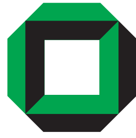


Universität Karlsruhe (TH)
Forschungsuniversität • gegründet 1825

Fakultät für **Informatik**

Lernende und selbstorganisierende Putzroboter

Diplomarbeit
von
Daniel Pathmaperuma

Betreuer : Prof. Dr. Hartmut Schmeck
Referent: Dipl.-Wi.-Ing. Urban Richter

Institut für Angewandte Informatik und Formale Beschreibungsverfahren (AIFB)

Forschungsgruppe Effiziente Algorithmen
Prof. Dr. Hartmut Schmeck

April 2008

Hiermit bestätige ich, die vorliegende Arbeit selbständig
durchgeführt und keine anderen als die angegebenen
Literaturhilfsmittel verwendet zu haben.

Karlsruhe, 30. April 2008

Daniel Pathmaperuma

Ich danke

Urban Richter
für Hinweise, Tips und stets ein offenes Ohr

Caroline Helde
Frank Eichiger
Sebastian Kaluza
für Anregungen, Verbesserungsvorschläge
und moralischen Beistand

meinen Eltern
für stete Unterstützung
und engelsgleiche Geduld

Karlsruhe im Jahr 2008

Daniel Pathmaperuma

Inhaltsverzeichnis

Abbildungsverzeichnis xii

Tabellenverzeichnis xiii

1 Einleitung & Motivation 1
1.1 Evolution & Entwicklung 3
1.2 Bionik & Organic Computing 5
1.3 Aufbau dieser Arbeit 6

2 Einführung & Problemstellung 7
2.1 Genetik 7
2.2 Evolution 8
2.2.1 Arten 9
2.2.2 Konkurrenz 9
2.2.3 Reproduktion 10
2.2.4 Selektion 10
2.2.5 Genese 11
2.3 Agenten 11
2.4 Problemstellung 12
2.4.1 Einordnung in das Organic Computing 12
2.4.2 Szenario 13

3 Stand der Technik 15
3.1 Schwärme und Pheromone 15
3.2 Gene und Evolution 17
3.2.1 Artificial Embryogeny 18
3.3 Kommunikation 19
3.4 Künstliche Neuronale Netze 19
3.5 NEAT 24
3.5.1 Algorithmus 24
3.6 JNEAT 27
3.6.1 Evolution/Experiment 27
3.6.2 Population 28
3.6.3 Network 29

3.6.4 Species . . . 29
3.7 Anwendung . . . 31

4 Design & Implementierung **33**
4.1 Aufgabe . . . 33
4.2 Modell . . . 34
4.3 Begriffe . . . 36
4.4 Simulationsumgebung . . . 37
4.4.1 Repast . . . 37
4.4.2 Bewertung . . . 40
4.5 MyOwnWorld . . . 41
4.5.1 Hauptklassen . . . 41
4.6 Agenten . . . 45
4.6.1 ObstacleAgent . . . 45
4.6.2 SimpleAgent . . . 45
4.6.3 VectorAgent . . . 45
4.6.4 OwnAgent . . . 45
4.6.5 NeatAgent . . . 46
4.6.6 SmartNeatAgent . . . 47

5 Experimente & Ergebnisse **49**
5.1 Einleitende Versuchsreihen . . . 52
5.2 OwnAgent . . . 54
5.2.1 Versuch Own-1500 . . . 54
5.3 NEAT-Versuche . . . 61
5.3.1 Vergleichsversuche . . . 62
5.3.2 SmartNEAT Versuche . . . 63
5.3.3 Abschließender Vergleich . . . 64
5.4 Fazit . . . 64

6 Schlussfolgerungen & Ausblick **67**
6.1 Rückblick . . . 67
6.2 Analyse . . . 68
6.3 Ausblick . . . 69
6.3.1 Modifikationen . . . 69
6.3.2 Erweiterungen . . . 70
6.3.3 Alternativen . . . 71

A Versuchslogbuch **73**
A.1 Erste Gehversuche . . . 74
A.2 Erste Versuchsreihen . . . 77
A.3 Versuchsreihe Statistiktests . . . 81
A.3.1 Versuch Nr. 14 . . . 81
A.3.2 Versuch Nr. 16 . . . 83

A.3.3 Versuch Nr. 16-02 83
A.4 Versuchsreihe 24 87
A.5 Versuchsreihe r02 87
A.5.1 Versuch r02-Neat-2-5000 87
A.5.2 Versuch r02-Neat-3-5000 88
A.6 Versuchsreihe r04 92
A.6.1 Versuch Nr. r04-Own-* 92
A.6.2 Ergebnis 92
A.6.3 Schlussfolgerung 93
A.6.4 Versuch Nr. r04-Vector* 93
A.7 Versuchsreihe 25 95
A.7.1 Versuch Nr. 25-Neat-simple-01 95
A.7.2 Versuch Nr. 25-Neat-simple-02 96
A.7.3 Versuch Nr. 25-Neat-simple-03 96
A.7.4 Versuch Nr. 25-Neat-simple-04 97
A.7.5 Versuch Nr. 25-Neat-simple-06 97
A.8 SimpleNeat 97
A.8.1 Versuch Nr. 25-SimpleNeat-01 97
A.8.2 Versuch Nr. 25-Neat-04-GRID 98
A.8.3 Versuch Nr. 25-Neat-04-GRID-randomEvaluate 98
A.8.4 Versuch Nr. 25-Neat-05 98
A.9 Versuchsreihe 26 101
A.9.1 Versuch Nr. 26-PrefixedNeat-01-c-test 101
A.9.2 Versuch Nr. 26-PrefixedNeat-01-c-test02 102
A.9.3 Versuch Nr. 26-PrefixedNeat-01-c-test03 102
A.9.4 Versuch Nr. 26-PrefixedNeat-01-c-test04 103
A.9.5 Versuch Nr. 26-PrefixedNeat-01-d bis -05 103
A.9.6 Fazit 104
A.10 Versuchsreihe 28 105
A.10.1 Versuch Nr. 28-SmartNeat-01 105
A.10.2 Versuch Nr. 28-SmartNeat-03 106
A.10.3 Versuch Nr. 28-SmartNeat-04 107
A.10.4 Versuch Nr. 28-SmartSimpleNeat-05 108
A.10.5 Fazit 108
A.11 Versuchsreihen r05 und r06 109
A.12 Versuchsreihe 29 110
A.12.1 Versuch Nr. 29-SmartNeat-01 110
A.12.2 Versuch Nr. 29-SmartNeat-02 110
A.12.3 Versuch Nr. 29-SmartSimpleNeat-01 111
A.12.4 Versuch Nr. 29-SmartSimpleNeat-02 111
A.12.5 Versuch Nr. 29-SmartSimpleNeat-03 112
A.12.6 Fazit 112

B CD-ROM **113**

C Softwareinstallation **115**
C.1 Bugfixes 115

Literaturverzeichnis **123**

Abbildungsverzeichnis

3.1 Ein Neuron . . . 20
3.2 künstliches neuronales Netz . . . 21
3.3 Nebeneinanderstellung verschiedener Schaltfunktionen . . . 22
3.4 NEAT Genom Kodierung (Quelle: [SBM05b]) . . . 25
3.5 NEAT Mutationsoperationen (Quelle: [SBM05b]) . . . 26
3.6 Klassendiagramm der wichtigsten JNEAT Klassen . . . 30

4.1 Design, Szenario . . . 35
4.2 Repast: Verschiedene Visualisierungen . . . 38
4.3 Klassendiagramm: MyOwnWorld . . . 42
4.4 Design: NeatAgent – Neuronales Netz . . . 47
4.5 Design: SmartNeatAgent – Prinzip . . . 48

5.1 Experiment 24: OwnAgent dirtMaps . . . 56
5.2 Experiment r02-Neat-2-5000: NEAT-Agent pathMaps . . . 57
5.3 Experiment 24: OwnAgent pheromoneMaps . . . 58
5.4 Experiment 24: OwnAgent Fitnessplot . . . 59
5.5 Experiment 24: OwnAgent Cellplot . . . 60
5.6 Experiment 24: OwnAgent Dirtplot . . . 61
5.7 Vergleich verschiedener Agenten . . . 62
5.8 Experiment 26-PrefixedNeat-01-c: FitnessPlot . . . 63
5.9 Experiment 29-SmartSimpleNeat-02-dirtplot: DirtPlot . . . 65

A.1 Testexperiment 1: Dirtmap . . . 74
A.2 Testexperiment 1: Pheromonemap . . . 75
A.3 Testexperiment 2: SimpleVectorAgent-100 Pheromondichte . . . 76
A.4 Testexperiment 1: SimpleAgent Pfade . . . 77
A.5 Testexperiment 1: Dirtplot . . . 78
A.6 Testexperiment 1: Fitnessplot . . . 79
A.7 Testexperiment 2: Fitnessplot . . . 79
A.8 Testexperiment 2: Dirtplot . . . 80
A.9 Testexperiment 2: Dirtplot-Vergleich . . . 80
A.10 Testexperiment 16/20: MeanFitnessplot . . . 84
A.11 Testexperiment 16-02-5: MeanFitnessplot . . . 86
A.12 Experiment r02-NEAT-2-5000 CleanCellsPlot . . . 88

A.13 Experiment r02-NEAT-2-5000: pathMaps 89
A.14 Experiment r02-NEAT-3-5000 CleanCellsPlot 90
A.15 Experiment r04-Own: MeanCleanCellsPlot 93
A.16 Experiment r04-Vector: MeanCleanCellsPlot 94
A.17 Versuch 25-Neat-04-GRID: Path-Map 98
A.18 Versuchsreihe 26: selbst entworfenes KNN 101
A.19 Experiment 26-PrefixedNeat-01-f CleanCellsPlot 104
A.20 Experiment 28-SmartNeat-04 : CleanCellsPlot 107
A.21 Versuch r05/ro6: MeanCellPlot 109

Tabellenverzeichnis

5.1 Simulationsparameter für Versuch 24-Own-1500 54
5.2 Von je 50 Agenten sauber gehaltene Zellen 66

A.1 Simulationsparameter für Versuch 14 81
A.2 NEAT-Parameter für Versuchsreihe *Statistiktests* 82
A.3 Simulationsparameter für Versuch 16 83
A.4 Simulationsparameter für Versuch 16-02-[a-d] 85
A.5 Simulationsparameter für Versuch r02-Neat-2-5000 87
A.6 NEAT-Parameter für Versuch 91
A.7 Simulationsparameter für die Versuche r04-Own-* 92
A.8 Simulationsparameter für Versuch 25-Neat-simple-01 95
A.9 NEAT-Parameter für Versuch 25-Neat-simple-01 100
A.10 Simulationsparameter für Versuch 28-SmartNeat-01 106

Kapitel 1

Einleitung & Motivation

Nichts in der Geschichte des Lebens ist beständiger als der Wandel.

Charles Darwin

Auf dem Gebiet der Automatisierungstechnik und Robotik wurden im letzten Drittel des 20. Jahrhunderts beträchtliche Fortschritte gemacht, wodurch ganze Industrien revolutioniert wurden. Niemals zuvor konnten so viele verschiedene Produkte so schnell und effizient hergestellt werden. Mit Beginn des 21. Jahrhunderts erweitert sich das Anwendungsgebiet der Robotik langsam aber stetig auch auf die Welt außerhalb der Fabrikhallen. Im Militär- und Sicherheitsbereich sind (teil-)autonome Systeme schon seit längerem im Einsatz, hier vor allem in der Aufklärung [Sch05c]. Seit kurzem sind nun auch erste autonome Kleinroboter für den Heimbereich verfügbar, zu ihren Aufgaben gehören vor allem bei Menschen eher unbeliebte oder lästige Arbeiten wie das Reinigen von Böden oder das Rasenmähen [Dr.08, Eic06].

