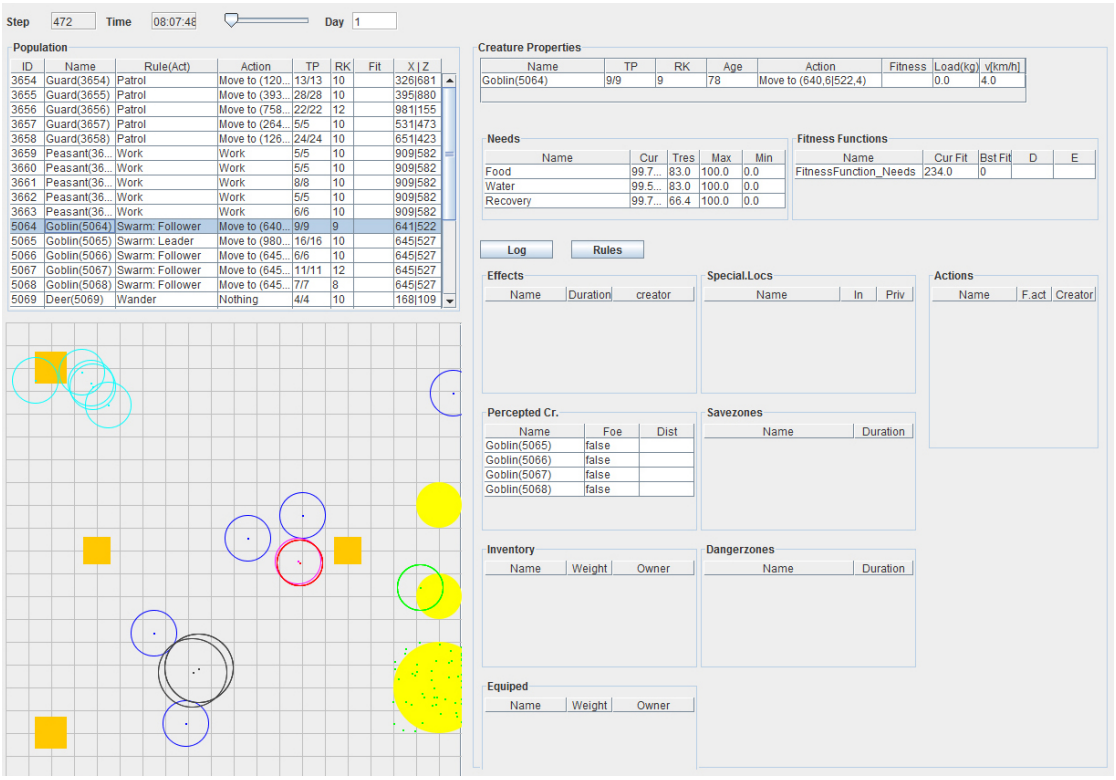


Abbildung 5.2.: Simulationsfenster

Wahrnehmungsbereich, welcher je nach NSC-Gruppe oder auch aktiver Regeln unterschiedlich ausfallen kann. Schleichende NSCs werden jedoch nicht automatisch entdeckt, sobald sie den Wahrnehmungsbereich eines anderen NSCs betreten, sondern nur mit einer gewissen Wahrscheinlichkeit, die mit abnehmender Distanz zunimmt.

Auf dem Boden liegende Gegenstände werden ebenfalls als Punkte dargestellt, jedoch ohne einen Wahrnehmungsbereich.

Besondere Orte wie „Weiden“ oder „Äcker“ werden als ausgefüllte Formen, meistens Kreise, in gelber Farbe angezeigt. Orangene Formen, meistens Quadrate, symbolisieren hingegen Gebäude.

5.2.1. Funktionen

Während einer Simulation kann die Geschwindigkeit, mit der sie ablaufen soll, verändert werden. Die maximal erreichte Geschwindigkeit hängt stark von der Anzahl der zu simulierenden Kreaturen und deren Regeln ab und reicht von etwa 500 facher Geschwindigkeit bei 50 NSCs[3] bis zu 175.000 facher Geschwindigkeit bei nur einem simulierten NSC. Des Weiteren können die Regeln der Kreaturen mit Hilfe des Regeleditors (s. Regeleditor) während der Simulation geändert werden.

[3] jeweils zehn NSCs von den Gruppen Bauern, Wachen, Goblins, Raub- und Beutetiere

5.2.2. Aufbau des Simulators

Jeder Simulator besitzt ein oder mehrere *World-Objekte*, in denen die eigentliche Simulation stattfindet. Der Simulator ist dafür ausgelegt mehrere Welten gleichzeitig zu simulieren, damit die Auswirkungen unterschiedlicher Regeln parallel beobachtet werden können.

Ein World-Objekt stellt die Spielwelt dar und besitzt Listen für die sich in dieser Welt befindenden *WorldObject-Objekte*. Alle physischen Objekte innerhalb der Spielwelt werden mit Hilfe der WorldObject-Objekte dargestellt. Sie besitzen grundlegende Eigenschaften, die jedes physische Objekt in der Welt haben muss, wie einen Ort oder die zugehörige Welt, in der sie sich befinden. Es gibt drei verschiedene Arten von WorldObject-Objekten: *Gebäude*, *Kreaturen* und *Gegenstände*.

Zusätzlich bietet das World-Objekt einige Einstellungsmöglichkeiten wie die Ausdehnung der (zwei-dimensionalen) Welt[4], die Stunde des Sonnenauf- und untergangs, so wie eine Liste von *besonderen Orten*. Ein besonderer Ort beschreibt eine Fläche innerhalb der Spielwelt, die besondere Eigenschaften aufweisen kann. Ein Beispiel hierfür sind die Felder der Bauern. Bei den Feldern handelt es sich um *öffentliche*, besondere Orte, die für alle Kreaturen sichtbar sind. Jedoch haben sie für die NSC-Gruppe der Bauern eine besondere Bedeutung, denn nur auf den Feldern können sie ihrer Feldarbeit nachgehen. Ein Beispiel für einen *privaten*, besonderen Ort sind Gefahrenzonen. Sie können von jedem NSC eingerichtet werden, sind aber nur für diesen NSC sichtbar und von Bedeutung. Alle anderen NSCs haben keine Kenntnis über private, besondere Orte, die sie nicht selbst eingerichtet haben.

Kreaturen, Gegenstände und Gebäude sind sehr komplexe Objekte mit vielen Methoden und Eigenschaften wie physische Merkmale und die Größe des Wahrnehmungsbereichs. Auf die meisten wird hier jedoch nicht näher eingegangen, da es sich um mehr als 150 verschiedene Eigenschaften, Methoden und/oder Komponenten handelt. Eine sehr wichtige Komponente der Kreaturen ist der *Regelmanager*, welcher in Kapitel 4.3 ausführlich beschrieben wurde.

Gebäude und besondere Orte können von Kreaturen betreten und verlassen werden und merken sich die Kreaturen, die sich momentan innerhalb des Gebäudes oder des Ortes aufhalten. Je nach Ort oder Gebäude können spezielle Funktionen aufgerufen werden, die alle Kreaturen betreffen, die sich gerade im Inneren aufhalten.

Komplexität des Simulators

Der Simulator unterstützt eine Reihe von Aspekten und Eigenschaften der realen Welt. Die Aspekte und ihre Eigenschaften beruhen jedoch nur auf persönlichen Annahmen und müssen nicht mit den realen Bedingungen übereinstimmen. Durch nicht korrekte Annahmen können sich die einzelnen Verhaltensmuster möglicherweise von den gewünschten unterscheiden. Es wurden nur Annahmen verwendet, da eine gründliche Analyse der Auswirkungen der Aspekte und Eigenschaften auf Lebewesen in der realen Welt den Rahmen dieser Arbeit überschritten hätte.

[4] Die Grenzen können nicht überschritten werden und stellen unpassierbare Hindernisse dar.

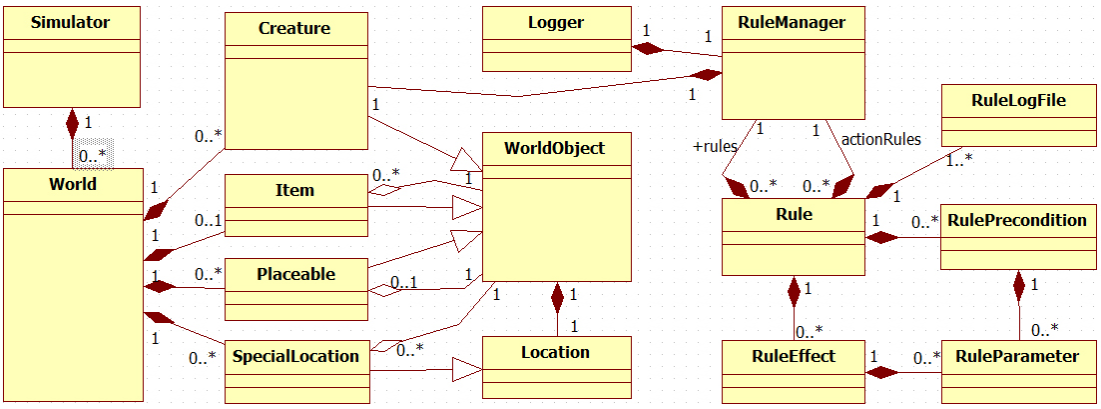


Abbildung 5.3.: Ein einfacher Überblick über die Klassen

Tag-/Nachtwechsel: Die Tageszeit hat einen Einfluss auf die Tätigkeiten der NSCs. Zum Beispiel arbeitet ein Bauer nachts nicht so effektiv auf seinem Feld wie bei Tageslicht. In diesem Simulator wird die Effektivität von Tätigkeiten, die am Tage erledigt werden sollen, um den Faktor fünf reduziert, falls sie nachts erledigt werden. Dies soll bewirken, dass die NSCs im Laufe der Simulation herausfinden, dass es besser wäre tags über zu arbeiten und nachts zu schlafen. In dem Kapitel 6.1.1 wird ein Beispiel dafür gegeben, was passiert, wenn der Tag- und Nachtwechsel nicht berücksichtigt wird.

Bedürfnisse: Ein Lebewesen, welches nicht isst, nicht trinkt oder nicht schläft, wirkt wenig realistisch. Die Bedürfnisse bestimmen im hohen Maß die Aktionen der NSCs und liefern oft die Motivation für viele Aktionen. Die meisten NSCs werden nicht um der Arbeit selbst willen viel Zeit auf ihren Feldern verbringen, sondern weil sie durch diese Aktion ihre Bedürfnisse erfüllen können. Die Arbeit auf dem Feld bringt den Bauern Geld ein, so dass sie sich etwas zu essen und zu trinken leisten können. Andere Tätigkeiten, wie der abendliche Besuch einer Taverne, sind auf den Wunsch nach Unterhaltung und nach sozialen Kontakten zurückzuführen. Ein Spiel muss daher geeignete Mechanismen bieten, um diese Bedürfnisse abzudecken. So erhalten viele Aktionen wie „Essen“, „Trinken“, „Schlafen“ und „Arbeiten“ eine realistische Motivation. Die unterstützten Bedürfnisse in diesem Simulator sind „Hunger“, „Durst“ und „Erschöpfung“.