Beim Schritt von der industriellen Anwendung hin zum Heimgebrauch stehen Forscher und Entwickler jedoch vor zwei großen Herausforderungen. Die Erste besteht darin, dass der Heimbereich um ein vielfaches dynamischer und komplexer als eine Fabrik mit ihren genormten Fertigungszellen und gleichbleibenden Fertigungsschritten ist. Zugleich muss der Einheitenpreis für den Privatnutzer um ein vielfaches niedriger liegen, als dies bei industriellen Anwendungen der Fall ist. Mit anderen Worten: Heimroboter müssen sich mit stark beschränkter Sensorik in komplizierten, oft unbekannten und sich verändernden Umgebungen zurechtfinden, um ihre Aufgaben erledigen zu können.

Es gibt sicherlich mehrere Auswege aus diesem Dilemma. Zum einen kann der Stückpreis durch Massenproduktion und Verwendung billiger Komponenten gesenkt werden, zum Anderen müssen neue Herangehensweisen an die Bearbeitung der Aufgaben entwickelt werden. Von der Vorstellung, dass ein Roboter eine ihm übertragene Aufgabe auf die gleiche Weise löst, wie ein Mensch dies tun würde, muss Abstand

genommen werden. Androiden à la *Robocop*[1] oder *C3PO*[2] erfreuen im Kinosaal zwar das Publikum, eine Umsetzung, zudem noch eine kostengünstige, ist in naher Zukunft jedoch nicht realistisch[3] [VDE]. Deshalb ist der Weg nicht der, Roboter zu entwickeln, die menschliche Werkzeuge (z.B. einen Staubsauger) benutzen können, sondern die Werkzeuge direkt zu automatisieren (z.B. autonome Staubsauger[4]).

Folgt man dem Ansatz der Bionik [BIO08] und sieht sich in der Natur nach Vorbildern um, die einen Lösungsansatz bieten könnten, so stößt man nach kurzer Suche u.a. auf Insektenvölker, genauer: Ameisen. Diese sind sehr einfach in ihrem Aufbau, haben eine sehr begrenzte Sensorik und können doch in ihrem Schwarm komplexe Leistungen vollbringen, unter anderem eben auch große Flächen nach bestimmten Dingen, meist Futter, absuchen und diese einsammeln. Dabei sind sie durch ihre gegenseitige und doch dezentrale Koordination sehr effizient. Um ähnlich geartete Probleme aus der technischen Welt zu lösen, wird versucht, dieses Verhalten möglichst nachzubilden und sich so dessen Vorzüge zu Nutze zu machen. Ein Beispiel hierfür ist der *Ameisenalgorithmus* [DCG99], mit dessen Hilfe verschiedene Optimierungsprobleme gelöst werden können. Das *Organic Computing* verfolgt ähnliche Ansätze wie die Bionik, fokussiert jedoch die Probleme der technischen Informatik. In Abschnitt 1.2 wird das Forschungsgebiet kurz vorgestellt und in Abschnitt 2.4.1 wird diese Arbeit darin eingegliedert.

Die Natur bietet jedoch noch einen weiteren Mechanismus, den man sich zu Nutze machen kann: die *Evolution*. Mit ihrer Hilfe entwickelt die Natur seit Milliarden von Jahren immer neue Lösungen für bestimmte Probleme, passt bestehende Lösungen an und optimiert sie so. Eine kurze Geschichte der Evolution wird in Abschnitt 1.1 dargestellt, in Abschnitt 2.2 werden ihre Mechanismen genauer erklärt. Sie ist jedoch (soweit bisher verstanden) ein ungerichteter Prozess und kann deshalb nicht ohne weiteres zur Lösung technischer Probleme herangezogen werden. Aus diesem Grund werden in dieser Arbeit Ansätze aus Bionik und Evolution kombiniert und auf das Problem der Koordination von kleinen Reinigungsrobotern für den zivilen Einsatz angewendet. Der gewählte evolutionäre Algorithmus *NEAT* [SBM05b] scheint für diese Aufgabe geeignet, da er mit evolutionären Methoden *Künstliche Neuronale Netze* (KNN) entwickelt. Um zu zeigen, dass dies effizient und effektiv möglich ist, entwickelten die Autoren das Computerspiel NERO [SM06, SBM05b, SBM05a]. Darin wird mit Hilfe von NEAT ein KNN entwickelt, welches dann zur Steuerung eines Schwarms von Roboterkriegern verwendet wird, die wiederum gegen andere Roboterkrieger kämpfen.

[1]Eine Figur aus dem gleichnamigen Film, der Polizeiaufgaben übertragen werden.

[2]Ein Sprachübersetzungsroboter aus der *Star Wars* Film-Reihe.

[3]Einer der am weitesten fortgeschrittenen humanoiden Roboter, den es heute gibt, ist das Robotermodell Asimo-P3 von Honda [HO07]. Es ist nur als Prototyp verfügbar und hat eine sehr begrenzte Sensorik und dementsprechend einfache Funktionalitäten, die über Laufen und Hände schütteln kaum hinausgehen. Abgesehen davon, dass er nicht käuflich zu erwerben ist, liegt der Stückpreis dieses Models jenseits von einer Million Dollar.

[4]Die Firma iRobot bietet unter www.irobot.com verschiedene „Staubsaugroboter“ zu Preisen ab 120 Dollar an, weitere verfügbare automatische Bodenreinigungssysteme werden unter anderem von den Firmen Kärcher (RC 3000), Electrolux (Trilobite) oder infinuvo (Cleanmate QQ2) angeboten.

Zunächst wird in dieser Arbeit eine Simulationsumgebung entwickelt. In dieser wird der NEAT-Algorithmus und seine Anwendung auf das Reinigungsproblem mittels Simulation evaluiert und die Ergebnisse mit einer vorgegebenen, sehr guten, Lösung verglichen. Dabei zeigt sich, mit welchen Schwierigkeiten es verbunden ist, den Algorithmus auf ein bestimmtes Problem anzupassen. Die Parametrierung ist so komplex, dass ein hinreichend gutes Lernverhalten im Rahmen dieser Arbeit nicht erreicht wird. Abschließend werden im Ausblick die Versuchsergebnisse diskutiert und ein Ausblick auf mögliche Weiterentwicklungen gegeben.

Will man sich die Evolution zu Nutze machen, so ist ein Grundverständnis ihres Funktionierens unerlässlich. Deshalb wird im Folgenden eine kurze Zusammenfassung ihrer Geschichte dargestellt. Danach wird erläutert, wie die Bionik und dabei besonders das Organic Computing in diesen Themenkomplex einzuordnen sind. Danach wird ein Überblick über die Gliederung dieser Arbeit gegeben.

1.1 Evolution & Entwicklung

Ob am Anfang die Ursuppe und ein Blitz (nach Miller/Urey [Mil53, MU59]) stand oder ob in einem sehr frühen Stadium der Erdgeschichte organische Verbindungen aus dem All auf die Erde gelangten (Panspermie-Hypothese, u.a. [Hor99, CO73]), ist noch immer Gegenstand wissenschaftlicher Debatten. Weitgehende Einigkeit herrscht hingegen darüber, dass seit dem Proterozoikum vor ca. 3.700 Millionen Jahren einzellige Organismen auf der Erde leben.

Dabei wird unter dem Begriff *Leben* ein Prozess verstanden, bei dem bestimmte (Protein-)Strukturen Substanzen umwandeln (Stoffwechsel) und sich selbstständig vervielfältigen (Reproduktion). Je effizienter, stabiler und schneller eine bestimmte Struktur bei diesem Reproduktionsprozess ist, umso häufiger existiert sie, sie ist *erfolgreicher*. Aus diesen Proteinstrukturen entwickelten sich die ersten (einzelligen) Organismen. Nach dem *trial-and-error* Prinzip entwickelten diese Strukturen durch Mutation immer komplexere und ausgefeiltere Strategien für den Stoffwechsel (vor allem Enzyme) und zum Selbstschutz (z.B. die Zellwände). Dabei stellte sich heraus, dass unter verschiedenen Umgebungsbedingungen verschiedene Strategien erfolgreich waren und die jeweils erfolgreichsten setzten sich über die Zeit durch. Aus diesen verschiedenen Strategien entwickelten sich verschiedene Arten, auch Spezies genannt, die fortan in Konkurrenz zueinander um die zur Verfügung stehenden Umgebungsressourcen konkurrieren.

Von da an dauerte es weitere ca. 2.700-3.000 Millionen Jahre bis der Evolutionsprozess, mit Beginn des Kambriums, komplexere (mehrzellige) Lebewesen hervorbrachte. Eine große Vielzahl von Lebewesen entstanden in einer, geologisch gesehen, kurzen Zeit, weshalb diese Phase in der Entwicklung des Lebens auf der Erde auch *Kambrische Explosion* genannt wird.

In den nun folgenden knapp 550 Millionen Jahren entwickelten sich immer neue Arten, die gegenseitig um die verfügbaren Ressourcen konkurrierten. Dabei komplexifizierte sich das Leben mehr und mehr und eroberte immer neue (ökologische)

Nischen. Der eigentliche Prozess blieb dabei jedoch immer der gleiche: Erfolgreiche Arten – und innerhalb dieser Arten erfolgreiche Individuen – vermehrten sich mit größerem Erfolg und verdrängten so ihre evolutionär unterlegene Konkurrenz. Im Laufe dieses Prozesses kristallisierten sich eine Vielzahl von Strategien heraus, um in diesem ewigen Kampf bestehen zu können. Manche dieser Strategien sind über lange Zeiträume erfolgreich (z.B. harte Schalen, die Muscheln und Schnecken vor Räubern schützen, erfolgreich seit über 500 Millionen Jahren), andere scheitern schon nach relativ kurzer Zeit.

Vor ca. 2,5 Millionen Jahren begann mit dem Auftauchen der ersten Vertreter der Gattung *Homo* die Erprobung einer neuen Strategie. Die Gattung Homo brachte mehrere Arten hervor. Der *Homo Sapiens* ist der aktuelle Vertreter und gleichzeitig auch die einzige überlebende Art dieser Gattung. Vor ca. 0,5 Millionen Jahren begann er, sich das Feuer zu Nutze zu machen, vor 0,01 Millionen Jahren bildeten sich höhere Kulturen heraus und seit gut 0,0001 Millionen Jahren beginnt er zu verstehen, wie dieser evolutionäre Prozess, aus dem er selbst hervorging, funktioniert. Eine entscheidende Rolle spielt dabei jene Strategie, welcher der Homo Sapiens seinen evolutionären Erfolg zu verdanken hat: Seine Intelligenz, ermöglicht durch eine Ansammlung von hochvernetzten Nervenzellen in seinem Kopf, dem Gehirn.

Diese vernetzten Nervenzellen werden *Neuronale Netze* genannt. In Abschnitt 3.4 wird darauf eingegangen, worum es sich dabei genau handelt, für den Moment soll es genügen, davon auszugehen, dass ein Neuronales Netz in der Lage ist, eingehende Nervenimpulse zu verarbeiten und daraus Handlungen abzuleiten, die es über Ausgabeneuronen ausgibt.

Diese Verarbeitung ist in biologischen Systemen über die (Millionen) Jahre zu einer Effizienz herangereift, die durch menschengemachte, technische Systeme bisher unerreicht ist. Insbesondere die Komplexität der Verarbeitung ist bis heute nicht vollständig verstanden. Im Zuge der Erforschung von neuronalen Netzen versucht der Mensch seit gut 60 Jahren diese nachzubilden (die ersten Modelle künstlicher Neuronen wurden von McCulloch et al. bereits 1943 vorgestellt [MP88]). Mit den immer weiter fortschreitenden technischen Möglichkeiten, insbesondere der Simulationsmöglichkeiten durch Computer, ist auch ein fortschreitender Erkenntnisgewinn verbunden. Heute werden künstliche neuronale Netzwerke auf vielfältige Weise vor allem in der Informatik eingesetzt, etwa bei der Sprach- und Mustererkennung [Wai90] oder bei der Steuerung von Robotern[5].

Um die genannten Zahlen noch einmal in Relation zueinander zu stellen: Die Zeit, seit der es organisches Leben auf der Erde gibt, entspricht 3.500 Millionen Jahren, davon gibt es den Menschen (Homo) seit 2,5 Millionen Jahren (weniger als ein Promille dieser Zeit, oder genauer $\frac{1}{1400}$). Kulturen bildet er seit knapp 10.000 Jahren ($\frac{1}{250}$ der Zeit, in der es die Gattung gibt und $\frac{1}{350.000}$ der Zeit, seit der es Leben gibt) und an der aktiven Erforschung des Gehirns arbeitet er seit weniger als 100 Jahren. Um noch weiter zu vergleichen: Ein Simulationslauf im Rahmen dieser

[5] Versuchsroboter wie z.B. Tekken oder BISAM verwenden rekurrente KNNs zur robusten Erzeugung von Bewegungsmustern für vierbeiniges Laufen [ALBD03, BID98].

Arbeit dauert ca. 1 Stunde, das entspricht einem Faktor von $1 : \frac{1}{3{,}066 \times 10^{13}}$.

1.2 Bionik & Organic Computing

Wie bereits dargelegt, entwickelte die Natur, repräsentiert durch die Evolution, eine ganze Reihe erfolgreicher Strategien und Mechanismen. Die Wissenschaft der *Bionik* beschäftigt sich damit, diese Strategien und Mechanismen zu identifizieren, zu abstrahieren und auf andere, meist technische Systeme, anzuwenden [BIO08].

Nicht zuletzt deshalb heißt es in [MSMW04] „Eine der folgenträchtigsten Entwicklungen der Informatik ist ihr Zusammenspiel mit der Biologie.“. Damit ist nicht nur der vermehrte Einsatz der Informatik in der Biologie gemeint, etwa bei der Sequenzierung des menschlichen Erbgutes. Auch der umgekehrte Weg ist erfolgsversprechend: die Anwendung biologischer Prinzipien in der Informatik.

Bereits heute haben viele technische Systeme einen Komplexitätsggrad erreicht, der es ihren Nutzern schwer macht, den Überblick und die Kontrolle zu behalten. Auch Entwickler und Wartungspersonal stehen mit zunehmender Komplexität vor immer größeren Herausforderungen. Da der Trend zu immer mehr und kleineren Systemen geht, die unsere Umgebung „intelligenter“ machen, ist absehbar, dass diese Komplexität eher noch weiter steigen wird. Deshalb ist es unabdingbar, dass neue Organisationsformen gefunden werden, um dieser Komplexität Herr zu werden.

Besonders im Hinblick darauf, dass die Systeme immer weiter in den menschlichen Alltag vordringen, müssen sie sich auch mehr an menschlichen Bedürfnissen orientieren.

In diesem Sinne kann das *Organic Computing* als Teilgebiet der Bionik aufgefasst werden. Es befasst sich damit, die in der Natur entdeckten Strategien der Selbstorganisation, Selbstkonfiguration, Selbstoptimierung und Selbstheilung auf technische Systeme, vornehmlich Computersysteme, anzuwenden. Dadurch soll vor allem eine bessere Wartbarkeit der Systeme sowie eine größere Robustheit und höhere Flexibilität erreicht werden [Sch05b].

Ein möglicher Weg zu diesem Ziel ist die Selbstorganisation nach dem Schwarmvorbild, bei dem viele einfache Komponenten eine komplexe Aufgabe in Kooperation lösen. Dabei ist die komplexe Lösung keine einfache Folge einfacher Aktionen, sie ist vielmehr (bisher) nicht vorhersagbar. Dieses „emergent“ genannte Verhalten kann, aufgrund seiner Nichtvorhersagbarkeit sowohl positiv als auch negativ sein. Da eine Vorhersage nicht möglich ist, jedoch andererseits ein technisches System mit unbekanntem Verhalten auch nicht erstrebenswert ist, bedarf es anderer Herangehensweisen, um es zu kontrollieren und zu nutzen.

„Observer/Controller“ Architekturen stellen eine Möglichkeit dar, Systeme mit emergentem Verhalten nutzen zu können ohne dabei die Kontrolle über ihre Effekte aufgeben zu müssen. Sie befähigen den Benutzer ein System zu beobachten, zu analysieren und – wenn nötig – geeignete Maßnahmen zu treffen [Sch05a]. Diese Kontrolle ist vor allem auch für die tatsächliche Anwendung unbedingt notwendig, da ein nicht vorhersagbares System auch neuartige Fragen z.B. hinsichtlich der Verantwortung

für komplexe Folgen oder der Vertrauenswürdigkeit solcher Systeme aufwirft.

Das Organic Computing ist eine sehr junge Disziplin die vor einer wichtigen und weitläufigen Problemstellung steht. Nicht nur deshalb hat sie viele Überschneidungspunkte zu anderen Disziplinen wie z.B. der künstlichen Intelligenz.

1.3 Aufbau dieser Arbeit

Im 2. Kapitel werden zunächst einige Grundbegriffe erläutert und Mechanismen vorgestellt, die zum weiteren Verständnis der Arbeit notwendig sind. Anschließend wird die Aufgabenstellung erklärt und in das Forschungsgebiet des Organic Computing eingeordnet. Abschließend wird der Lösungsansatz skizziert. Kapitel 3 gibt einen Überblick über bisher auf diesem Gebiet geleistete Arbeiten und grenzt dadurch den Lösungsansatz weiter ein. Das zur letztendlichen Lösung entwickelte Modell wird in Kapitel 4 vorgestellt. Kapitel 5 gibt einen Überblick über die durchgeführten Experimente und deren Ergebnisse. In Kapitel 6 werden diese noch einmal zusammengefasst und die daraus zu ziehenden Schlussfolgerungen genannt. Außerdem wird ein Ausblick auf mögliche Weiterentwicklungen gegeben. Im Anhang findet sich neben einigen Anmerkungen zu den genutzten und getesteten Softwarekomponenten und den ausführlicheren Versuchsprotokollen auch eine CD-ROM. Auf ihr finden sich die entwickelten Quellcodes, ein Großteil der verwendeten Literatur (als .pdf-Dateien), sowie eine Auswahl aus den Daten, die im Laufe der Experimente angefallen sind.

Kapitel 2

Einführung & Problemstellung

Wer ein Problem definiert, hat es schon halb gelöst.

Julian Huxley

Evolutionäre Prozesse spielen in dieser Arbeit eine große Rolle. Da es sich dabei um den Versuch einer Nachbildung biologischer Prozesse handelt, vor allem aus dem Teilgebiet der Genetik, werden hier zunächst einige Kernbegriffe erläutert. Dies dient dem Leser dazu, ein Grundverständnis der Vorgänge zu erlangen und ihm zu ermöglichen, die beschriebenen Verfahren nachvollziehen zu können. Die Beschreibungen sind daher stark vereinfacht. Eine detaillierte biologische Beschreibung ist nicht Ziel dieser Arbeit und würde den gegebenen Rahmen sprengen. Sie kann unter anderem in [CR06] gefunden werden.

Anschließend wird das biologische Vorbild der evolutionären Prozesse vorgestellt. Dabei wird vor allem auf die einzelnen Mechanismen eingegangen, aus denen die Evolution besteht. Sie sind für das spätere Verständnis des gewählten Algorithmus wichtig. Da im weiteren Verlauf der Arbeit der Begriff „Agent“ häufig Verwendung findet, wird er kurz eingegrenzt. Abschließend wird auf das Problem eingegangen, welches diese Arbeit motiviert. Es wird in das Umfeld des Organic Computing eingeordnet und die Herangehensweise dargelegt.

2.1 Genetik

Die folgende Aufzählung erläutert einige Begriffe der Genetik. Dabei besteht nicht der Anspruch einer allgemeingültigen Definition, vielmehr sollen die Begriffe voneinander unterschieden werden.

DNS/DNA DNS ist die Abkürzung für Desoxyribonukleinsäure (DNA ist der entsprechende Begriff im Englischen mit dem A für Acid anstelle von S für Säure). Die DNS ist ein (sehr komplexes) Molekül, das in biologischen Zellen als Träger der Erbinformation dient.

Gen Ein Gen ist ein Abschnitt der DNS und repräsentiert die kleinste kodierte Informationseinheit. Sie besteht aus einer variablen Anzahl von Basenpaaren, die in ihrer Reihenfolge eine bestimmte Information kodieren. Ein Gen kann sowohl funktionale (Produktion bestimmter Enzyme) als auch strukturelle (Aufbau des Körpers, Zellteilung etc.) Informationen darstellen.

Genom Als Genom wird die Gesamtheit aller Gene eines Organismus bezeichnet.

Genotyp Die in einem Genom (manchmal auch einem einzelnen Gen) enthaltenen Informationen werden als Genotyp bezeichnet.

Phänotyp Der Phänotyp bezeichnet die Ausprägung eines Organismus (manchmal auch einzelner Zellen, Strukturen oder Stoffe), die sich aus einem Genotypen entwickelt. Die letztendliche Ausprägung wird dabei nicht allein durch Erbinformationen bestimmt, da die Entwicklung auch von Umwelteinflüssen abhängt.

Crossover Als Crossover wird die Operation der genetischen Vermischung bezeichnet, die bei der mehrgeschlechtlichen Reproduktion entsteht.

Mutation Die Mutation stellt eine zufällige, nicht zielgerichtete Veränderung der Erbinformation dar. Sie tritt (nicht ausschließlich) während der Reproduktion auf und ist ein Hauptquell für genetische Vielfalt. Da die Änderungen rein zufällig sind, haben sie in den meisten Fällen negative Auswirkungen auf den resultierenden Phänotyp. Dennoch ist die Mutation die Quelle für alle größeren evolutionären Innovationen.

Es sei an dieser Stelle angemerkt, dass in der Biologie Gene nicht direkt mit einer kodierten Information gleichgesetzt werden können. So kommt es sehr oft vor, dass einzelne Informationen redundant gespeichert sind (teilweise in millionenfacher Ausführung), in anderen Fällen enthält das Genom Gene, die inaktiv sind und lediglich weitervererbt werden, ohne eine tatsächliche Funktion zu erfüllen. Auch kann die Länge der einzelnen Gene stark variieren. Ein sehr langer DNS-Strang muss also nicht notwendigerweise sehr viele Gene enthalten.

All diese Eigenschaften ergeben sich auf die ein oder andere Art aus der Entstehungsgeschichte der einzelnen Genome bzw. ihrer Träger.

2.2 Evolution

In Abschnitt 1.1 wurde bereits ein kurzer Abriss der Geschichte der Evolution auf der Erde gegeben. Der prägnante Satz „Survival of the fittest“ (zu Deutsch etwa „Das Überleben des Stärksten“) des Entdeckers der Evolutionstheorie *Charles Darwin* dürfte jedem Schulkind bekannt sein. Die Mechanismen, auf denen diese Theorie aufbaut, werden z.B. in dem Biologie-Lehrbuch von Campbell et al. [CR06] sehr genau beschrieben, die folgenden Abschnitte (2.2.1 - 2.2.4) fassen sie grob zusammen.

2.2.1 Arten

Die verschiedenen bekannten Lebewesen werden nach *Arten* (auch *Spezies*) voneinander unterschieden. Die Zuordnung einzelner Lebewesen kann nach unterschiedlichen Kriterien geschehen, man unterscheidet hier verschiedene *Artkonzepte*.

Ein einfaches und ursprüngliches Artkonzept ist das morphologische Artkonzept. Hier erfolgt die Zuordnung aufgrund von *äußeren Unterscheidungsmerkmalen*. So ist z.B. ein Pferd von einem Zebra an der Musterung der Fells unterscheidbar. Dieses Artkonzept hat jedoch einige Schwachstellen. In Insektenstaaten z.B. haben die verschiedenen Ausprägungen (Kasten) ein sehr unterschiedliches Aussehen, Königin und Arbeiter müssten folglich (aufgrund ihrer verschiedenen Morphologien/ihrem Aussehen) verschiedenen Arten angehören.