Umwelt: Es ist sehr wichtig, dass die Umgebung, in der sich der Agent bewegen soll, mit dem Agenten zugänglichen Informationen angereichert wird. Ohne sie kann man sich die Umgebung als große, weiße Fläche vorstellen, in der Objekte nur als abstrakte Gegenstände enthalten sind. Ein menschlicher Spieler kann zwar aus der Farbe und Form der Landschaft darauf schließen, dass es sich um einen Acker oder eine Straße handelt und dementsprechend handeln, der Agent hat diese Informationen jedoch nicht. Die Umgebung muss also zwei unterschiedliche Arten von Informationen besitzen, einmal für die menschlichen Spieler und einmal für den Agenten. Ebenso verhält es sich mit Objekten der Spielwelt. Aufgrund unserer Erfahrung wissen wir, wie und wofür man einen Eimer benutzen kann, für den Agenten ist ein Gegenstand aber vorerst nur ein „Ding“ mit

dem Namen „Eimer“. Ohne weitere Informationen kann der Agent nichts mit diesem Gegenstand anfangen. Grundlegende Aspekte wie Benutzbarkeit, Gewicht, Wirkungsweise, Sinn und Zweck müssen ihm bekannt gemacht werden, damit sie mit dem Objekt interagieren können. Die Sims (2000) von Electronic Arts zeigen mit ihren *Smart Objects*, wie sich dieses Problem lösen lässt.

Terrain und besondere Orte: Mit dem *Terrain* ist die Beschaffenheit und die damit verbundenen Eigenschaften der (bearbeiteten) natürlichen Umgebung gemeint. So haben verschiedene Untergründe wie Gras, Matsch und Schlamm oder Fels Auswirkungen auf die Bewegungsgeschwindigkeit eines NSCs. Andere Eigenschaften wie die Höhe von Hügeln oder Bergen verändern die Sichtverhältnisse. Soll ein Verhalten, wie das Besteigen eines Hügels, für eine bessere Übersicht simuliert werden, so muss dieser Aspekt auch vom Spiel unterstützt werden.

Das Beispiel des sich verändernden Sichtfeldes durch den Hügel macht die Wichtigkeit eines anderen Aspektes des Terrains deutlich: die Bedeutung von *besonderen Orten*. Nicht jeder Punkt in der Spielwelt hat die gleiche Bedeutung. Für einen menschlichen Spieler ist es leicht, diese anhand von visuellen Informationen, seinem Gedächtnis und dem Kontext zu erkennen. So deuten beispielsweise herumliegende Knochen Gefahren an, während Gebäude uns mitteilen, dass wir uns innerhalb einer Siedlung oder Stadt befinden. Der Ort bestimmt ebenfalls maßgeblich die Aktionen eines NSCs. So sollte kein NSC anfangen, die Tiere innerhalb einer Stadt zu erlegen, wie er es in einem Wald machen würde, oder das nächst beste Bett in der feindlichen Festung zu benutzen, weil er gerade müde ist. Große, statische Objekte stellen auch Informationen zur Verfügung. So kann es sich um ein öffentliches oder privates Gebäude handeln, wie eine Taverne oder ein Wohnhaus. Jedes Gebäude besitzt auch einen speziellen Zweck, eine ebenfalls wichtige Information für die NSCs.

In diesem Simulator wurden folgende Aspekte implementiert:

- *Die Goblinhöhle*: Ein Gebäude, in dem die Goblins sich Essen und Wasser besorgen sowie sich ausruhen können.
- *Die Stadthalle*: Ein Gebäude analog zur Goblinhöhle, nur für die Bauern und Wachen.
- *Die Raubtierhöhle*: Ein Gebäude, in dem die Raubtiere übernachten können.
- *Die Beutetierhöhle*: Ein Gebäude analog zur Raubtierhöhle, nur für die Beutetiere.
- *Gras*: Ein Gebiet, in dem periodisch Gras-Gegenstände erscheinen, die von den Beutetieren als Nahrung genutzt werden können.
- *Felder*: Ein Gebiet, auf dem die Bauern arbeiten können.

Zwei weitere Eigenschaften ,die implementiert wurden, sind:

- *Auswirkungen der Aktion „Schlafen“*: Schläft ein NSC, so verringert sich in dieser Zeit die Zunahme seiner Bedürfnisse. Dies soll verhindern, dass ein NSC während des Schlafes Mangelerscheinungen aufgrund von Durst oder Hunger entwickelt.

- *Gewichtsbelastung*: Die Gewichtsbelasung eines NSCs durch mitgeführte Gegenstände beeinflusst seine Bewegungsgeschwindigkeit. Jeder NSC kann ein bestimmtes Gewicht ohne negative Folgen tragen. Wird diese Grenze überschritten, verringert sich die Geschwindigkeit des NSCs proportional zur Überlastung.

5.2.3. Ablauf der Simulation

Nachdem der Simulator gestartet wurde, führt er solange eine Schleife aus, bis die zu simulierenden Tage abgelaufen sind. Dabei entspricht jeder Simulatorschritt einer simulierten Sekunde. Alle sechs[5] Iterationen wird ein sog. *Heartbeat* erzeugt. Bei jedem Heartbeat werden alle Kreaturen, Gebäude und besonderen Orte zu einer Aktualisierung (ebenfalls Heartbeat genannt) aufgefordert. Aus Gründen der Leistung findet dieser nur alle sechs Sekunden statt. Ein höherer Wert verbessert zwar die Reaktionszeit der Objekte, verringert im Gegenzug dafür aber die Simulationsgeschwindigkeit. Unter NWN wurde die Annahme getroffen, dass sechs Sekunden als Reaktionszeit ausreichen, um auf Veränderungen der Umwelt zu reagieren.

Die Reihenfolge, in der die Objekte aufgerufen werden, hängt von ihrem Typ ab. Zuerst werden die besonderen Orte, dann die Gebäude und anschließend die Kreaturen aktualisiert. Die Kreaturen stehen an letzter Stelle, damit sie immer mit den neusten Informationen arbeiten können, die sie von den anderen Objekten erhalten. Die Reihenfolge der Elemente innerhalb der einzelnen Gruppen hängt von der Reihenfolge in den Listen des World-Objektes ab. Es gibt also keine getrennte Berechnungs- und anschließende Ausführungsphase.

Was in einem Heartbeat geschieht, hängt maßgeblich von dem Typ des Objektes ab. Im Folgenden wird geschildert, was beim Heartbeat einer Kreatur passiert.

Wird ein Heartbeat für eine Kreatur ausgelöst, werden mehrere Methoden der Reihe nach aktiviert. Zu diesen zählen:

- Effekte-Update
- Wahrnehmung-Update
- Regelmanager-Update
- Aktion ausführen
- Fitness berechnen

Zu allererst werden alle Effekte (wie z.B. der *Schlaf-Effekt*) auf die Kreatur angewendet. Dieser Schritt muss als erster erfolgen, da im Falle des Schlaf-Effekts alle anderen Methoden davon beeinflusst werden. Nachdem die Effekte angewendet wurden, wird die Wahrnehmung der Kreatur aktualisiert, damit dem Regelmanager im nächsten Schritt die neuesten Informationen vorliegen. Der Ablauf des Regelmanagers wurde in Kapitel 4.3 im Detail beschrieben. Dieser besitzt je eine Liste für Regeln und Aktionsregeln, welche der Reihe nach abgearbeitet werden. Zuerst werden alle Regeln behandelt, anschließend die Aktionsregeln. Nachdem der Regelmanager die nächste Aktion für den NSC bestimmt hat, wird diese ausgeführt. Anschließend werden die einzelnen Fitnesswerte aktualisiert.

[5] Die Zahl kann beliebig geändert werden.

Dieser Ablauf wiederholt sich bei jedem Heartbeat des World-Objekts für jede Kreatur. Die Heartbeat-Methoden anderer WordlObject-Objekte werden nicht weiter im Detail erläutert, laufen aber vom Prinzip her ähnlich ab.

Gebäude und besondere Orte besitzen zusätzliche Methoden, um auf verschiedene Ereignisse (beim Betreten und Verlassen des Gebäudes) zu reagieren.

Dieser Ablauf wird fortgesetzt, bis der Benutzer die Simulation manuell abbricht oder die zu simulierenden Tage abgelaufen sind. Im Anschluss wird eine Übersicht über den Verlauf der Simulation angezeigt.

5.2.4. Abschlussübersicht

Nachdem eine Simulation beendet wurde, werden die gesammelten Informationen über die Regeln angezeigt (Abbildung 5.4). Zunächst muss aus der Liste oben links ein NSC ausgewählt werden. Wenn dies geschehen ist, werden dem Benutzer alle Informationen über dessen Regeln, Aktionsregeln und Fitnessfunktionen angezeigt. Die Regeln und Aktionsregeln sind nach den verschiedenen Lernintervallen chronologisch sortiert. Welche Informationen gesammelt werden, wurde in Kapitel 4.3.2 vorgestellt.

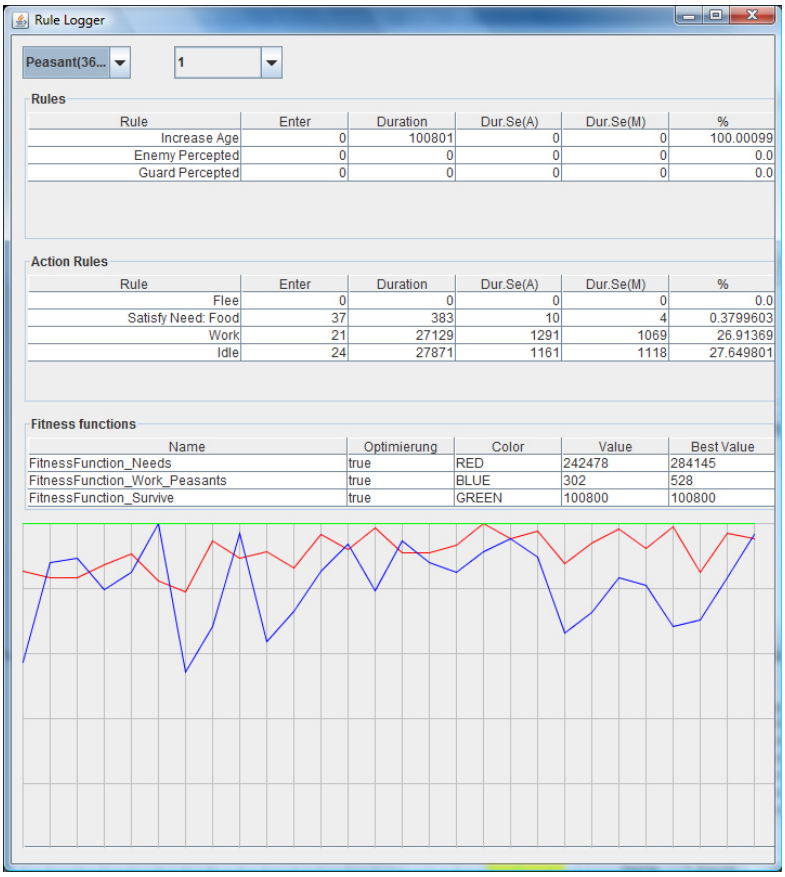


Abbildung 5.4.: Zeigt den Verlauf der Fitnessfunktionen und die Informationen der RuleLogFiles an.

Die Angaben der prozentualen Zeitanteile der einzelnen Regeln ergeben in der Summe nicht 100%. Dies liegt darin begründet, dass die Zeit, in der der NSC schläft, nicht in der Statistik aufgeführt wird. Demnach verbringt der NSC *100 - die Summe der prozentualen Zeitanteile aller Regeln %* damit zu schlafen.

Als nächstes folgt eine Liste der Fitnessfunktionen des NSCs. Dabei werden *Name*, *Optimierungszustand*, die *Farbe der Funktionskurve*, *der letzte* und *der beste Wert* angezeigt. Darunter befindet sich der Verlauf der Fitnessfunktionen über alle Lernintervalle.