Eine andere Klassifizierung liefert das biologische Artkonzept. Hierbei werden Lebewesen derselben Art zugeordnet, wenn sie unter natürlichen Bedingungen fruchtbare Nachkommen erzeugen können. Auch dieses Modell hat einige Schwachstellen. So müssten geographisch voneinander getrennte Populationen verschiedenen Arten zugeordnet werden, da sie sich unter natürlichen Bedingungen nicht kreuzen können. Auch werden unfruchtbare Individuen, wie es beipielsweise die Arbeiter eines Insektenstaates sind, von dieser Beschreibung nicht abgedeckt.

Die für diese Arbeit interessanteste Klassifizierung ist das populationsgenetische (genealogische) Artkonzept. Es unterscheidet Populationen, deren Genpools gegenüber anderen Arten generativ isoliert sind. Es werden also Gruppen von Organismen unterschieden, die eine einzigartige genetische Geschichte haben und sich so von anderen Gruppen von Organismen hinsichtlich einzigartiger genetischer Marker unterscheiden. Dieses Verfahren ist in der Biologie oft sehr umständlich, da vor jeder Zuordnung der Genpool geprüft werden muss. Im Rahmen dieser Arbeit ist das Genom jedes betrachteten virtuellen „Lebewesens“ jedoch bekannt, die Zuordnung kann also auf diese Art erfolgen.

2.2.2 Konkurrenz

Nach der Evolutionstheorie stehen die einzelnen Arten in einem permanenten Wettbewerb miteinander. Sie konkurrieren um die naturgegebenen Ressourcen ihres Lebensraums, vor allem um Futter. In diesem Wettstreit kommt es zu immer weiter fortschreitenden Spezialisierungen der Strategien und inzugedessen auch zur Bildung immer neuer Spezies. Ein sehr einfaches Beispiel ist die Herausbildung von Pflanzen- bzw. Fleischfressern (lateinisch Herbi- bzw. Carnivoren). Aufgrund dieser verschiedenen Strategien stehen nicht alle Arten in direktem Wettstreit um die gleiche Ressource.

Stehen zwei Arten im Wettbewerb um die gleiche Ressource, so verdrängt im Normalfall die Art mit der erfolgreicheren Strategie ihre Konkurrenz. Diese stirbt entweder aus oder spezialisiert sich auf eine alternative Ressource. Ein Beispiel hierfür stellen z.B. Löwen und Hyänen dar. Beide sind Carnivoren, jedoch sind Löwen bei der Jagd erfolgreicher. Hyänen sterben deshalb jedoch nicht automatisch aus, sie

sind darauf spezialisiert, sich von Aas zu ernähren.

Neben dem Wettbewerb unter den Arten spielt der Wettbewerb innerhalb einer Art eine ebenso große Rolle im Evolutionsprozess. Hierbei stehen die Mitglieder einer Population in direktem Wettbewerb untereinander. Erfolgreichere Individuen obsiegen dabei über ihre Artverwandten. Dies begründet sich zum Beispiel darin, dass sie gesünder und kräftiger sind.

2.2.3 Reproduktion

Eine Grundeigenschaft von Lebewesen ist die Reproduktion. Sie findet in der Natur, mit einigen Ausnahmen, zweigeschlechtlich statt. Dabei werden die Erbinformationen (Genome) der beiden Eltern nicht einfach addiert, sondern vermischt, so dass das Genom der Kinder sich in der Größe nicht wesentlich von dem seiner Eltern unterscheidet. Die Kinder erben also Gene beider Eltern. Bei dieser Vermischung kommen zwei wichtige Mechanismen der Evolution zum Tragen: Die Kreuzung und die Mutation.

Kreuzung bezeichnet den Prozess der Aufteilung der einzelnen Elterngene auf das Kind. Welches Elternteil dabei welche Gene auf die Kinder vererbt, hängt in der Natur von verschiedenen Faktoren ab. Im Ganzen erhalten die Kinder ein Genom, welches sich von jedem der beiden Eltern unterscheidet.

Im Zuge des Reproduktionsprozesses kommt es auch immer wieder zu Mutationen. Sie stellen eine zufällige Veränderung des Erbgutes dar. So können Kinder Gene erhalten, die keiner ihrer Eltern besaß. Solche Mutationen haben in der Regel sehr geringe Auswirkungen auf die Nachkommen. In der Natur verändern Mutationen nur einzelne Gene, in der Regel verlieren diese daraufhin ihre Funktion. In seltenen Fällen erfüllen sie danach jedoch eine neue Funktion, die dem Individuum eine bestimmte Eigenschaft verleiht. Obwohl sie auf das Individuum nur einen so geringen Einfluss haben, stellen Mutationen die Hauptquelle von Innovationen im Evolutionsprozess dar.

Jedes Individuum besitzt also eine ganze Reihe von Eigenschaften, die seinen evolutionären Erfolg maßgeblich beeinflussen.

2.2.4 Selektion

Der wohl wichtigste Mechanismus der Evolution ist die Selektion. Eine Art ist erfolgreich, wenn sie sich erfolgreich reproduzieren kann. Dies lässt sich direkt auf einzelne Individuen übertragen.

Je erfolgreicher ein Individuum bei der Reproduktion ist, umso mehr Nachkommen hat es. Sie sind natürlicherweise ihren Erzeugern sehr ähnlich. Somit ist die Wahrscheinlichkeit, dass zwei erfolgreiche Eltern erfolgreiche Nachkommen erzeugen, relativ hoch. Diese können sich wiederum gegen weniger erfolgreiche Konkurrenten durchsetzen.

Als Resultat setzt sich mit der Zeit innerhalb einer Art ein erfolgreiches Genom gegenüber einem weniger erfolgreichen durch und verdrängt dieses auf eine ähnliche

Weise, wie eine erfolgreiche Art eine weniger Erfolgreiche verdrängt. Gerade bei der Verdrängung spielt der Evolutionsdruck eine entscheidende Rolle. Darunter versteht man die Summe aller Faktoren, die einen Einfluss auf das Überleben einzelner Lebewesen haben. Je ungünstiger diese Faktoren sind, umso schwieriger ist es, für einzelne Individuen zu überleben und sich erfolgreich zu reproduzieren. Ein hoher Evolutionsdruck führt deshalb oft zu Veränderungen der Spezies, da besser angepasste Individuen ihre weniger erfolgreichen Artverwandten verdrängen und so der Spezies eine leicht veränderte, angepasstere Form geben.

2.2.5 Genese

Das Wort Genese (auch Genesis) leitet sich aus dem Griechischen ab und bedeutet soviel wie „Geburt, Schöpfung, Entstehung“. In verschiedenen Wissenschaften wird dieser Begriff genauer unterschieden, in diesem Zuge wird er mit erklärenden Präfixen versehen. Einige davon kommen in dieser Arbeit zur Sprache und werden hier kurz erläutert.

Ontogenese bezeichnet die Entwicklung eines Organismus aus einem Genom. Dabei nehmen Umweltbedingungen Einfluss auf diese Entwicklung. Aus dem gleichen Genom können sich so in verschiedenen Umgebungen verschiedene Formen entwickeln.

Epigenese bezeichnet den Lernprozess eines Organismus. Dieser kann von der ontogenetischen Ausprägung abhängig sein.

Phylogenese beschreibt nicht die Entwicklung eines einzelnen Organismus, sondern die Entwicklung von Arten im genetischen Auswahlprozess.

Im Evolutionsprozess gibt es also keine direkten Rückwirkungen von Ontogenese und Epigenese auf das Genom. Mögliche indirekte Rückwirkungen bestehen lediglich in Selektionsfaktoren, die im Zuge von Ontogenese und Epigenese gewonnen werden.

2.3 Agenten

Da im Folgenden oft von *Agenten* gesprochen wird, soll auch dieser Begriff kurz näher beleuchtet werden. Er leitet sich aus dem lateinischen Wort *agere*, zu Deutsch *handeln*, *darstellen*, ab und lässt sich als *Handelnder* übersetzen. Das Wort „Agent“ wird im Deutschen für eine Vielzahl von Bereichen verwendet, es kann ebenso den Vermittler für Schauspieler bezeichnen, wie den Geheimagenten, der als verdeckter Aussendienstmitarbeiter für einen Staat tätig ist. Allen Agenten ist eines gemeinsam: Sie führen Handlungen für jemand anderen aus.

Im vorliegenden Kontext ist vor allem der *Softwareagent* von Interesse. Auch dieser Begriff ist sehr weit gefasst, in dieser Arbeit ist damit ein Programm gemeint, welches, in gewissen Grenzen, selbstständig ihm übertragene Aufgaben erledigen kann.

2.4 Problemstellung

Nachdem die für später relevanten Begriffe definiert sind, wird nun die Problemstellung erörtert. Eine Gruppe von Robotern mit einer Aufgabe zu betrauen, die sie in einer unbekannten Umgebung lösen sollen, ist ein schwieriges Unterfangen. Wenn die Roboter dann auch noch ohne eine zentrale Koordinierung und mit begrenzter Sensorik auskommen müssen, so sind bisher keine vollständigen Lösungen bekannt, die ohne Weiteres einsetzbar sind. Biologisch motivierte Ansätze scheinen hier noch am erfolgsversprechendsten.

2.4.1 Einordnung in das Organic Computing

Wie gut verträgt sich jedoch die Idee einer komplexen Strategie mit dem Ansatz von sehr primitiven Agenten?

Valentin Braitenberg zeigt in [Bra93], dass (scheinbar) komplexes Verhalten nicht immer von komplexen Regeln erzeugt werden muss. Dazu entwirft er eine Reihe von Vehikeln, denen er über sehr einfache analoge Regler bestimmte Verhaltensmuster gibt, z.B. Vermeidung von Licht oder Anziehung durch Licht. Das Verhalten, welches sich aus der einfachen Kombination dieser Regler ergibt, kann bereits so komplex sein, dass ein Beobachter Mühe hat, es zu verstehen oder gar vorauszusagen, man kann es also *emergent* nennen.

Ein anderes Beispiel liefern McPartland et al. in [MNA05]. Sie führen ein Experiment durch, bei dem ein KNN aus nur 41 Nervenzellen eine Gruppe von fünf Agenten steuert. Die Aufgabe besteht darin, in Konkurrenz zu einem zweiten Team eine Karte zu erkunden und dabei der jeweils letzte Besucher einer Kartenzelle zu sein. Das zweite Team wird von einer fest programmierten Strategie gesteuert, während das KNN-Team seine Strategie dynamisch lernen kann. In fünf Experimenten geht es u.a. darum, die größte Fläche zu erkunden und das gegnerische Team daran zu hindern bzw. darin zu unterstützen. Es zeigt sich, dass trotz der geringen Anzahl an Neuronen Strategien entwickelt werden, die eine Aufgabenverteilung und die Ablenkung der gegnerischen Teams beinhalten.

Während das KNN bei McPartland noch mit „klassischen“ Verfahren lernt (Back-Propagation), motiviert Braitenberg in [Bra93] einen *evolutionären* Ansatz, den er in [Bra94] noch weiter erläutert. Danach fallen diejenigen Vehikel von einem gedachten Experimentiertisch, die die Tischkante nicht meiden. Die auf dem Tisch verbleibenden Vehikel werden kopiert. Bei der Kopie der Vehikel entstehen nun Ungenauigkeiten und Fehler[1] die über lange Zeit dazu führen, dass sich Vehikel herausbilden, die die Tischkante „instinktiv“ meiden.

In [SBM05b] wird der Ansatz der Steuerung eines Agenten durch ein KNN mit einer evolutionären Entwicklungsweise kombiniert. Eine genaue Beschreibung dieses Verfahrens findet sich in Abschnitt 3.5.

[1]Da es sich um analoge Regler handelt, wirken sich Ungenauigkeiten beim Kopieren schneller und direkter aus als dies bei Digitalen der Fall wäre.

Auch aus der Natur ist derartig emergentes Verhalten bekannt. So fliegen viele Vogelschwärme in einer charakteristischen V- oder L-Formation. Dies minimiert den Energieverbrauch des Gesamtschwarms [WMC+01]. Dabei plant kein einzelner Vogel die Formation oder berechnet den Energieverbrauch für das Fliegen im Windschatten, sondern die Vögel reagieren instinktiv auf die Situation, so wie sie diese wahrnehmen.

Dieses Verhalten ist nicht nur emergent, es ist auch in gewisser Weise selbstheilend. Sobald z.B. der Führungsvogel (für den es keine Energieersparnis gibt) ermüdet, fällt er zurück und ein anderer Vogel übernimmt automatisch die Führung, ohne dass dies abgesprochen ist.

Wenn es gelänge, ein derartiges Verhalten in einer Simulation nachzubilden, dann wäre man in der Lage ein Problem so zu modellieren und zu kontrollieren, dass sich aus den einfachen Komponenten ein komplexes Verhalten bildete. Man wäre dann ebenfalls in der Lage, Emergenz nach Wunsch herbeizuführen. Mit anderen Worten: Der Lösung des Problems der Nichtvorhersagbarkeit wäre man einen Schritt näher.

Es wäre dann beispielsweise möglich, einen Satz einfacher Regeln zu entwickeln, die, z.B. in eine PKW-Steuerung eingebaut, in ihrer Gesamtheit dazu beitrügen, Staus auf Autobahnen zu vermeiden. Dies ist Grund genug, sich mit der Modellierung von Schwarmverhalten näher auseinanderzusetzen.

2.4.2 Szenario

Die Grundmotivation dieser Arbeit liegt darin, einen „Schwarm“ von Reinigungsrobotern mit Techniken des *Organic Computing* dazu zu bringen, eine gegebene Fläche mit Hindernissen (z.B. ein Büro) möglichst effizient zu reinigen.

Dabei sollen die einzelnen Roboter nur über eine sehr begrenzte Wahrnehmung ihrer Umgebung verfügen und trotzdem ohne eine zentrale, planende Instanz auskommen. Zusätzlich sollen sie auch ohne Vorwissen oder eine vorgegebene Strategie auf die Aufgabe angesetzt werden, sie sollen also selbst lernen, auf welche Weise sie die Aufgabe lösen können. Außerdem sollen die Einzelroboter – nach Möglichkeit – ihr gelerntes Wissen an andere Roboter des Schwarms weitergeben können.

Die Aufgabe besteht also vor allem im gleichmäßigen Abdecken einer Fläche. Dies ist jedoch eine Aufgabe, die mit steigender Zahl von Akteuren höchstens linear skalierbar ist, zwei Roboter können die Aufgabe also höchstens doppelt so schnell lösen wie ein einzelner Roboter. Von daher stellt die Aufgabe nicht unbedingt die beste Demonstrationsmöglichkeit für emergentes Verhalten dar. Allerdings ist schon das Lernen der effizienten Koordinierung der einzelnen Schwarmroboter eine nicht triviale Aufgabe.

Außerdem kann in einem späteren Schritt eine Lösung problemlos durch weitere Randbedingungen erweitert werden[2], so dass man die gestellte Aufgabe als ersten Schritt betrachten kann.

[2]Eine Möglichkeit hierzu wäre z.B. die Einführung eines Putzgerätes, das von zwei Robotern gleichzeitig bedient werden muss, dafür aber soviel Schmutz aufsammelt, wie drei Roboter auf die herkömmliche Art.

Als Qualitätskriterien für eine erfolgreiche Reinigung können verschiedene Werte angesetzt werden. Zum einen ist es von Interesse, ob eine entwickelte Strategie tatsächlich jeden Punkt der Karte abdeckt. Zum anderen ist die Gleichmäßigkeit der Abdeckung von Interesse, werden Punkte mehrfach besucht, so sollten alle ungefähr gleich häufig angefahren werden und die Zeit zwischen den einzelnen Besuchen sollte ebenfalls gleichmäßig sein. Schließlich ist auch die Aufteilung der geleisteten Arbeit auf die einzelnen Agenten von Interesse.

In [Gor02] wird beschrieben, wie eine Ameisenkolonie ein ähnliches Problem angeht. Dabei teilen Ameisen sich die Aufgabe untereinander auf. Eine Gruppe ist für das Auffinden von aufzusammelnden Ressourcen zuständig, während eine zweite Gruppe diese abtransportiert. Eine derartige Aufteilung in zwei unterschiedliche Arten von Agenten soll in diesem Schritt noch nicht vorgenommen werden.

Nachdem ein geeigneter Algorithmus gefunden ist, wird er in einer Simulation angewendet und das Ergebnis in verschiedenen Experimenten überprüft. So kann seine Eignung zur Lösung auch anderer Probleme bewertet werden.

Kapitel 3

Stand der Technik

The truth is that the most expensive weapon that technology can produce is worth not an iota more than the skill and will of the man who uses it.

General Bruce C. Clarke, 1959

Diese Arbeit befasst sich mit einem Thema, das bereits seit längerem Gegenstand intensiver Forschungsarbeit ist. Ausgehend von Arbeiten im Bereich der *Künstlichen Intelligenz* (KI) entwickelten sich mehrere Vertiefungsgebiete, die sich mit der Zeit immer weiter spezialisierten und auseinander entwickelten. So kommt es, dass in dieser Arbeit Werke von zum Teil sehr unterschiedlichen Disziplinen zitiert bzw. referenziert werden.

3.1 Schwärme und Pheromone

Ein Ausgangspunkt für diese Arbeit ist die Arbeit von Wagner et al. [WLB96]. Darin entwickeln die Autoren drei Algorithmen, die simulierten Ameisenrobotern unter Zuhilfenahme von Pheromonspuren dazu bringt, eine Fläche abzudecken. Dabei liegt der Schwerpunkt darauf, dass jeder Punkt auf einer (unbekannten) Karte mindestens einmal angefahren wird. Die Autoren erwähnen die mögliche Anwendung ihres Verfahrens zur Reinigung von Flächen, betonen jedoch, dass in einem solchen Fall die Strategie dahingehend erweitert werden sollte, dass sie die Gleichmäßigkeit der Flächenabdeckung in den Vordergrund stellt. Der pheromonbasierte Ansatz für die Abdeckung einer Fläche ist jedoch praktikabel.

Zeng et al. stellen in [ZHB07] eine Studie über pheromonbasierte Schwarmalgorithmen vor, die 3D-Konstruktionen errichten. Damit sollen realistischere Computergegner in Computerspielen entwickelt werden können.

Die Koordination von Bienen wird in [AFB05] untersucht. Bienen sind in der Lage, die Position und Richtung einer Futterquelle ebenso zu kommunizieren wie deren Ergiebigkeit. Diese Kommunikation erfolgt in Form eines „Tanzes“. Unbeschäftigte

Bienen halten sich im Bienenstock auf und beobachten Bienen, die von der Futtersuche zurückkehren und Informationen über Futterquellen mitbringen. Diese „Kundschafter“ bewegen sich dann wiederholt rhythmisch in einer Form, die den wartenden Bienen Rückschlüsse auf die Position der Futterquelle erlaubt. Sobald sie diese Position verinnerlicht haben, schließen sie sich der tanzenden Biene an. Sind der Futterquelle angemessen viele Bienen versammelt, brechen sie zur „Ernte“ auf. So wird über die Länge des Tanzes die Zahl der Arbeiter geregelt, die auf eine Futterquelle angesetzt werden.

Es gibt in der Natur also mindestens zwei verschiedene Strategien, Schwärme mit Hilfe einer relativ einfachen Kommunikation zu koordinieren. Sie unterscheiden sich darin, dass die Pheromonspuren der Ameisen direkt in der Landschaft angebracht sind, während die Bienen nur direkt kommunizieren.

In [BDAM04] berichten Bruemmer et al. über ihre Fortschritte bei der Anwendung eines realen Roboterschwarms auf echte Probleme. Grundlage ihrer Arbeit sind vier Hauptcharakteristika von Schwarmverhalten. Dies sind nach Tsetlin [Tse73] Zufälligkeit, Dezentralisierung, indirekte Interaktion und Selbstorganisation. Richtigerweise kommt hier auch die Frage der Benutzbarkeit eines solchen Systems auf.

Wenn man von Schwarmverhalten fordert, dass es adaptiv, selbstregulierend und nichtdeterministisch ist, dann kann es nicht überraschen, dass das Verhalten des resultierenden Systems schwer vorhersagbar ist. Ebenso ist es schwierig, für solche Systeme Garantien bezüglich ihrer Leistung oder Trainingszeiten zu geben. Die Autoren folgern daraus, dass der Schwarmansatz kein Allheilmittel ist, wenn es darum geht, eine Vielzahl von Robotern zu koordinieren.

Eine interessante Erkenntnis der Arbeit ist, dass die richtige Anzahl von Robotern einen großen Einfluss auf die Effizienz sowohl der einzelnen Roboter als auch des ganzen Schwarms hat. Zu wenige Einheiten führen hier ebenso zu suboptimaler Leistung wie eine zu hohe Zahl und damit eine zu hohe Einheitendichte.

Die Autoren kommen zu mehreren Schlussfolgerungen. Zum einen stellen sie fest, dass eine künstliche Schwarmintelligenz, die sich an biologischen Insekten orientiert sehr wahrscheinlich auch bestimmten Limitierungen hinsichtlich ihrer kurzfristigen Anpassungsfähigkeit unterworfen ist und dass diese Beschränkungen vermutlich proportional zum gewählten Analogietätsgrad sind. Außerdem kommen sie zu dem Schluss, dass eine effiziente Anwendung von Schwarmrobotern vor allem voraussetzt, dass kostengünstige und energieeffiziente Sensorkomponenten verfügbar sind.

Abschließend stellen sie fest, dass eine Anwendung vor allem ein robustes Verhalten der Schwarmroboter voraussetzt und dass dieses mit einer Benutzerschnittstelle ausgestattet sein muss, welches menschlichen Benutzern eine Interaktions- bzw. Steuermöglichkeit gibt.

In [Mar04] wird vor allem die biologische Evolution von sozialen Insekten beschrieben und zusammengefasst.

Die Autoren stellen dabei fest, dass Parameter wie Schwarmgröße, Größenverhältnisse der einzelnen „Kasten“ [1] u.a. in bestimmten Parametern der Schwarmmit-

[1]Als Kasten werden hier Gruppen gleicher Ausprägungen bezeichnet, z.B. die Arbeiterkaste,

glieder verankert ist.

Solche Parameter in einer Simulation von außen vorzugeben könnte folglich einen Einfluss auf das Ergebnis der simulierten Evolution haben. Der Versuch, diese Parameter ebenfalls in die Evolution einfließen zu lassen bzw. evolutionär zu bestimmen scheint deshalb angebracht.

Des weiteren werden einige interessante, mögliche Experimente, vor allem mit Futter suchenden Bienen, beschrieben. Voraussetzung hierfür ist eine hinreichend genaue Nachbildung der biologischen Vermehrung inklusive genetischer Verwandtschaft, Futtersuche und Verbrauch und eines realistischen Modells der Fortpflanzung.

So könnten bei gleichem Genotyp mehrere Phänotypen entstehen, die durch äußere Faktoren bei der Erzeugung beeinflusst werden. So könnten verschiedene Individuen zwar einen gemeinsamen Gencode haben, jedoch verschiedene Spezialisierungen hervorbringen und so die Entstehung von verschiedenen Kasten unterstützen. Denkbar wäre hier z.B. die Entwicklung einer Krieger-Ausprägung, falls die Eltern gut mit Nahrung versorgt sind, andernfalls würde sich aus der gleichen Elternkombination eine Arbeiter-Einheit entwickeln.