Im Folgenden werden der Regeleditor, die verwendeten Regeln und deren Vorbedingungen und Effekte vorgestellt.

5.3. Regeleditor

Abbildung 5.5 zeigt den *Regeleditor*, der es dem Benutzer erlaubt, Regeln und Aktionsregeln von NSCs vor und während der Simulation zu erstellen, zu editieren oder auch zu löschen.

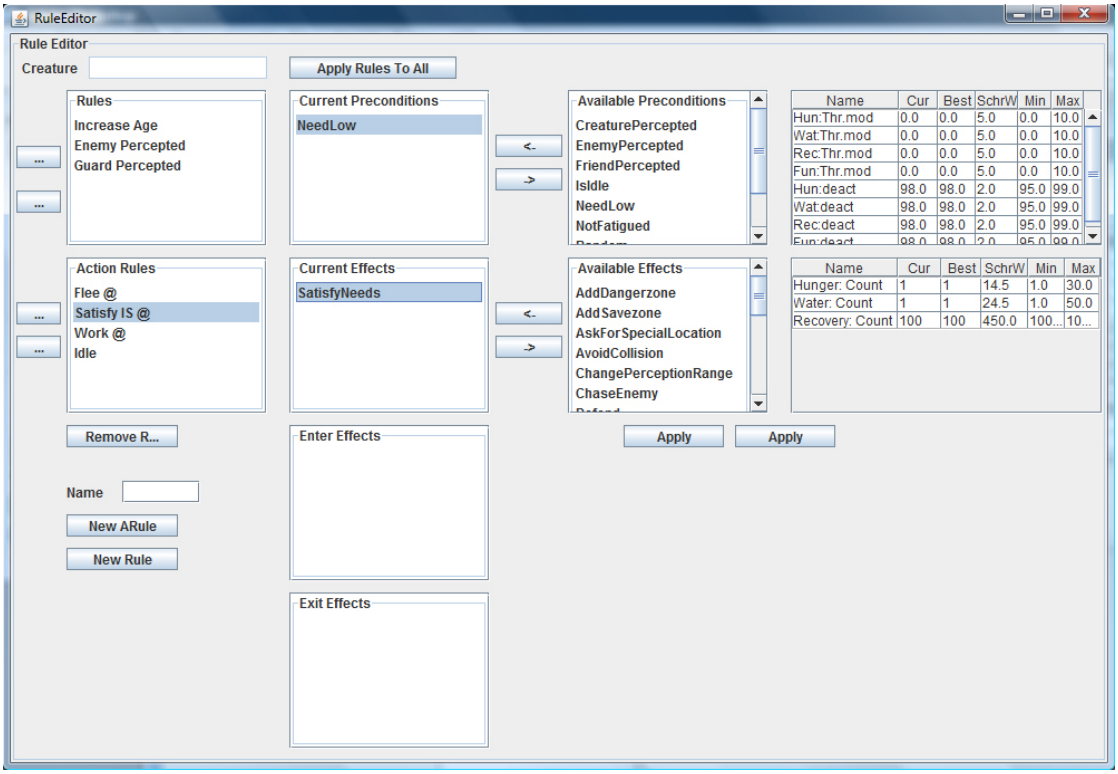


Abbildung 5.5.: Der Regeleditor

Um die Regeln eines bestimmten NSCs zu bearbeiten, muss er in dem Simulator ausgewählt werden. Nachdem ein NSC ausgewählt wurde, erscheint sein Name im Textfeld in der linken, oberen Ecke und alle Regeln und Aktionsregeln, die er momentan besitzt, in den entsprechenden Listen *Rules* und *Action Rules*. Links neben den beiden Listen befinden sich Schalter, mit denen man die Reihenfolge der Regeln verändern kann. Das

Symbol @ zeigt an, ob eine Regel einer Fitnessfunktion zugeordnet wurde oder nicht.

Wird eine Regel oder Aktionsregel ausgewählt, werden die *Vorbedingungen*, *Effekte*, *Enter-Effekte* und *Exit-Effekte* angezeigt. Diesen Listen können anschließend andere Vorbedingungen oder Effekte aus den Listen *Available Preconditions* und *Available Effects* hinzugefügt werden.

Bei der Auswahl einer Vorbedingung oder eines Effektes werden die *Parametereinstellungen* dieser Elemente in den Tabellen rechts angezeigt[6]. Falls es sich um primitive Datentypen handelt, können diese editiert werden. Falls die Änderungen für alle NSCs der gleichen Gruppe angewendet werden sollen, kann der Benutzer den Schalter *Apply Rules to All* benutzen.

5.3.1. Vorbedingungen, Effekte und Regeln

Zunächst folgt eine Beschreibung einiger Vorbedingungen und Effekte, die zur Verfügung stehen und anschließend werden einige der verwendeten Regeln samt ihrer Vorbedingungen und Effekte vorgestellt.

Vorbedingungen

Kreatur gesichtet: Diese Vorbedingung gilt als erfüllt, wenn eine Kreatur den festgelegten Wahrnehmungsbereich betritt.

Parameter:

- *Distanz [Float] Standardwert = 1.0*: legt fest, ab wieiel % der Wahrnehmungsreichweite auf eine Kreatur reagiert werden soll.
- *Type [Vector<String>] Standardwert = alle NSC-Gruppen*: ist eine Liste von NSC-Gruppennamen, auf die reagiert werden soll.
- *Alive [Int] Standardwert = 1(nur lebende)*: gibt an, ob nur auf lebende und/oder auch tote Kreaturen reagiert werden soll.

Feind gesichtet: Diese Vorbedingung gilt als erfüllt, wenn eine feindliche Kreatur den festgelegten Wahrnehmungsbereich betritt.

Parameter:

- *Distanz [Float] Standardwert = 1.0*: legt fest, ab wieiel % der Wahrnehmungsreichweite auf eine Kreatur reagiert werden soll.
- *Exceptions [Vector<String>] Standardwert = leerer Vektor*: ist eine Liste von NSC-Gruppennamen, auf die nicht reagiert werden soll.
- *Alive [Int] Standardwert = 1(nur lebende)*: gibt an, ob nur auf lebende und/oder auch tote Kreaturen reagiert werden soll.

[6] Parameter für Vorbedingungen oben und für Effekte unten

Freund gesichtet: Diese Vorbedingung gilt als erfüllt, wenn eine freundliche Kreatur den festgelegten Wahrnehmungsbereich betritt.
Parameter:

- *Distanz [Float] Standardwert = 1.0*: legt fest, ab wieiel % der Wahrnehmungsreichweite auf eine Kreatur reagiert werden soll.
- *Exceptions [Vector<String>] Standardwert = leerer Vektor*: ist eine Liste von NSC-Gruppennamen auf die nicht reagiert werden soll.
- *Alive [Int] Standardwert = 1(nur lebende)*: gibt an, ob nur auf lebende und/oder tote Kreaturen reagiert werden soll.

Untätig: Diese Vorbedingung gilt als erfüllt, wenn ein NSC noch keine Aktion ausführt.

Bedürfnisse niedrig: Diese Vorbedingung gilt als erfüllt, wenn ein Bedürfnis des NSCs den Grenzwert + Modifikator (s. Parameter) unterschritten hat.
Parameter:

- *Hun:Thr.mod [Float] Standardwert = 0.0*: modifiziert den Grenzwert für das Unterschreiten des Bedürfnisses „Hunger“, so dass die Vorbedingung schon früher auslöst.
- *Thi:Thr.mod [Float] Standardwert = 0.0*: ist analog zu Hun:Thr.mod (s.u.) für das Bedürfnis „Wasser“.
- *Rec:Thr.mod [Float] Standardwert = 0.0*: ist analog zu Hun:Thr.mod (s.u.) für das Bedürfnis „Erholung“.
- *Hun:deact [Float] Startwert = 98.0*: das Bedürfnis „Hunger“ wird solange erfüllt, bis es den hier angegebenen Wert wieder erreicht hat. Dies diente ursprünglich für den nicht linearen Anstieg des Sättigungsgefühls der NSCs bei der Nahrungsaufnahme. Dieser Parameter wird aktuell nicht verwendet, da die Nahrungsquellen den Sättigungswert immer um einen festen Wert erhöhen.
- *Thi:deact [Float] Startwert = 98.0*: ist analog zu Hun:deact, nur für das Durstgefühl.
- *Rec:deact [Float] Startwert = 98.0*: ist analog zu Hun:deact, nur für die Erschöpfung.

Nicht erschöpft: Diese Vorbedingung gilt als erfüllt, wenn der NSC nicht von dem Effekt *Erschöpfung* betroffen ist.

Zufall: Diese Vorbedingung gilt als erfüllt, wenn die Zufallszahl einen bestimmten Wert überschreitet.
Parameter:

- *Prob [Float] Standardwert = 0.1*: ist eine Zahl zwischen 0 und 1, welche die Wahrscheinlichkeit angibt, dass diese Vorbedingung als erfüllt gilt.

Zeit: Diese Vorbedingung gilt als erfüllt, wenn die aktuelle Tageszeit der Simulation innerhalb eines bestimmten Intervalls liegt.
Parameter:

- *Start Hour [Int] Standardwert = 0*: ist die untere Grenze des Intervalls.
- *End Hour [Int] Startwert = 24*: ist die obere Grenze des Intervalls.

Effekte

Gefahrenzone hinzufügen: Dieser Effekt speichert für eine gewisse Zeit die Position des nächsten Feindes als Gefahrenzone.
Parameter:

- *Duration [Int] Standardwert = 7200*: gibt die Dauer in Schritten an, bis die Gefahrenzone wieder aus dem Gedächtnis des NSCs gelöscht werden soll.
- *Lock [Int] Startwert = 50*: gibt an, wieviele Schritte vergehen müssen, bevor eine weitere Gefahrenzone der Liste des NSCs hinzugefügt werden kann, um zu vermeiden dass eine „Spur“ von Gefahrenzonen gezogen wird, wenn ein NSC verfolgt wird.

Sicherheitszone hinzufügen: Siehe Gefahrenzone hinzufügen, nur für befreundete Wachen.

Wahrnehmungsradius verändern: Dieser Effekt modifiziert den Wahrnehmungsradius um einen festgelegten Wert. Dieser Wert stellt keinen Parameter dar, da er nicht von den NSCs beeinflusst werden kann, sondern ist eine Folge ihrer Unaufmerksamkeit, während sie andere Tätigkeiten ausführen.
Parameter: keine.

Feind verfolgen: Der NSC verfolgt einen Feind und greif ihn an, sobald er nahe genug herangekommen ist.
Parameter:

- *Max Chase Time(Sec) [Int] Standardwert = 100*: gibt die Dauer in Schritten an, bis der NSC die Verfolgung abbricht.
- *No Chase for x Sec [Int] Startwert = 100*: gibt an, wieviele Schritte vergehen müssen, bevor der NSC wieder eine Verfolgung aufnimmt.

Verteidigen: Der NSC greift den nächsten Feind an, wenn dieser sich in Angriffsreichweite befindet, andernfalls flieht er vor dem nächsten Feind.
Parameter: keine

Feldarbeit: Der NSC sucht ein Feld und arbeitet auf diesem. Falls eine gewisse Menge an Feldfrüchten eingesammelt wurde, bringt er sie in die Stadthalle.
Parameter:

- *Collect [Int] Standardwert = 5*: gibt an, wie viele Feldfrüchte gesammelt werden sollen, bevor der NSC sie abliefert.

Fliehen: Der NSC flieht vor dem nächsten Feind.
Parameter: keine.

Patrouillieren: Der NSC läuft alle Gefahrenzonen periodisch ab.
Parameter:

- *Prob [Float] Standardwert = 0.1*: gibt an, mit welcher Wahrscheinlichkeit der NSC statt der nächsten Gefahrenzone einen zufälligen Punkt in der Nähe der nächsten Gefahrenzone ansteuern soll.

Leise Bewegen: Dieser Effekt aktiviert den Schleichmodus des NSCs. Dadurch verringert sich seine Bewegungsrate und er wird schwerer entdeckt.
Parameter:

- *Movementreduction [Float] Standardwert = 0.5*: gibt an, um wie viel Prozent die Bewegungsrate, und damit gleichzeitig die Chance entdeckt zu werden, gesenkt werden soll.

Rennen: Dieser Effekt aktiviert den Rennenmodus des NSCs. Dadurch erhöht sich seine Bewegungsrate um einen gewissen Faktor, er wird aber schneller müde und kann nur eine gewisse Zeit lang rennen.
Parameter:

- *Runfactor [Int] Standardwert = 3*: gibt an, um das wie viel-fache die Bewegungsrate gesteigert werden soll. Die Zeit, die ein NSC rennen kann, nimmt exponentiell zu diesem Faktor ab.

Bedürfnisse erfüllen: Dieser Effekt bewegt den NSC dazu, das Bedürfnis zu erfüllen, welches die Vorbedingung „Bedürfnisse niedrig“ aktiviert hat.
Parameter:

- *Hunger [Int] Standardwert = 3*: gibt an, wie viele Essensvorräte der NSC mitnehmen soll, wenn er sich in einem Haupthaus befindet.
- *Water [Int] Standardwert = 10*: gibt an, wie viele Wasservorräte der NSC mitnehmen soll, wenn er sich in einem Haupthaus befindet.

Wandern: Der NSC läuft ziellos umher.
Parameter: keine.

5.3.2. Regeln und Aktionsregeln

Hier werden einige Regeln vorgestellt, welche von den NSC-Gruppen innerhalb der Simulation verwendet worden sind.

Alle NSC-Gruppen

- *Bedürfnisse*:
  - *Vorbedingungen*: Bedürfnisse niedrig
  - *Enter-Effekte*: keine
  - *Update-Effekte*: Bedürfnisse erfüllen
  - *Exit-Effekte*: keine

Bauern

- *Wache gesichtet*:
  - *Vorbedingungen*: Kreatur (Wache) gesichtet
  - *Enter-Effekte*: Sicherheitszone hinzufügen
  - *Update-Effekte*: keine
  - *Exit-Effekte*: keine
- *Flucht*:
  - *Vorbedingungen*: Feind gesichtet
  - *Enter-Effekte*: Gefahrenzone hinzufügen
  - *Update-Effekte*: Flucht, Rennen
  - *Exit-Effekte*: keine
- *Arbeiten*:
  - *Vorbedingungen*: Zeit
  - *Enter-Effekte*: keine
  - *Update-Effekte*: Feldarbeiten
  - *Exit-Effekte*: keine

Wachen

- *Patrouillieren*:
  - *Vorbedingungen*: Nicht erschöpft, Zeit
  - *Enter-Effekte*: keine
  - *Update-Effekte*: Patrouillieren
  - *Exit-Effekte*: keine
- *Selbst verteidigen*:

  - *Vorbedingungen*: Nicht erschöpft, Feind (Raubtier) gesichtet (nahe Distanz)
  - *Enter-Effekte*: keine
  - *Update-Effekte*: Verteidigen
  - *Exit-Effekte*: keine

- *Verfolgen*:
  - *Vorbedingungen*: Nicht erschöpft, Feind (Goblin) gesichtet (mittlere Distanz)
  - *Enter-Effekte*: keine
  - *Update-Effekte*: Feind verfolgen, Rennen
  - *Exit-Effekte*: keine

Goblins

- *Flucht*:
  - *Vorbedingungen*: Feind (Wache) gesichtet
  - *Enter-Effekte*: keine
  - *Update-Effekte*: Fliehen, Rennen
  - *Exit-Effekte*: keine

- *Selbst verteidigen*:
  - *Vorbedingungen*: Nicht erschöpft, Feind (Guard) gesichtet (nahe Distanz)
  - *Enter-Effekte*: keine
  - *Update-Effekte*: Verteidigen
  - *Exit-Effekte*: keine

- *Verfolgen*:
  - *Vorbedingungen*: Nicht erschöpft, Feind (Bauer, Beutetier) gesichtet
  - *Enter-Effekte*: keine
  - *Update-Effekte*: Feind verfolgen, Rennen
  - *Exit-Effekte*: keine

Raubtiere

- *Flucht*:
  - *Vorbedingungen*: Feind (Wache, Goblin) gesichtet (mittlere Distanz)
  - *Enter-Effekte*: keine
  - *Update-Effekte*: Fliehen, Rennen
  - *Exit-Effekte*: keine

- *Selbst verteidigen*:

  - *Vorbedingungen*: Nicht erschöpft, Feind (Wache, Goblin) gesichtet (nahe Distanz)
  - *Enter-Effekte*: keine
  - *Update-Effekte*: Verteidigen
  - *Exit-Effekte*: keine

- *Jagen, schleichend*:
  - *Vorbedingungen*: Nicht erschöpft, Feind (Beutetier) gesichtet
  - *Enter-Effekte*: keine
  - *Update-Effekte*: Feind verfolgen, Schleichen
  - *Exit-Effekte*: keine

- *Jagen, rennend*:
  - *Vorbedingungen*: Nicht erschöpft, Feind (Beutetier) gesichtet (mittlere Distanz)
  - *Enter-Effekte*: keine
  - *Update-Effekte*: Feind verfolgen, Rennen
  - *Exit-Effekte*: keine

Beutetiere

- *Flucht*:
  - *Vorbedingungen*: Feind gesichtet
  - *Enter-Effekte*: keine
  - *Update-Effekte*: Fliehen, Rennen
  - *Exit-Effekte*: keine

- *Wandern*:
  - *Vorbedingungen*: Zeit
  - *Enter-Effekte*: keine
  - *Update-Effekte*: Wandern
  - *Exit-Effekte*: keine

6. Ergebnisse und Erkenntnisse

Der Ansatz der Schwarmintelligenz als Steuerungselement für die KI in Videospielen ist ein interessanter Ansatz. Im Rahmen diese Arbeit wurden viele Vor- und Nachteile, die die SI mit sich bringt, aufgedeckt und sollen nun hier vorgestellt werden. Aber zuerst werden die gewonnenen Erkenntnisse im Bereich der Spiele-KI im Allgemeinen dargestellt.

6.1. Komplexität der Spielwelt

Wenn man etwas simulieren möchte, ist es immer sehr hilfreich, den zu simulierenden Vorgang und seine Umgebung so genau wie möglich nachzubilden. Ein gutes Beispiel für fehlende Informationen und Komplexität der Modelle sind Wettervorhersagen. Bei der Simulation von Lebewesen, wie es in Videospielen versucht wird, hängt die Qualität ebenfalls von der Komplexität der Simulation ab. Allerdings ist die Komplexität nur eine notwendige, keine hinreichende Bedingung für eine glaubwürdige Simulation. Im Allgemeinen kann gesagt werden, je komplexer der Simulator oder das Spiel, desto glaubwürdiger kann die Simulation von Lebewesen werden.

Dass die Komplexität nicht nur für die korrekte Simulation von Lebewesen wichtig ist, sondern auch dazu beitragen kann, fehlerhafte Verhaltensweisen aufzudecken, wurde im Rahmen dieser Arbeit mit Hilfe der Tag- und Nachtzyklen festgestellt.

6.1.1. Tag- und Nachtzyklus

Die Uhrzeit ist ein wichtiger Aspekt bei der Simulation von Lebewesen und muss von der Spiele-KI beachtet werden. Die Tageszeit bestimmt maßgeblich, welche Aktionen ein NSC durchführen sollte. So erscheint es wenig sinnvoll, wenn ein NSC nachts arbeitet und tags schläft. Solch ein Verhalten würde sofort als nicht realistisch eingestuft werden, es sei denn, es gibt einen speziellen Grund dafür. Jedoch ist die Spiele-KI noch weit davon entfernt, dass es zu solchen speziellen Gründen kommen kann. Selbst wenn es einen Grund dafür gäbe, so wird der Spieler wahrscheinlich nicht die Hintergründe kennen und das Verhalten als eher nicht realistisch einstufen. Daher ist es sehr wichtig, dass die NSCs ihre Aktionen an die Tageszeit anpassen.

Ein Problem, welches während der Simulationen innerhalb dieser Arbeit beobachtet wurde, ist die zeitliche Verschiebung der Tätigkeiten: Ein Bauer-NSC beginnt am ersten simulierten Tag morgens um 8 Uhr seinen Tag und beginnt mit der Arbeit. Mit Fortschreiten der Simulation verschiebt sich seine Arbeitszeit auf dem Feld von den Morgenstunden über den Mittag hin zu Arbeitszeiten bei Nacht. Dieser Effekt wurde anhand der stark schwankenden Fitnesswerte der *Feldarbeiten-Funktion* während der Simulation entdeckt.

Wie man an Abbildung 6.1 sehr gut erkennen kann, schwanken die Fitnesswerte periodisch bei aktiviertem Tag- Nachtzykus. Es dauert abwechselnd 4 oder 5 Lernintervalle,

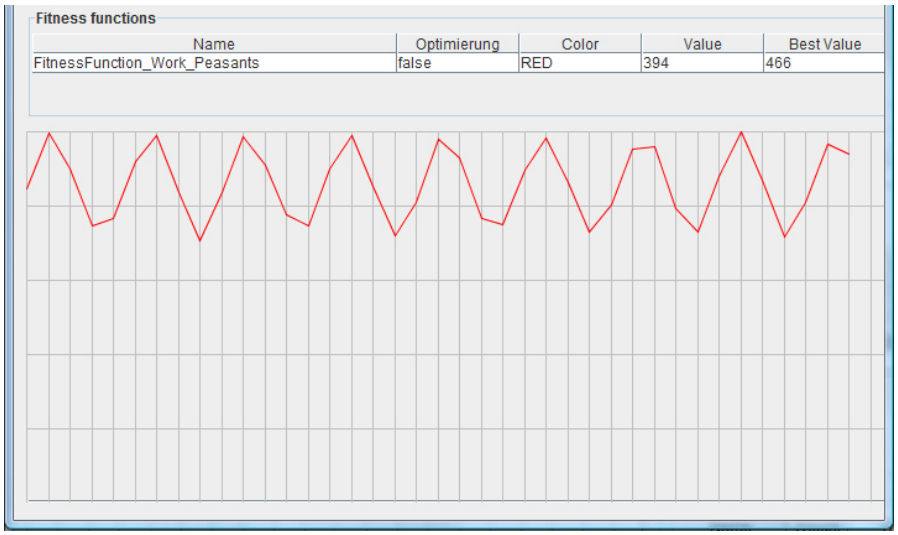


Abbildung 6.1.: Die Verschiebung der Arbeitszeiten ist bei aktiviertem Tag-Nachtzyklus anhand der schwankenden Fitnesswerte gut erkennbar