In [Lee06] wird folgerichtig erkannt, dass ein System, welches eine allgemeingültige Strategie entwickeln soll, ein agentenrelatives Koordinatensystem benutzen muss. Jede absolute Kodierung von z.B. Bewegungsbefehlen würde (mit hoher Wahrscheinlichkeit) zu einer Strategie führen, deren Leistung von der Ausgangssituation des Agenten abhängig ist. Somit verbietet sich eine Kodierung in absolute „Himmelsrichtungen“ (Nord, Süd, ...). Im Gegenteil ist es erforderlich, von den Agenten relative Richtungsangaben zu verlangen (Vorne, hinten, rechts, links). Selbstverständlich müssen dann auch sämtliche Eingabedaten relativ zum Agenten und seiner aktuellen Orientierung kodiert sein. Dies setzt jedoch voraus, dass der Agent überhaupt ein „Vorne“ von einem „Hinten“ unterscheiden kann.

Besitzt ein Agent keine eigentliche Orientierungsrichtung, so stellt sich die Frage, ob diese Annahme ebenfalls gültig ist. Ebenfalls fraglich ist, ob einen eingeführte Orientierung einen tatsächlichen Unterschied macht, auch wenn sie die Freiheiten des Agenten nicht einschränkt. Wenn also der Agent unabhängig von seiner aktuellen Orientierung in alle Richtungen gleichermaßen gut agieren kann, dann erreicht man mit der Einführung einer Orientierung evtl. nur einen erhöhten Rechenaufwand. Dies müsste besonders dann gelten, wenn die Agenten bei jedem Simulationsdurchlauf eine zufällige Startposition erhalten und über keinerlei Gedächtnis verfügen.

3.2 Gene und Evolution

Klaus Meffert gibt in [Mef04] und [Mef05] eine kurze Einführung in die Grundprinzipien der genetischen Algorithmen und Programmierung. Er ist einer der Mitentwickler von *JGAP* [MMM+07], einem Framework für genetische Programmierung mit Java.

die Brutpflegekaste usw.

In [Sch00] gibt Schmutter eine Einführung in das Gebiet der künstlichen Evolution.

Künstliche Evolution und evolutionäre Algorithmen werden häufig verwendet, um Lösungen für Probleme zu finden, deren Suchraum sehr groß ist. Es sei jedoch an dieser Stelle bemerkt, dass dies nicht die einzige denkbare Anwendung ist. In [May04] gehen die Autoren den umgekehrten Weg und versuchen über einen evolutionären Prozess einen Grund dafür zu identifizieren, dass die Schädeldecke des *Homo Erectus* dicker war als die aller Homoide vor und nach ihm. Der Homo Erectus lebte zu einer Zeit, in der er direkt mit dem Homo Sapiens konkurrierte. Die Autoren modellieren ein Agentensimulationssystem in dem das Verhalten der Agenten auf einfache Weise aus einzelnen Genen gesteuert wird. Die einzelnen Gene stehen dabei für bestimmte Verhaltensweisen bzw. Eigenschaften, die mit einer variablen Gewichtung in das Verhalten der Agenten einfließen. Die Eigenschaften sind dabei gegenläufig kodiert, so dass z.B. eine dickere Schädeldecke den Agenten zwar widerstandsfähiger gegen Angriffe macht, dafür aber seine Sichtweite einschränkt.

Derartige Versuche haben für sich genommen natürlich keine Aussagekraft, aber sie können Zusammenhänge aufdecken, die erst durch das Zusammenspiel komplexer Faktoren zu Tage treten.

Die Rolle der Ontogenese ist im Prozess der Evolution nicht zu unterschätzen. Durch die Ontogenese wird es möglich, sehr komplexe Phänotypen aus relativ einfachen Genotypen zu generieren. Hierbei kommen Mechanismen wie Symmetrie und Wiederholung zum tragen. So können z.B. komplexe Organe wie ein Auge im Genom einmal kodiert, im resultierenden Organismus jedoch zweifach vorkommen. In [Sta06] werden einige biologische Merkmale von Genomen aufgeführt, die in künstlichen evolutionären Systemen oft vernachlässigt werden. Dazu zählen vor allem auch die erwähnten Symmetrien.

Des weiteren legt Gruna in seiner Arbeit [Gru07] nahe, dass die Wahl der Genotyp-Phänotyp Abbildung einen großen Einfluss auf die Evolbilität haben kann.

3.2.1 Artificial Embryogeny

Die Anwendung Evolutionärer Algorithmen unter Einbeziehung der Ontogenese wird unter dem Begriff *Artificial Embryogeny* erforscht. Je nachdem wie ausgefeilt die Simulation des Ontogeneseprozesses ist, kann man so mit recht einfachen (kurzen) Genomen bereits komplexe Strukturen erzeugen. Nicht unterschätzen sollte man hierbei die Möglichkeit der Einflussnahme der Umgebung auf die Entwicklung. So können sich aus dem gleichen Genom – abhängig von Umwelteinflüssen – verschiedene Phänotypen entwickeln. Stanley und Miikkulainen geben in [SM03b] einen Überblick über die verschiedenen Ausprägungen dieser Disziplin.

Ein Beispiel für eine konkrete Anwendung liefert Federici in [Fed06]. Dort wird ein Neurokontroller für einen einfachen simulierten Agenten entwickelt. Die Entwicklung erfolgt evolutionär, allerdings unter Einbeziehung der Ontogenese. Ausgehend von einem Genom entwickelt sich aus einem einzelnen Ei eine 8×8 Matrix von Neuronen, die dann mit dem zu simulierenden Agenten verbunden werden und ihn so

steuern. Der Evolutionäre Prozess erfolgt hier also in zwei gekoppelten Simulationsschritten. Zunächst wird ein Genom wie üblich mittels Crossover und Mutation erzeugt. Dann wird in einem zweiten Schritt das Heranwachsen des Neurokontrollers aus dem Genom simuliert. Aus der Leistung des so erzeugten Neurokontrollers leitet sich wiederum der Fitnesswert für den nächsten Evolutionsschritt ab.

3.3 Kommunikation

Für ein koordiniertes Verhalten mehrerer Einheiten ist eine Kommunikation nahezu unerlässlich. Dabei kann die Form der Kommunikation sehr verschieden sein. Beispiele aus der Natur reichen von Signalstoffen (Bakterien) über akustische und optische Signale (Tierwelt) bis hin zu komplexen Sprachen (Mensch). In [Pao97] wird untersucht, ob sich eine Kommunikation allein durch evolutionäre Methoden herausbilden kann. Dazu wurde ein Agentensystem simuliert in dem die Agenten sich paarweise über Energiequellen koordinieren müssen. War diese Koordination erfolgreich, so erhielten beide Agenten gleiche Teile der verfügbaren Energie. Erfolgreiche Individuen konnten sich vermehren, sobald sie eine bestimmte Menge an Energie aufgesammelt hatten, Erfolglose „verhungerten" über die Zeit. Die Autoren beobachteten eine Ausbildung von sehr einfachen Kommunikationssignalen, an die sich über die Zeit alle Agenten anpassten. In [Pao00] erweitern die Autoren ihre Untersuchungen auf die Lokalisierung einzelner Agenten. Dabei können zwei Agenten akustische Signale senden und empfangen und haben die Aufgabe, sich anzunähern. Auch dieses Experiment gelingt, so dass die Autoren zu dem Schluss kommen, dass Kommunikation sich auch über rein evolutionäre Methoden ausbilden kann.

Auch in [JG00] untersuchen die Autoren, ob eine Kommunikation den beteiligten Agenten tatsächlich Vorteile bringt. Dazu erweitern sie ein Predator-Prey Modell dahingehend, dass die Räuber über ein Blackboard untereinander Nachrichten austauschen können. Diese Nachrichten bestanden lediglich im Setzen einer binären Marke und konnte von allen andern Räubern, unabhängig von deren Position, wahrgenommen werden. Dabei stellen sie fest, dass Räuber, die sich einer „Sprache" bedienen können, signifikant bessere Leistungen erbringen, als diejenigen, die agieren ohne untereinander zu kommunizieren.

3.4 Künstliche Neuronale Netze

Damit ein Lebewesen in der Natur überleben kann, muss es permanent Entscheidungen treffen. Diese Entscheidungen beruhen zum Einen auf Informationen, die es seiner Umwelt entnehmen kann (der Wahrnehmung). Zum Anderen basieren sie auf dem Modell, welches ein Lebewesen von seiner Umwelt hat (Reflexe, Instinkte, Erfahrungen).

Diese große Menge von Informationen wird in der Natur von Neuronalen Netzen verarbeitet. Ein Neuronales Netz besteht aus einer großen Zahl einzelner Neuronen

die miteinander verschaltet sind.

Ein Neuron ist eine besondere Art von Nervenzelle, Abbildung 3.1 zeigt eine schematische Darstellung. Sie kann über so genannte *Dendriten* Reize (Impulse) von anderen Nervenzellen empfangen. Überschreitet die Summe der eingehenden Reize eine bestimmte Schwelle, so generiert das Neuron selbst einen Impuls, der über das *Axon* wiederum an andere Nerven weitergeleitet wird.

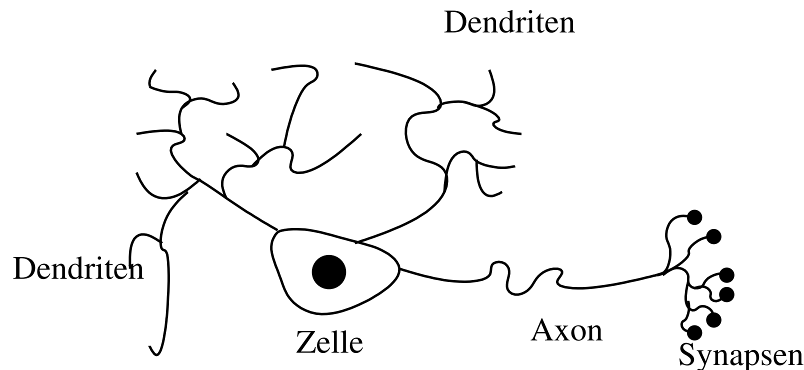


Abbildung 3.1: Ein Neuron empfängt Impulse über seine Dendriten. Sobald die Summe der eingehenden Impulse einen bestimmten Schwellwert überschreitet sendet die Nervenzelle einen Impuls über das Axon aus. Die Synapsen am Ende des Axons sind mit anderen Neuronen über deren Dendriten verbunden.

Durch die Art der Verknüpfung können so die unterschiedlichsten Funktionseinheiten gebildet werden. Angefangen von relativ einfachen Reflexen, die auf eine bestimmte *Eingabe* mit einer entsprechenden *Ausgabe* reagieren, etwa bei der Ansteuerung bestimmter Muskelgruppen während eines Bewegungsablaufes bei Wirbeltieren, über die sehr komplexen Prozesse, die etwa bei der Verarbeitung von optischen Reizen zu abstrakten Symbolen im visuellen Cortex stattfinden, bis hin zu den Vorgängen, die in ihrer Gesamtheit das menschliche Bewusstsein bilden und bis heute nicht verstanden sind.

Alle diese Funktionen entstehen durch die entsprechenden Verschaltungen der einzelnen Neuronen.

Diese Funktionen versucht man mit Hilfe von *Künstlichen* Neuronalen Netzen (KNN) nachzubilden. Dabei betrachtet man ein einzelnes Neuron als kleinste Schalteinheit. Die einzelnen Neuronen des Netzes sind über Kanten miteinander verbunden. Diese Kanten entsprechen den Dendriten und den Axonen des natürlichen Vorbildes und dienen den künstlichen Neuronen als Ein- und Ausgänge. Sie besitzen Gewichtungen, die den jeweiligen Einfluss eines Neurons auf ein anderes darstellen. Diese Gewichtung kann sowohl positiv als auch negativ sein, so dass auch eine Hemmung dargestellt werden kann. Ein künstliches Neuron *schaltet* (feuert) – analog zu seinem biologischen Vorbild – sobald die Summe seiner Reize einen bestimmten Schwellwert überschreitet.