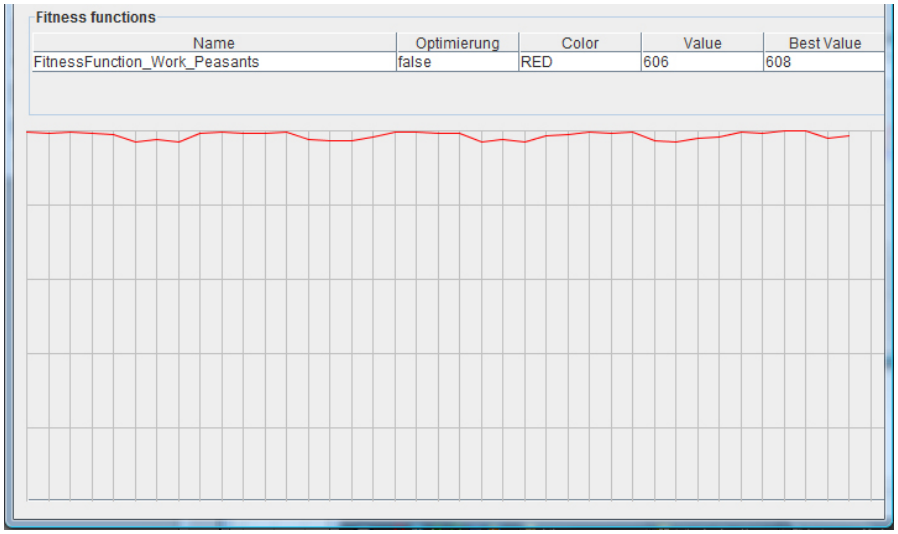


Abbildung 6.2.: Die Verschiebung der Arbeitszeiten ist bei deaktiviertem Tag-Nachtzyklus nicht mehr erkennbar

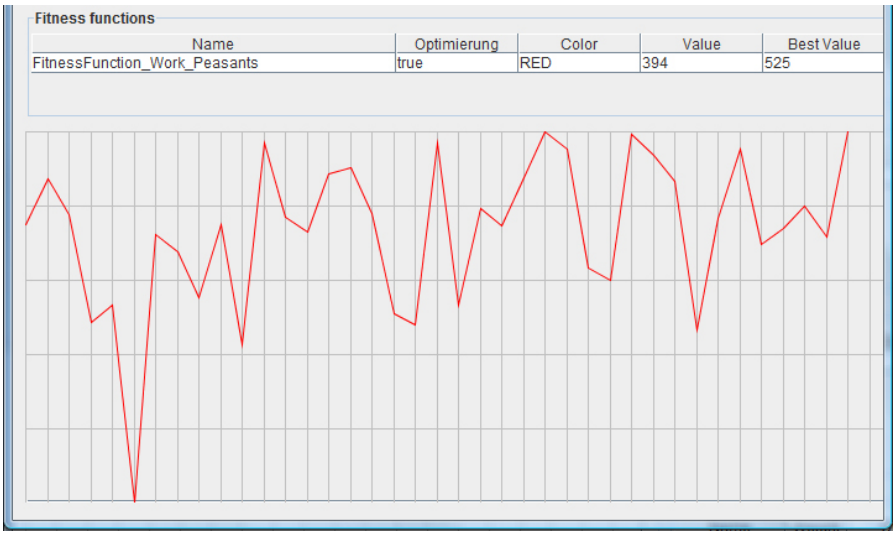


Abbildung 6.3.: Verlauf der Fitnesswerte bei aktiviertem Tag- und Nachtzyklus bei aktivierter Optmierung der Parametereinstellungen.

bis der alte Wert wieder erreicht wurde. Demnach beträgt die Verschiebung der Arbeitszeit 5,33 Stunden pro Woche[1]. Abbildung 6.2 zeigt den Verlauf der Fitnessfunktion mit deaktiviertem Tag- und Nachtzyklus. Die Verschiebung ist nicht mehr zu erkennen, da die Schwankungen viel geringer ausfallen.

Bei aktivierter Optimierung der Parametereinstellungen (Abbildung 6.3) wurde innerhalb von 40 Lernintervallen ein um 12,6% besserer Maximalwert (525 zu 466) der Fitness erreicht. Tabelle 6.1 zeigt die Ergebnisse nach 20 Simulationsdurchläufen. Dabei wurde ein Maximalwert von 606 mit einem Mittelwert von 498 und einer Standardabweichung von 23,9 erreicht, eine Steigerung des Maximalwerts um 30%. Es wurde nur die Funktion *Feldarbeiten* bei aktiviertem Tag- Nachtzyklus optimiert. Die periodischen Schwankungen konnten jedoch nicht beseitigt werden (s. Abbildung 6.3).

6.2. Erkenntnisse zur SI als KI in Videospielen

In diesem Abschnitt folgen die Erkenntnisse über den Einsatz der SI in Videospielen und eine Bewertung der Eignung dieser Technik hinsichtlich der Simulation von Lebewesen. Zunächst werden die Vorteile und anschließend die Nachteile der SI erläutert.

6.2.1. Vorteile

Die Vorteile bei der Nutzung der SI als KI sind zum einen die SI spezifischen Vorteile des geringen Resourcenverbrauchs und der Skalierbarkeit. Wie schon zu Anfang von Kapitel 5 erwähnt wurde, kann die Simulation mit einer hohen Geschwindigkeit von 175.000 facher Realzeit ablaufen.

[1] 24 Stunden Verschiebung innerhalb von 4,5 Wochen

Beste Fitness	**Start Hour**	**SW**	**End Hour**	**SW**	**Collect**	**SW**
545	0	1,7	23	11	5	3,8
598	1	1,2	23	4 ,7	13	14,5
498	3	4	24	1,3	3	1,4
552	1	7	22	7	6	1,1
539	3	12	22	12	11	14,5
564	0	0,9	23	2,3	6	2,3
528	1	2,8	24	10,4	4	6,3
568	0	2,5	23	6,2	8	5,3
572	2	12	24	7,3	11	14,5
598	1	12	23	1,5	13	3,3
532	1	1,7	24	2,3	4	3,8
591	1	4,6	24	1,4	10	10,6
606	0	6,7	24	6,7	11	7,4
568	2	9,7	24	5,1	8	8,3
524	1	2,8	23	7,8	4	1,8
588	0	3,7	23	4,6	14	14,5
558	2	3,1	22	8,7	9	11
600	0	1,6	24	1,8	15	13,2
574	1	6,2	22	1,4	14	10,5
585	1	3,9	23	6,3	11	11,6
MW	**MW**	**MW**	**MW**	**MW**	**MW**	**MW**
564,4	1,05	5,01	23,2	5,49	9	7,99
σ	σ	σ	σ	σ	σ	σ
23,96	0,68	3,06	0,64	2,85	3,3	4,34

Tabelle 6.1.: 20 Durchläufe mit jeweils 40 Iterationen (280 simulierte Tage). SW = Beste Schrittweite des vorangegangenen Parameters, MW = Mittelwert, σ = Standardabweichung.

Da die Regeln mit Hilfe von Vorbedingungen und Effekten zusammengesetzt werden können, welche wiederum durch ihre Parametereinstellungen angepasst werden können, ist es möglich, neue Verhaltensweisen in kurzer Zeit zusammenzusetzen. Das System hat sich daher als sehr modular herausgestellt.

6.2.2. Nachteile

Fehlendes Gedächtnis

SI-Algorithmen sind *Greedy-Algorithmen*. Zu jedem Zeitpunkt einer Entscheidung wird nur die aktuelle Lage betrachtet, ohne dabei auf vorherige Ereignisse zurückzublicken. Genau diese Eigenschaft ist einer der großen Nachteile der SI im Einsatz als AI in Videospielen. Während der Simulation sind Situationen aufgetreten, welche auf das Fehlen eines Gedächtnisses zurückzuführen sind:

Situation 1: Ein Tier wandert in der Spielwelt umher. Nach einiger Zeit bekommt das Tier Hunger und es sucht nach einer Nahrungsquelle. Da es nicht weiß, wo sich Nahrung befindet, sucht es zufällig die Umgebung ab, trifft nach einiger Zeit auf eine Futterquelle und frisst sich satt. Danach geht es wieder anderen Tätigkeiten nach. Bekommt es zu einem späteren Zeitpunkt wieder Hunger, muss das Tier erneut nach einer Futterquelle suchen, da es sich an die schon gefundene Futterquelle nicht mehr erinnert.

Situation 2: Ein Goblin wandert in der Umgebung umher auf der Suche nach Beutetieren oder Bauern. Nach einiger Zeit will er zu seiner Höhle zurückkehren, um dort zu schlafen und bewegt sich darauf zu. Unglücklicherweise befindet sich nun zwischen dem Goblin und seiner Höhle das Haus der Wachen, welche sich dort aufhalten. Der Goblin bemerkt die Wachen und flieht in die entgegengesetzte Richtung, weg von den Wachen und weg von seiner Höhle. Sofort nachdem die Wachen außer Sichtweite geraten, dreht der Goblin wieder um und will zu seiner Höhle zurückkehren. Dabei entdeckt er wieder die Wachen und kehrt um, usw. Der Goblin vergisst die Gefahr, sobald sie außerhalb seiner Sichtweite ist.

Durch solche Verhaltensweisen wird deutlich, dass SI-Algorithmen über kein Gedächtnis verfügen. Dies verhindert auch das Planen von Aktionsfolgen, welches benutzt werden können, um gewünschte, komplexe Verhaltensweisen hervorzurufen. Es gibt jedoch auch Beispiele aus der Natur (z.B. Ameisen), bei denen die SI zu sehr komplexen Verhaltensweisen führen kann, ohne das die einzelnen Individuen ein Gedächtnis besitzen. Jedoch ist es sehr aufwendig solche Regelsätze zu entwickeln, um den gewünschten Effekt zu erzielen. Dies wird deutlich, da seit dem ersten SI-Algorithmus 1995 nur eine Handvoll neuer Algorithmen entwickelt wurden, die sich wirklich voneinander unterscheiden. Die gefundenen, unterschiedlichen SI-Algorithmen sind:

- ACO
- PSO
- Bienen-Algorithmen

- Wespen-Algorithmen
- Stochastic Diffusion Search

Ein weiterer Punkt, warum auf ein Gedächtnis nicht verzichtet werden kann, ist, dass in Videospielen oft Lebewesen simuliert werden sollen, welche in der Realität ein Gedächtnis besitzen, wie Menschen, andere humanoide Lebewesen oder Tiere. Es fällt demnach, wie an dem oben gezeigten Beispiel deutlich wurde, sehr schnell auf, wenn diese NSCs kein Gedächtnis besitzen.

6.2.3. Portierungsproblem: Von der Intelligenz des Einzelnen zur Intelligenz der Masse

Unter den SI-Algorithmen findet man sehr viele, die sich stark an ihr natürliches Vorbild anlehnen. Das ist auch nicht weiter verwunderlich, da es in der SI genau darum geht. Es ist jedoch sehr aufwendig, mit Hilfe von SI, Verhaltensweisen, wie das Verhalten von Menschen, zu generieren, die nicht auf einer Art von Schwarmintelligenz beruhen.

In dieser Arbeit wird dieses Problem als Portierungsproblem bezeichnet. Es muss versucht werden, das Verhalten einer Intelligenzform, der Intelligenz des Einzelnen, in eine andere Intelligenzform, der Intelligenz der Masse, zu überführen. Bei der Entwicklung von SI-Algorithmen gab es dieses Problem nicht. Dort musste man „nur“ erkennen, nach welchen Regeln das einzelne Individuum handelt und diese nachbilden. Trotzdem gibt es, wie unter 6.2.2 beschrieben, nur sehr wenige, unterschiedliche SI-Algorithmen. Die meisten SI-Algorithmen kopieren das Verhalten von Lebewesen, die schon bereits auf SI beruhen.

6.2.4. Fehlende Lerneffekte während des Spielens

In SI-Algorithmen sind Lerneffekte nicht vorgesehen. Zwar können durch Particle Swarm Optimization (PSO) oder Ant Colony Optimization (ACO) Optimierungsprobleme gelöst werden, die eingesetzten Regeln der PSO (*Evaluieren*, *Vergleichen*, *Imitieren*) und ACO werden jedoch nicht verändert. Um Änderungen an den Verhaltensweisen der NSCs zu bewirken, müssen ihre Regeln jedoch während der Simulation verändert werden können. Dies ist zwar, wie in dieser Arbeit gezeigt wurde, mit Hilfe von Parametereinstellungen der Regeln möglich, erfordert aber eine lange Lernphase während der Entwicklungszeit, um vorteilhafte Parametereinstellungen zu finden. Der Suchraum der zu optimierenden Parameter ist sehr groß und es benötigt viele Individuen und viele Iterationen, bevor sinnvolle Parameter gefunden werden, da das Verhalten erst eine gewisse Zeit lang getestet werden muss. Eine Iteration stellt dabei eine simulierte Woche dar.

Um zu untersuchen, wie viele Iterationen nötig sind, damit sehr gute Ergebnisse gefunden werden, wurde dies mit dem Simulator untersucht. Für diese Untersuchung wird hier die Fitnessfunktion *Bedürfnisse erfüllen* genutzt. Der Tag- Nachtzyklus ist aktiviert und die Fitnessfunktion wird mit schlechten Parametereinstellungen gestartet. Die Wahl schlechter Parametereinstellungen beruht auf der Annahme, dass nicht für alle Parametereinstellungen a priori gute Werte gewählt werden können.

Die Tabellen 6.2 und 6.3 zeigen die Ergebnisse der Durchläufe bei 40 bzw. 750 Iterationen. Dabei wird deutlich, dass bei einem *(1+1)-ES* 40 Iterationen nicht ausreichen,

Beste Fitness	**HT**	**SW**	**TT**	**SW**	**HC**	**SW**	**WC**	**SW**
260232	2,5	0,7	1,5	0,8	9	7,8	14	10
266718	0,9	5	1,3	3	27	8,5	22	11
267804	0,8	4,3	1,3	2,5	0	3	18	24
239190	1,8	5	2,1	2	2	13	9	24
267294	2,9	3,5	1,5	2,3	3	3	2	10
278772	0,9	1,6	1,3	2,4	11	3,1	4	7,3
258006	0,2	0,1	0,1	2,2	15	3,5	8	7,7
265872	0,9	1	0,8	0,4	9	2,7	21	2,5
283284	1,7	1,2	1,2	1,7	4	0,3	7	1,5
287976	1	0,8	8,1	5	26	14	8	3,2
239100	0,1	0,1	0,8	0,7	3	1,6	15	8,2
256014	1,2	0,7	0,8	2,1	4	5,7	4	8,6
262896	0,4	0,5	0,6	0,9	4	2	3	3,7
269478	1,1	0,3	0,8	2,2	4	1,5	4	0,9
290544	1,7	2,9	0,1	0,6	3	3,1	5	4,9
268890	8,9	5	0,1	1,5	2	8,7	15	9,1
236376	0,8	3,7	1,5	1,7	6	6,6	1	6,1
251424	3	2,1	0,7	1,6	9	10,3	5	19
234186	1,4	1,2	0,5	1,8	4	3,9	6	3,4
253146	0,7	1,4	1	0,3	3	3,2	7	3,5
MW	**MW**	**MW**	**MW**	**MW**	**MW**	**MW**	**MW**	**MW**
261860	1,65	2,06	1,31	1,79	7,4	5,28	8,9	8,43
σ	σ	σ	σ	σ	σ	σ	σ	σ
12907	1,1	1,51	0,82	0,77	5,42	3,24	5,17	4,83

Tabelle 6.2.: 20 Durchläufe mit jeweils 40 Iterationen (280 simulierte Tage). HT = Hunger Treshold, TT = Thirst Treshold, HC = Hunger Count, WC = Water Count, SW = Beste Schrittweite des vorangegangenen Parameters, MW = Mittelwert, σ = Standardabweichung.

Beste Fitness	**HT**	**SW**	**TT**	**SW**	**HC**	**SW**	**WC**	**SW**
302322	0,2	1,7	3,5	1,5	2	7,8	9	2
311107	0,2	0,8	2,9	1,1	3	3,1	10	2,7
288144	2,1	0,3	3,1	3,4	5	4	9	3,1
297451	0,7	0,9	2,3	1,2	3	0,5	11	0,5
305512	0,4	1,3	3,1	2,1	3	3,7	9	0,7
289840	0,6	1,6	2,2	1,4	2	4,4	7	2,1
299610	1,5	1,2	4,6	0,1	3	0,9	7	1,5
288363	1,5	0,9	0,1	0,9	2	2,2	11	2,5
294564	1,2	1,1	2,1	0,5	4	3,8	8	4,2
289218	1,2	0,4	1,5	0,6	3	0,5	12	0,8
289080	0,6	1	2	3,1	4	1,5	8	24,5
MW	**MW**	**MW**	**MW**	**MW**	**MW**	**MW**	**MW**	**MW**
293102,5	1,13	1,09	2,54	1,39	3	3,56	8,7	4,7
σ	σ	σ	σ	σ	σ	σ	σ	σ
4219,6	0,42	0,41	1,12	0,79	0,8	2,08	1,24	3,96

Tabelle 6.3.: Mehrere Durchläufe mit 750 Iterationen (5250 simulierte Tage). HT = Hunger Treshold, TT = Thirst Treshold, HC = Hunger Count, WC = Water Count, SW = Beste Schrittweite des vorangegangenen Parameters.

um ein zuverlässiges Ergebnis zu erhalten. Bei 40 Iterationen liegt der Mittelwert der Fitness bei 261860 mit einer Standardabweichung von 12907 (4,93%). Dabei schwanken die Werte der Parameter *Water Count* und *Hunger Count*, mit Werten zwischen 0 und 27 bzw. zwischen 1 und 22, besonders stark. Bei 750 Iterationen schwanken die Werte nicht mehr so stark.

Im Grunde genommen bedeutet dies, sind die Regeln einmal erstellt worden (während der Entwicklung des Spiels), können sie auf Grund ihrer enormen Lernzeit nicht mehr verändert werden. Bei den Veränderungen durch EAs können auch sehr unsinnige Parametereinstellungen auftreten. Abbildung 6.4 zeigt ein Beispiel für die hohe Anzahl der Änderungen, die zu schlechten Ergebnissen geführt haben. Daher sollten die Parametereinstellungen, die das Verhalten der NSCs maßgeblich beeinflussen, nicht während des Spielbetriebs optimiert werden, da ein Spieler sonst diese unsinnigen Verhaltensweisen beobachten könnte. Dies kann zwar durch die Einführung von gewissen Grenzen bei den Parametern eingedämmt werden, erfordert jedoch a priori-Wissen.

Im Rahmen dieser Arbeit wurden keine Möglichkeit gefunden, wie verhindert werden kann, dass bei der Optimierung schlechte Parametereinstellungen getestet werden.

Glücklicherweise benötigen viele Spiele keine lernenden NSCs. In vielen Situationen ist die Begegnungsdauer von Spieler und NSC kurz genug, so dass das Fehlen des Lernens nicht auffällt. So wird einer feindlichen Begegnung meistens nicht die Zeit gegeben, ihr Verhalten anzupassen, da sie die Begegnung in der Regel nicht überlebt. Es ist wohl von Spieler zu Spieler unterschiedlich, ob er nun bevorzugt, dass andere NSCs von diesem Typ nun ihr Verhalten auf Grund der Erfahrungen der verstorbenen NSCs anpassen oder

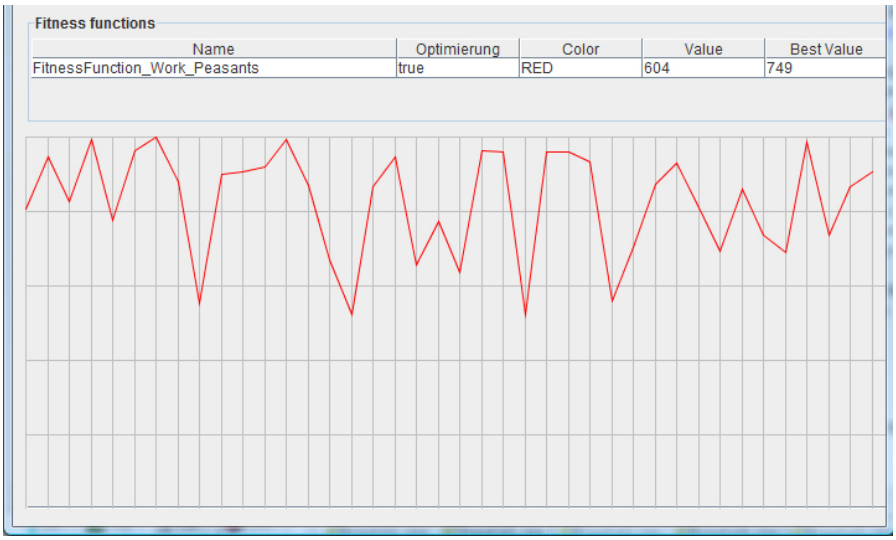


Abbildung 6.4.: Verlauf über 40 Iterationen. Viele Veränderung beeinflussen die Fitness negativ. Der Einfluss des Tag- und Nachtzyklus wurde deaktiviert, damit die Schwankungen nur auf die Veränderungen der Parameter zurückzuführen ist.

dass keine Anpassung stattfindet.

Bei anderen Spielen, wie Strategiespielen, ist die Lernfähigkeit jedoch erwünscht. Dabei handelt es sich jedoch um eine andere Art von Lernen, die nicht einen einzelnen NSC betrifft, sondern viel mehr die Spiele-KI, die die taktischen Entscheidungen trifft. Diese Lerneffekte sind nicht ohne weiteres miteinander vergleichbar.

6.2.5. Kein gezieltes Verhalten

Um ein neues, gezieltes Verhalten zu generieren, benötigt man sehr viel Zeit für die Entwicklung der Regeln. Sicherlich geht es wesentlich schneller, einigermaßen sinnvolles Verhalten mit neuen Regeln zu generieren, jedoch muss dies nicht den Vorstellungen des Entwicklers entsprechen. Jedes Mal, wenn die Regeln angepasst werden müssen, müssen die neuen Verhaltensweisen beobachtet und bewertet werden. Im Rahmen dieser Arbeit wurde deutlich gemacht, dass gerade der Prozess der Bewertung des Verhaltens eine wesentlich Rolle spielt, weshalb eine Automatisierung nicht ohne weiteres möglich ist.

Die Tabellen 6.4 zeigt, dass es schwierig ist, ein Verhalten zu generieren, welches das Ziel hat, dass die Bauern nur während des Tages arbeiten und nicht in der Nacht. Da der Parameter *Collect* einen zu großen Einfluss auf die Fitness hat, werden die anderen beiden Parameter innerhalb der 40 Iterationen nicht richtig optimiert. Um Nebeneffekte, die bei der Optimierung anderer Fitnessfunktionen auftreten können, auszuschließen, wurde nur die Fitnessfunktion *Feldarbeiten* optimiert. Der Tag- Nachtzyklus ist dabei aktiviert.