Es gibt eine Vielzahl von verschiedenen Ausprägungen Künstlicher Neuronaler Netze. Die Verschiedenheit kann sich dabei auf sehr unterschiedliche Eigenschaften

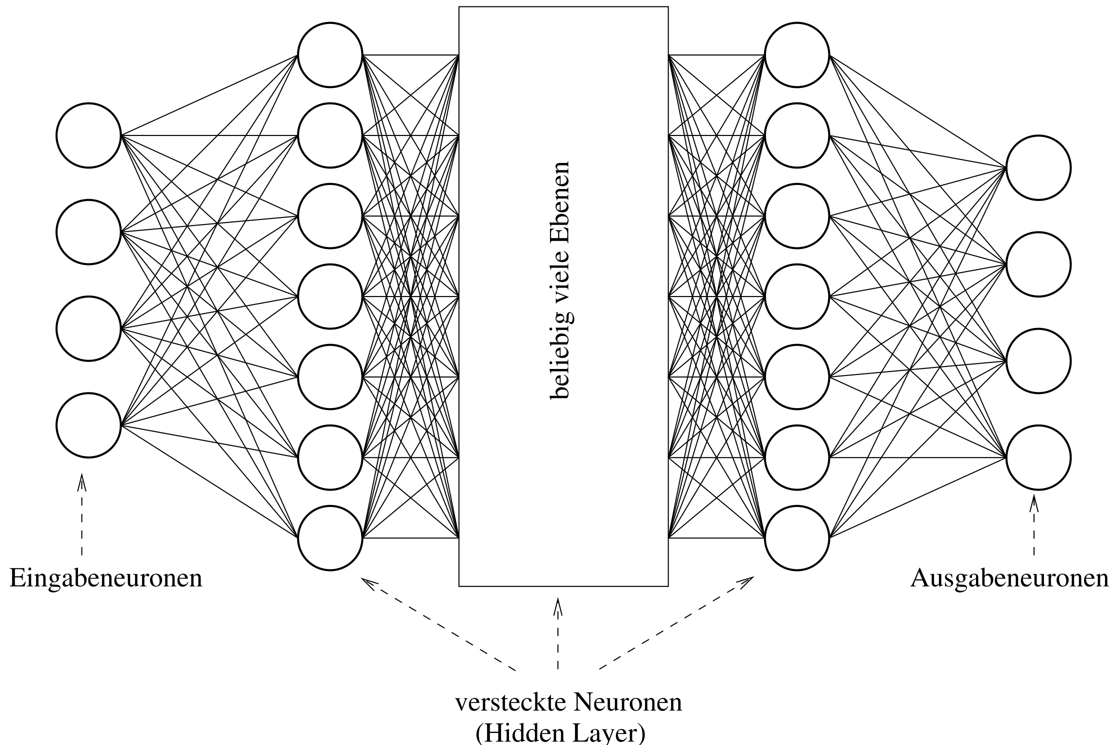


Abbildung 3.2: Strukturbild eines künstlichen neuronalen Netzes: Die einzelnen Neuronen sind in Schichten angeordnet und über gewichtete Kanten miteinander verbunden.

beziehen. Da eine vollständige Erläuterung der unterschiedlichen Formen den Rahmen dieser Arbeit sprengen würde, werden an dieser Stelle nur einige Einzelkriterien vorgestellt, die zum Verständnis notwendig sind. Eine ausführlichere Beschreibung findet sich in [Pat97] und [RN04].

Ein einfaches Unterscheidungsmerkmal ist die Topologie der Netze. Jedes Netz hat eine Anzahl von Ein- und Ausgabeneuronen. Sie können entweder direkt miteinander verbunden werden oder es können sich weitere Neuronen zwischen ihnen befinden. Diese Zwischenneuronen werden Versteckte (Hidden) Neuronen genannt, sie werden in Schichten (Layern) organisiert, den so genannten Hidden Layern. Die Anzahl dieser Hidden Layer kann zwischen Null und beliebig liegen. Abbildung 3.2 zeigt eine schematische Darstellung eines vielschichtigen KNN.

Ein weiteres Merkmal ist die Eigenschaft der *Rekurrenz*. Prinzipiell ist die Verarbeitungsrichtung durch die Neuronen vorgegeben. Da sie jedoch beliebig viele Ein- und Ausgangskanten haben können, spricht grundsätzlich nichts dagegen, dass (bei mehrschichtigen Netzen) die Ausgabe eines Neurons mit der Eingabe eines seiner Vorgänger verknüpft ist. Ist dies nicht der Fall, so spricht man von einfachen *Feed-Forward* Netzen, im anderen Fall von *Rekurrenten* Netzen.

Als letztes Merkmal soll die „Schaltfunktion“ der einzelnen Neuronen betrachtet werden. Hier ist eine beliebige Funktion denkbar, Abbildung 3.3 zeigt drei Plots von

möglichen Schaltfunktionen zum Vergleich nebeneinander.

Die einfachste Schaltfunktion ist die Summe der k Eingangskanten W_{in_k} , so dass sich für den Ausgabewert W_{out} eines Neurons die lineare Funktion

$$W_{out} = \sum_{k=1}^{n} W_{in_k}$$

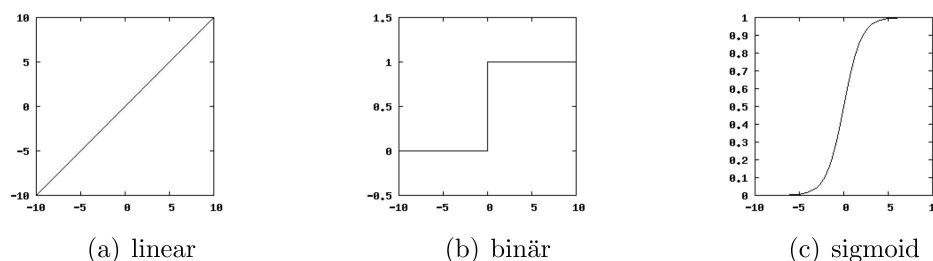
ergibt.

Eine andere Möglichkeit ist die binäre Schaltweise. Solange die Summe der Eingaben nicht über Null liegt, ist auch der Ausgang des Neurons mit Null besetzt. Für den Fall, dass die Eingabesumme größer als Null ist, liefert das Neuron an seinem Ausgang eine Eins.

Schließlich gibt es noch die Möglichkeit, die Eingabesumme des Neurons über eine so genannte *Sigmoide Funktion*

$$W_{out} = \frac{1}{1 + e^{-W_{in}}}$$

mit dem Ausgang zu verknüpfen. Die Funktion bildet einen Definitionsbereich von $-\infty$ bis ∞ auf einen Wertebereich zwischen Null und 1 ab. Dies ist die Variante, die dem Verhalten eines natürlichen Neurons am nächsten kommt, da diese eine leichte „Glättung“ aufweisen und nicht binär funktionieren. Der Grund hierfür liegt in ihre biologischen Funktionsweise.



(a) linear (b) binär (c) sigmoid

Abbildung 3.3: Nebeneinanderstellung verschiedener Schaltfunktionen für ein Neuron

In der Natur verarbeiten Neuronale Netze auch sehr komplexe Informationen in kürzester Zeit und ermöglichen Lebewesen so, angemessen auf ihre Umwelt zu reagieren. Obwohl die „Schaltzeiten“ der einzelnen Neuronen bei relativ langen 0,01-0,02 Sekunden liegen, ist ein Mensch problemlos in der Lage sich zu unterhalten und gleichzeitig ein Auto zu lenken. Ein derartiges Verhalten mit Computern nachzubilden ist bisher nicht (in Echtzeit) gelungen, obwohl die dabei eingesetzten Computer Schaltzeiten von weniger als 0,000001 Sekunden haben. Die Begründung dafür liegt in zwei Eigenschaften Neuronaler Netze. Zum einen arbeiten *Natürliche* Neuronale Netze höchstgradig parallel, alle Neuronen können praktisch gleichzeitig schalten und damit jeweils tausende anderer Neuronen zum Schalten anregen, während ein

herkömmlicher Computer (mehr oder weniger) immer eine Berechnung nach der Anderen ausführen muss. Außerdem eignen sich Neuronale Netze hervorragend zur Klassifizierung komplexer Daten [Wai90], da sie sehr robust gegenüber Störungen (Rauschen) sind und somit relativ zuverlässige Ergebnisse liefern. Genau diese Eigenschaft möchte man sich zu Nutze machen und sie ist auch ein Hauptgrund, warum man an Künstlichen Neuronalen Netzen forscht.

Die Funktion eines Neuronalen Netzes steckt hauptsächlich in der richtigen Verbindung der einzelnen Neuronen und deren Gewichtung. Genau hier liegt aber auch die Schwierigkeit bei der Erzeugung Künstlicher Neuronaler Netze durch den Menschen. Die Aufgabe der Verbindungsfindung ist normalerweise nicht trivial. Will man ein Künstliches Neuronales Netz für eine bestimmte Aufgabe erzeugen, so wird klassischerweise ein vollständig verbundenes Netz (jedes Neuron einer Schicht ist mit jedem Neuron der Folgeschicht verbunden, siehe Abbildung 3.2) genommen und dieses mit einer großen Zahl von Beispielen *trainiert*. Dabei generiert das Netz für jedes Beispiel eine Ausgabe, die dann mit einem gewünschten Sollwert verglichen wird. Aus der Differenz der Ausgabe und des Sollwertes wird dann eine Änderung der Verbindungsgewichte berechnet und das Netz entsprechend angepasst. Ob man diesen Vorgang tatsächlich als „Lernen“ im Sinne von „Begriffsbildung“ bezeichnen kann oder hier nur eine „Reflexmodifikation/Konditionierung“ stattfindet [CS92] ist für diese Arbeit unerheblich.

Es gibt eine große Vielzahl von Lernmethoden die aus den bereits erwähnten Quellen [Pat97] und [RN04] zu entnehmen sind. An dieser Stelle sei lediglich erwähnt, dass in der Natur jedes Lebewesen mit einem vorbelegten Neuronalen Netz geboren wird. Das bedeutet, dass die allermeisten Funktionen bereits als Verbindungen der richtigen Stärke vorhanden sind. Zusätzlich findet ein lebenslanger Lernprozess statt, der das Netz permanent verändert und der Umwelt anpasst [Bru99]. Der Umfang dieser Lernmöglichkeit bestimmt dabei u.a. die erreichbare Intelligenz eines Lebewesens. Einfache Lebewesen werden mit einem nahezu kompletten Verhaltensmuster geboren, welches sich auch im Laufe ihres Lebens nicht ändert. Das hat den Vorteil, dass sie sich in ihrer Umwelt von Anfang an zurechtfinden können, jedoch sind sie dadurch auch stark auf dieses Verhalten festgelegt. Auf der anderen Seite stehen Lebewesen mit komplexerem Verhalten. Sie werden oft nur mit rudimentären Fähigkeiten geboren. In einer Jugendphase lernen sie dann weitere Fähigkeiten von Artgenossen hinzu. Dadurch sind sie in ihrem Verhalten sehr viel flexibler, jedoch in der Lernphase noch nicht eigenständig und deshalb auf die Unterstützung ihrer Artgenossen angewiesen.