Der Mittelwert der Fitnessfunktion liegt bei 564,4 mit einer Standardabweichung von 23,96 (4,25%).

Tabelle 6.5 zeigt hingegen, dass 200 Iterationen ausreichten, damit der Parameter *Collect* optimiert werden konnte, jedoch konnte die Arbeitszeit *Start Hour* und *End Hour*

Beste Fitness	**Start Hour**	**SW**	**End Hour**	**SW**	**Collect**	**SW**
545	0	1,7	23	11	5	3,8
598	1	1,2	23	4,7	13	14,5
498	3	4	24	1,3	3	1,4
552	1	7	22	7	6	1,1
539	3	12	22	12	11	14,5
564	0	0,9	23	2,3	6	2,3
528	1	2,8	24	10,4	4	6,3
568	0	2,5	23	6,2	8	5,3
572	2	12	24	7,3	11	14,5
598	1	12	23	1,5	13	3,3
532	1	1,7	24	2,3	4	3,8
591	1	4,6	24	1,4	10	10,6
606	0	6,7	24	6,7	11	7,4
568	2	9,7	24	5,1	8	8,3
524	1	2,8	23	7,8	4	1,8
588	0	3,7	23	4,6	14	14,5
558	2	3,1	22	8,7	9	11
600	0	1,6	24	1,8	15	13,2
574	1	6,2	22	1,4	14	10,5
585	1	3,9	23	6,3	11	11,6

Tabelle 6.4.: 20 Durchläufe mit jeweils 40 Iterationen (280 simulierte Tage). SW = Beste Schrittweite des vorangegangenen Parameters.

Beste Fitness	**Start Hour**	**SW**	**End Hour**	**SW**	**Collect**	**SW**
594	0	0,2	24	4,9	9	1,2
583	2	6,1	22	0,3	11	3,4
575	0	4,1	23	2,3	10	3,1
585	2	3	23	5	9	3,5
600	0	6,2	24	1,6	11	4,1
576	2	0,2	22	0,3	12	4,3
580	5	1,1	24	3,8	15	3,9
624	0	2,9	24	0,9	24	7,8
585	4	1	23	3	15	4,2
605	2	4,7	24	0,5	16	7,5
594	1	0,1	23	0,7	11	7,4
609	1	0,6	24	3,7	16	6,1
585	0	11,4	22	5,5	13	6,1
578	2	0,6	24	7,9	8	3,9
592	1	12	23	6,8	16	14,5
572	2	0,1	22	0,5	11	1,7
513	0	1,2	24	1,4	3	0,1
570	3	1	23	10,5	8	2,8
595	2	1	24	2,2	17	2,9
525	3	12	24	2,7	4	0,3
MW	**MW**	**MW**	**MW**	**MW**	**MW**	**MW**
582	1,6	3,48	23,3	3,23	11,95	4,44
σ	σ	σ	σ	σ	σ	σ
16,7	1,14	3,22	0,7	2,23	3,65	2,28

Tabelle 6.5.: 20 Durchläufe mit jeweils 200 Iterationen (1400 simulierte Tage). SW = Beste Schrittweite des vorangegangenen Parameters, MW = Mittelwert, σ = Standardabweichung

nicht auf ein realistisches Intervall eingegrenzt werden. Eine mögliche Ursache dafür, könnte es sein, dass die Parameter der Arbeitszeiten viel weniger Bedeutung haben als der Parameter *Collect*. Ein weiterer, möglicher Grund für die schlechten Ergebnisse in Bezug auf die Arbeitszeiten werden in Kapitel 6.4 ausgeführt.

Die Tatsache, dass das gezielte Hervorrufen von bestimmten Verhaltensmustern ein langwieriger Prozess ist, macht die SI für den Einsatz in Videospielen sehr unattraktiv. Anders als bei anderen Ansätzen, kann bei komplexeren Verhaltensweisen nicht gut abgeschätzt werden, wie lange es dauern wird, einen guten Regelsatz für das gewünschte Verhalten zu erstellen. Das Bestimmen der Verhaltensweisen ist für die Entwickler eine sehr wichtige Eigenschaft, denn sie wird benötigt, um die Geschichte eines Spiels in die richtige Bahn zu lenken.

Jedoch gibt es auch Situationen, in denen diese Eigenschaft nicht ganz so wichtig ist, wie z.B. bei der Simulation von „Statisten“, beispielsweise der Bevölkerung oder Tiere,

die nicht aktiv am Spielgeschehen teil nehmen. Bei feindlichen NSCs ist eine Kontrolle jedoch sehr wichtig.

6.3. Fazit

Betrachtet man die in dieser Arbeit herausgefundenen Vor- und Nachteile, fällt das Urteil über die Eignung der SI für die Simulation von Lebewesen wie Menschen oder Tieren ernüchternd aus. Die SI eignet sich nicht sehr gut für die Erschaffung von komplexen Verhaltensweisen, besonders nicht, wenn beabsichtigt wird, ganz bestimmte Verhaltensweisen zu simulieren. Der Aufwand für ein nicht vorhersehbares Ergebnis ist zu hoch, das Risiko, dass ein gewünschtes Verhalten nicht erzeugt wird, zu groß.

Für einfache NSCs bzw. Verhaltensmuster, wie Fluchtverhalten oder Gruppenbewegungen kann die SI zwar eingesetzt werden, jedoch fehlte im Rahmen dieser Arbeit die Zeit, um den Simulator komplex genug zu gestalten, damit komplexere Verhaltensweisen hätten erzeugt werden können.

Eine andere Einsatzmöglichkeit wäre der Einsatz in Simulationen von physikalischen Effekten wie der Schwerkraft oder der Kollision von Objekten. Die Regeln, nach denen diese Effekte funktionieren, sind ausreichend erforscht und es kommt zu keinen Problemen, wie dem Portierungsproblem oder dem Problem der Bewertung des Realismus. Die Simulation von diesen Effekten ist realistisch, falls die teilnehmenden Elemente den physikalischen Gesetzmäßigkeiten folgen.

6.4. Ausblick

Im Laufe der Arbeit wurden einige Möglichkeiten entdeckt, wie das Trainieren der Fitnessfunktionen effizienter oder besser hätte gemacht werden können. Jedoch war es nicht möglich, diese in der zur Verfügung stehenden Zeit umzusetzen, da sie zum Teil umfangreiche Änderungen des Simulators benötigt hätten.

Oszillationserkennung: Das Erkennen von Oszillationen könnte benutzt werden, um auf unerwünschte Verhaltensweisen der NSCs hinzuweisen. Ein Verfahren, welches dafür eingesetzt werden könnte, wäre das Erkennen häufiger Episoden nach
[Heikki Mannila, 1997]. Mit der Abbildung 4.3 in Kapitel 4.3.2 wurde auf die Existenz von häufigen Aktionssequenzen hingewiesen. Mit der Implementierung des oben genannten Verfahrens wurde im Rahmen dieser Arbeit zwar begonnen, sie konnte jedoch nicht innerhalb der zur Verfügung stehenden Zeit fertiggestellt werden.

Simulation von Szenarien: Eine Alternative zur Simulation von ganzen Wochen wäre die Entwicklung von Szenarien, in denen gezielt einzelne Verhaltensmuster getestet und bewertet werden könnten. Der Ablauf der Nahrungsaufnahme kann so in wenigen Sekunden bewertet werden und ggf. könnten Änderungen an den Regeln vorgenommen werden. Durch die Szenarien würde ebenfalls der Einfluss des Zufalls minimiert und es kämen viel häufiger die speziellen Situationen vor, die auch getestet werden sollen. Ein Beispiel hierfür wäre das Fluchtverhalten der NSCs. Bei einer rein zufälligen Simulation kommt es

in den meisten Fällen zu der immer gleichen Situation, dass ein NSC einen anderen verfolgt. Situationen, in denen ein NSC von zwei oder mehreren NSCs verfolgt wird, treten hingegen eher selten auf. Mit Hilfe der Szenarien könnte also gezielt ausgewählt werden, welche Situationen getestet werden sollten.

Für das Testen der Szenarien würde dann evtl. kein eigener Simulator benötigt, da diese kurzen Sequenzen auch in Echtzeit ablaufen könnten, ohne zu viel Zeit zu beanspruchen. Das Testen der Regeln nimmt dann zwar mehr Zeit in Anspruch, jedoch würde viel Zeit für die Entwicklung eines eigenen Simulators eingespart werden.

Es gibt jedoch auch einige, Dinge die beachtet werden müssen. Zum einen müssen geeignete Szenarien entwickelt werden, um die Verhaltensweisen zu überprüfen und zum anderen besteht das Problem der Überanpassung (Overfitting). Werden immer nur wenige Szenarien trainiert, kann es vorkommen, dass die NSCs in diesen zwar sehr gut, in anderen Situationen jedoch sehr schlecht reagieren.

Noch einfachere Verhaltensmuster: In Kapitel 4 und 5 wurde vorgeschlagen, keine einzelne Fitnessfunktion für den Grad an Realismus zu benutzten, sondern die einzelnen Verhaltensmuster wie „Überleben“ oder „Arbeiten“ separat zu betrachten. Dies fällt leichter, da die Verknüpfungen unter den einzelnen Aspekten entfallen, jedenfalls teilweise. In dem Beispiel zur Erstellung einer Fitnessfunktion für das Verhaltensmuster „Überleben“ wurde jedoch deutlich, dass die Separation noch nicht weit genug ging. Es gab Probleme bei der Vergabe von positiven Werten für die einzelnen Verhaltensweisen. So musste sehr genau darauf geachtet werden, dass die Fitnessfunktion nicht ein anderes, nicht gewünschtes Verhalten fördert wie das absichtliche Leben mit der Gefahr, indem der NSC wissentlich die Feinde immer in Sichtweite hält. Da die Simulation zu schnell abläuft, um das Verhalten zu beobachten, ist es schwierig, so einen Fehler in der Fitnessfunktion zu entdecken.

Eine Alternative wäre es, die Verhaltensmuster noch weiter zu unterteilen und noch mehr Fitnessfunktionen zu erstellen. Die „Überleben“-Funktion könnte unterteilt werden in eine Funktion, die die Distanz zu den Feinden berücksichtigt, eine die die Verletzungen berücksichtigt und eine, die die Zeit berücksichtigt, die der NSC verfolgt worden ist. Jede dieser Fitnessfunktionen würde dann von einer oder mehreren Regeln bestimmt werden. Daher müsste es zugelassen werden, dass eine Regel von mehreren Fitnessfunktionen optimiert werden kann. Dabei könnte man eine multikriterielle Optimierung verwenden.

Schwankungen der Fitnesswerte erkennen und eliminieren: Während die abschließenden Experimente für diese Arbeit erstellt wurden, sind besonders die periodischen Schwankungen der Fitnessfunktionen aufgefallen. Denn diese konnten trotz optimierter Parameter nicht beseitigt werden (Abbildung 6.5).

Daher wäre nach abschließender Optimierung der Parametereinstellungen eine weitere Optimierung in Bezug auf die Schwankung der Fitnesswerte nötig. Dieser Schritt sollte jedoch ausgeführt werden, bevor die Optimierung der Parametereinstellungen vorgenommen wird. Durch die Schwankungen wird die Anzahl der effektiven Mutationen stark reduziert, denn eine gute Mutation kann nur dann als solche erkannt werden, wenn sie zu einem Zeitpunkt auftritt, an dem die Schwankung ihren Maximalwert erreicht hat. Bei dem Beispiel, das Abbildung 6.5 zeigt, wird die Anzahl der effektiven Iterationen um

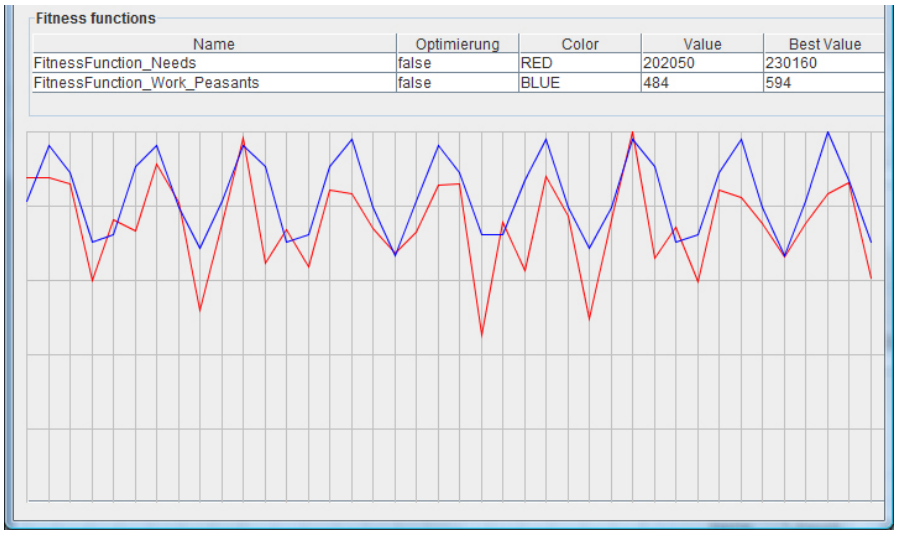


Abbildung 6.5.: Die Fitness bewegt sich zwar auf höherem Niveau, jedoch sind die Schwankungen der Fitnesswerte immer noch vorhanden.

den Faktor 4,5 reduziert.

Diese Eliminierung der Schwankungen konnte im Rahmen dieser Arbeit nicht vorgenommen werden, da es weiterer Verfeinerungen des Simulators bedurft hätte, besonders der Auswirkungen des Arbeitens bei Tag und Nacht. Aufgrund spezieller Eigenschaften des Simulators, die Bedürfnisse und derer Verwaltung betreffend, hätte dies umfangreiche Änderungen zur Folge gehabt.

Planung statt Emergenz: Viele der Probleme, die bei dem Einsatz von SI als Spiele-KI auftreten, können besser mit Techniken der klassischen KI bewältigt werden. Eines der wichtigsten wäre das Hervorrufen des gewünschten Verhaltens. Will ein Entwickler einen NSC zu eine, bestimmten, komplexen Verhalten bewegen, ist dies mit Hilfe der SI nur schwer möglich, da er nur die Möglichkeit hat, Regeln zu verändern ohne die genauen Auswirkungen zu kennen oder bestimmen zu können. Verwendet man hingegen einen planenden Ansatz, kann das gewünschte Ziel direkt vorgegeben werden. Es kostet den Entwickler zwar immer noch viel Zeit, die Pläne für die Erfüllung einer Zielvorgabe zu erstellen, jedoch könnte der Aufwand, im Vergleich zum „Herumprobieren“ beim emergenten Verhalten, gut abgeschätzt werden und wohl möglich weniger Zeit in Anspruch nehmen. Es würden sich klare, von einander abtrennbare Teilprobleme ergeben, die nacheinander bearbeitet werden könnten.

Literaturverzeichnis

[Angeline, 1998] Angeline, Peter (1998). *Evolutionary optimization versus particle swarm optimization: Philosophy and performance differences*. Evolutionary Programming VII, S. 601–610.

[B. Irrgang, 1990] B. Irrgang, J. Klawitter (1990). *Künstliche Intelligenz. Technologischer Traum oder gesellschaftliches Trauma?*. In: *Künstliche Intelligenz*, S. 7–54. Stuttgart: Hirzel, Edition Universitas Aufl.

[Brockhaus, 2003] Brockhaus (2003). *Der Brockhaus - Naturwissenschaft und Technik*, S. 1992. F.A. Brockhaus GmbH, Leipzig, Bibliografisches Institut und F.A. Brockhaus AG, Mannheim.

[Campbell et al., 2002] Campbell, Murray, A. J. Hoane, Jr. und F.-h. Hsu (2002). *Deep Blue*. Artif. Intell., S. 57–83.

[Champandard, 2007] Champandard, Alex J. (2007). *Top 10 Most Influential AI Games*.

[Clerc, 2006] Clerc, Maurice (2006). *Particle Swarm Optimization*. ISTE Publishing Company.

[Davison, 2005] Davison, Andrew (2005). *Killer Game Proframming in Java*. O'Reilly Media.

[Dorigo, 1992] Dorigo, Marco (1992). *Optimization, Learning and Natural Algorithms (in Italian)*. Doktorarbeit, Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy.

[Dorigo et al., 2006] Dorigo, Marco, M. Birattari, T. Stützle, U. Libre, D. Bruxelles und A. F. D. Roosevelt (2006). *Ant Colony Optimization - Artificial Ants as a Computational Intelligence Technique*. IEEE Comput. Intell. Mag, 1:28–39.

[Gerardo Beni, 1989] Gerardo Beni, Jing Wang (1989). *Swarm Intelligence*.

[G.Görz, 2003] G.Görz, C.-R. Rollinger, J. Schneeberger (Hrsg.) (2003). *Handbuch der Künstlichen Intelligenz*. Oldenbourg Verlag München Wien, Edition Universitas Aufl.

[Grassé, 1959] Grassé, Plerre-P (1959). *La reconstruction du nid et les coordinations interindividuelles chezBellicositermes natalensis etCubitermes sp. la théorie de la stigmergie: Essai d'interprétation du comportement des termites constructeurs*. Insectes Sociaux, 6(1):41–80.

[Heikki Mannila, 1997] Heikki Mannila, Hannu Toivonen, Inkeri A. Verkamo (1997). *Discovery of Frequent Episodes in Event Sequences*. Data Mining and Knowledge Discovery, 1(3):259–289.

[J. Kennedy, 2001] J. Kennedy, R. Eberhart, C. Russell (2001). *Swarm intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[J. Kennedy, 1995] J. Kennedy, R. Eberhart (1995). *Particle swarm optimization*. In: *Neural Networks, 1995. Proceedings., IEEE International Conference on*, Bd. 4, S. 1942–1948 vol.4.

[Julien Devade, 2003] Julien Devade, Dr. Jean-Yves Donnart, Dr. Emmanuel Chiva Stéphane Maruéjouls (2003). *Motivational graphs: A new architecture for complex behavior simulation*. In Rabin, Steve, ed.: *AI Game Programming Wisdom 2*. Charles River Media.

[Julio Obelleiro, 2008] Julio Obelleiro, Raúl Sampedro, David Hernández Cerpa (2008). *Rts terrain analysis image processing approach*. In Rabin, Steve, ed.: *Game AI Programming Wisdom 4*. Charles River Media, Cambridge, MA.

[Kent, 2001] Kent, Steven L. (2001). *The Ultimate History of Video Games: From Pong to Pokemon–The Story Behind the Craze That Touched Our Lives and Changed the World*. Three Rivers Press.

[Kinnebrock, 1996] Kinnebrock, Werner (1996). *Künstliches Leben - Anspruch und Wirklichkeit*. Oldenbourg Verlag München Wien, edition universitas ed.

[Langton, 1992] Langton, Christopher G. (1992). *Preface*. Addison-Wesley.

[Matt Gilgenbach, 2006] Matt Gilgenbach, Travis McIntosh (2006). *A flexible ai system through behavior composition*. In Rabin, Steve, ed.: *AI Game Programming Wisdom 3*. Charles River Media.

[Maurice Clerc, 2002] Maurice Clerc, James Kennedy (2002). *The particle swarm - explosion, stability, and convergence in a multidimensional complex space*. 6(1):58–73.

[Orkin, 2003] Orkin, J. (2003). *Applying goal-oriented action planning to games*. In Rabin, Steve, ed.: *AI Game Programming Wisdom 2*. Charles River Media.

[Paanakker, 2006] Paanakker, Ferns (2006). *Risk-adverse pathfinding using influence maps*. In Rabin, Steve, ed.: *Game AI Programming Wisdom 3*. Charles River Media, Cambridge, MA.

[Paul Marden, 2008] Paul Marden, Forrest Smith (2008). *Dynamically updating a navigation mesh via efficient polygon subdivision*. In Rabin, Steve, ed.: *Game AI Programming Wisdom 4*. Charles River Media, Cambridge, MA.

[Rabin, 2006] Rabin, Steve (2006). *AI Game Programming Wisdom 3*. Charles River Media.

[Rabin, 2008] Rabin, Steve (2008). *AI Game Programming Wisdom 4*. Charles River Media.

[Rabin, 2009] Rabin, Steve (2009). *#define game_ai*. Vortrag auf der Game Developers Conference in San Francisco.

[Rechenberg, 1973] Rechenberg, I. (1973). *Evolutionsstrategie Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Friedrich Frommann Verlag, Stuttgart-Bad Cannstatt.

[Silver, 2006] Silver, David (2006). *Cooperative pathfinding*. In Rabin, Steve, ed.: *Game AI Programming Wisdom 3*. Charles River, Cambridge, MA.

[Stuart J. Russell, 2003a] Stuart J. Russell, Peter Norvig (2003a). *Artificial Intelligence. A Modern Approach*, p. 21. Pearson Studium, 2 ed.

[Stuart J. Russell, 2003b] Stuart J. Russell, Peter Norvig (2003b). *Artificial Intelligence. A Modern Approach*, pp. 97–101. Pearson Studium.

[Stuart J. Russell, 2003c] Stuart J. Russell, Peter Norvig (2003c). *Artificial Intelligence. A Modern Approach*, pp. 422–430. Pearson Studium, 2 ed.

[Stuart J. Russell, 2008a] Stuart J. Russell, Peter Norvig (2008a). *Künstliche Intelligenz. Ein moderner Ansatz*, pp. 66–70. Pearson Studium, 2., überarb. a. ed.

[Stuart J. Russell, 2008b] Stuart J. Russell, Peter Norvig (2008b). *Künstliche Intelligenz. Ein moderner Ansatz*. Pearson Studium, 2., überarb. a. ed.

[Tozour, 2001] Tozour, Paul (2001). *Influence mapping*. In Rabin, Steve, ed.: *Game Programming Gems 2*. Charles River Media, Cambridge, MA.

[Tozour, 2002] Tozour, Paul (2002). *The evolution of the game ai*. In *AI Game Programming Wisdom 1*. Charles River Media.

[Trelea, 2003] Trelea, Ioan Cristian (2003). *The particle swarm optimization algorithm: convergence analysis and parameter selection*. Inf. Process. Lett., 85(6):317–325.

[Turing, 1950] Turing, Alan (1950). *Computing machinery and intelligence*. Mind Vol. LIX.

[Uwe Lämmel, 2008] Uwe Lämmel, Jürgen Cleve (2008). *Künstliche Intelligenz*. Hanser Verlag, third ed.

[Walter Brenner, 1998] Walter Brenner, Rüdiger Zarnekow, Hartmut Wittig (1998). *Intelligente Softwareagenten - Grundlagen und Anwendung*. Springer.

[Weicker, 2007] Weicker, Karsten (2007). *Evolutionäre Algorithmen (2. Auflage)*. Teubner, Stuttgart.