Es muss jedoch noch einmal betont werden, dass – auch beim Menschen – der allergrößte Teil der „Topologie“ bereits bei der Geburt festgelegt ist. Anders wäre es nicht möglich, einfache Grundbewegungen wie z.B. das Atmen von Geburt an auszuführen. Alles was als *Lernen* bezeichnet wird kann deshalb als „feintunig“ angesehen werden, da größtenteils lediglich die Gewichtungen bestimmter Verbindungen modifiziert werden. Mit anderen Worten: In der natürlichen Evolution wird vor allem die *Topologie* des Netzes entwickelt. Erlerntes Wissen hat keinen (oder

nur einen sehr geringen, indirekten[2]) Einfluss auf das Genom, welches sich auf die nächste Generation vererbt.

3.5 NEAT

Der in [SBM05b] beschriebene NEAT Algorithmus wird von mehreren Autoren als State-of-the-Art bezeichnet, wenn es darum geht, Neuronale Netze evolutionär zu entwickeln.

Da er für die durchzuführenden Versuche ausgewählt wurde wird er im Folgenden eingehend erklärt. Die Autoren stellen zu dem vorgestellten Algorithmus eine Referenzimplementierung in C++ zur Verfügung. Da das entwickelte Simulationssystem (vgl. Abschnitt 4.5) jedoch auf Java basiert, wird hier auch die Neat Implementierung in Java *JNEAT* [Vie02] vorgestellt und kurz erläutert.

3.5.1 Algorithmus

Die Abkürzung *NEAT* steht für *NeuroEvolution of Augmenting Topologies*. Dies lässt sich ungefähr als *Neuro-Evolution sich entwickelnder Topologien* übersetzen. Die grundsätzliche Idee, die hinter diesem Begriff steckt, ist die Folgende: Der Algorithmus entwickelt ein Neuronales Netz, insbesondere auch dessen Topologie, unter Zuhilfenahme von evolutionären Techniken.

Dabei werden neuronale Netze genetisch kodiert und als *Organismen* betrachtet. Diese müssen sich in einem evolutionären Auswahlprozess bewähren und pflanzen sich fort. Dabei kommt es, analog zum biologischen Vorbild, zu Kreuzung und Mutation des ursprünglichen Erbguts. Die natürliche Auslese wird durch eine Evaluationsfunktion ersetzt, die den einzelnen Netzen einen Fitnesswert zuordnet. Dieser orientiert sich daran, wie gut das entsprechende Netz die zu lösende Aufgabe löst bzw. sich dieser Lösung annähert.

Es wird angenommen, dass eine neue Innovation eine gewisse Anzahl an Generationen braucht, um ihr volles Potential auszuschöpfen. Um zu verhindern, dass neu entstandene, noch nicht fertige Innovationen schnell wieder aussortiert werden, wird die Gesamtmenge der Organismen (Population) in verschiedene Spezies unterteilt. So kann sich eine Innovation in einer separaten Spezies zunächst vollständig herausbilden, bevor sie in Konkurrenz mit anderen, möglicherweise schon fertigen Innovationen tritt.

Ausgangspunkt ist dabei immer das kleinstmögliche Netz, welches nur aus den unbedingt benötigten Neuronen (den Ein- und Ausgangsneuronen) besteht. Evtl. benötigte weitere Knoten (*Hidden Layer*) werden im Laufe der Entwicklung nach und nach hinzugefügt, ebenso weitere Verbindungen zwischen den einzelnen Neuronen. So soll erreicht werden, dass das entwickelte KNN nur so groß ist, wie es die anzunähernde Funktion auch verlangt.

[2]der sogenannte Baldwin-Effekt

Ziel ist die Entwicklung eines Künstlichen Neuronalen Netzes, dass ein bestimmtes Verhalten bzw. eine Funktion annähert. Ein Standardbeispiel hierfür ist die XOR-Funktion, die der Referenzimplementierung der Autoren beiliegt. Hier werden die beiden Eingangswerte an zwei Eingangsneuronen des KNN gelegt, das Ergebnis wird aus dem einzigen Ausgangsneuron gelesen. NEAT entwickelt nun ein KNN, welches immer dann am Ausgangsneuron feuert, wenn die an den Eingängen des Netzes angelegten Werte in der gewünschten XOR-Funktion eine 1 liefern.

Das XOR Beispiel ist natürlich sehr einfach, da es nur zwei Eingangs- und ein Ausgangsneuron hat. Theoretisch kann die gesuchte Funktion jedoch beliebig komplex sein und somit jedes KNN entwickelt werden. Es ist also auch denkbar, dass ein KNN entwickelt wird, welches eine komplexe Mustererkennung durchführt und dabei 30×30 oder mehr Eingangsneuronen auf z.B. 26 Ausgangsneuronen abbildet und damit eine Erkennung von Buchstaben durchführt.

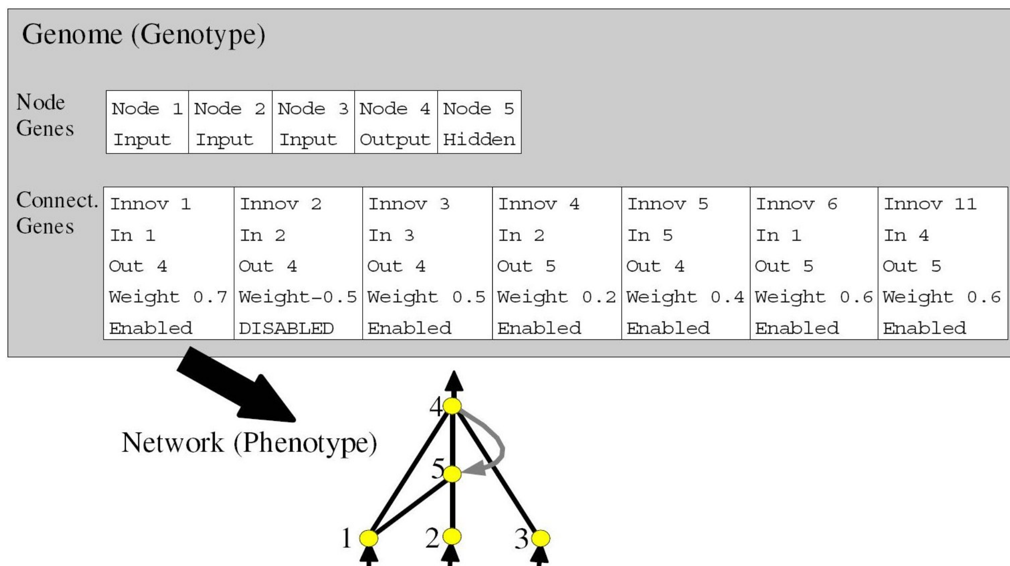


Abbildung 3.4: *Beispiel für die Zuordnung von Genotyp zu Phänotyp bei NEAT:* Das abgebildete Genom (Genotyp) erzeugt das darunter abgebildete Netz (Phänotyp). Es gibt drei Input-Nodes, ein Output-Node und ein Hidden-Node, außerdem insgesamt sieben Verbindungsgene. Das zweite Gen ist abgeschaltet, so dass es im Phänotypen nicht ausgeprägt ist. Das Genom kann beliebig lang sein und somit auch beliebig komplexe Netze darstellen. Über jedem (Verbindungs-) Gen steht die Innovationsnummer, die NEAT erlaubt, sich entsprechende Gene beim Crossover zu identifizieren. Diese Kodierung ist effizient und erlaubt eine Änderung der Netzwerkstruktur im Laufe der Evolution. (aus: [SBM05b])

Wie bereits in Abschnitt 3.2 angesprochen, spielt die Kodierung der Gene und somit die Zuordnung von Phänotyp zu Genotyp eine wichtige Rolle. Für NEAT wurde eine direkte Kodierung gewählt, so dass ein Gen genau einer Ausprägung des Netzes entspricht. Dabei wird zwischen Node-Genes (Knoten/Neuronen-Genen) und

Connection-Genes (Verbindungsgenen) unterschieden.

Wie in Abbildung 3.4 zu sehen, beinhaltet ein Knoten-Gen lediglich die (eindeutige) Knotennummer mit dem zugehörigen Knotentyp. Ein Verbindungsgen enthält hingegen fünf wichtige Informationen: den Start- bzw. Endknoten der Verbindung, die Verbindungsgewichtung, ein enable-Flag und die Innovationsnummer. Letztere spielt eine wichtige Rolle bei der Rekombination zweier Genome. Jedes neue Gen erhält eine eindeutige Innovationsnummer. Bei einer Kreuzung (Crossover) werden die Innovationsnummern der Gene verglichen, Gene mit gleicher Nummer entsprechen dann immer auch den gleichen topologischen Eigenschaften im Netz und können entsprechend vereinigt werden. Dieser Mechanismus wird in [SBM05a] näher erläutert.

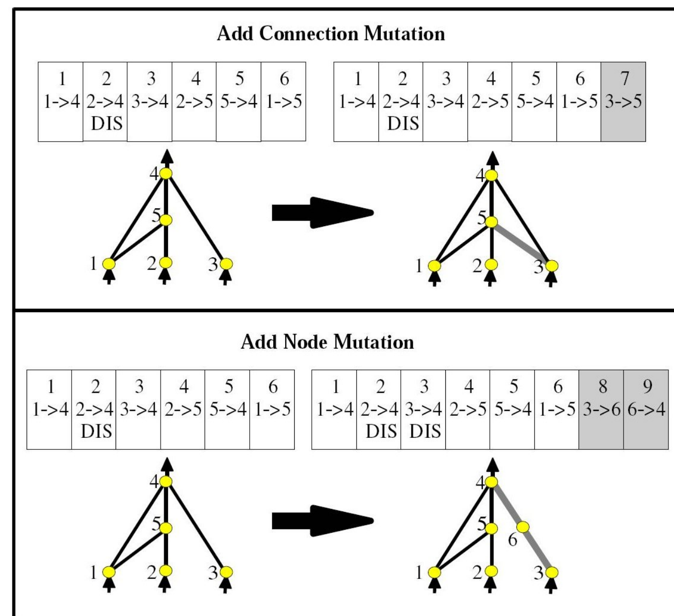


Abbildung 3.5: Es gibt zwei Arten der Mutation bei Neat: die Add-Connection-Mutation und die Add-Node-Mutation (aus: [SBM05b])

Wie bereits in Abschnitt 2.2.3 erwähnt ist die Mutation ein wichtiger Faktor bei der evolutionären Suche. Bei Neat sind zwei Arten der strukturellen Mutation vorgesehen: die Add-Connection-Mutation und die Add-Node-Mutation (vgl. Bild 3.5.1). Bei der Ersten werden zufällig zwei Nodes ausgewählt, zwischen denen noch keine Verbindung besteht. Diese wird dann, in Form eines neuen Verbindungsgens, dem Genom hinzugefügt, die Gewichtung der neuen Verbindung wird dabei ebenfalls zufällig bestimmt.

Die andere Mutation ist die Add-Node-Mutation. Da NEAT immer mit einem minimalen Netz ohne versteckte Neuronen startet, ist der Schlüssel zu komplexerem Verhalten das Hinzufügen weiterer Neuronen. Da jedoch das Risiko einer negativen Mutation minimiert werden soll werden neue Neuronen immer auf bestehenden Verbindungen eingefügt. Es werden also zwei zufällige Neuronen bestimmt, die bereits

verbunden sind. Das Gen welches diese Verbindung repräsentiert wird „abgeschaltet“ und es werden zwei neue Verbindungen eingefügt, von der ursprünglichen Quelle hin zum neuen Neuron und von dort weiter zum ursprünglichen Ziel. Eine dieser Verbindungen erhält die Gewichtung der abgeschalteten Verbindung, die andere erhält eine Gewichtung von eins. Dadurch soll gewährleistet werden, dass etablierte Verhaltensweisen nicht verloren gehen. Eine Add-Node-Mutation fügt also insgesamt drei Genen hinzu und schaltet eines ab.

Zum Verständnis der vorliegenden Arbeit sollte diese Beschreibung ausreichen, nähere Erläuterungen finden sich vor allem in [SM06, SBM05b, SBM05a, SM03a].

3.6 JNEAT

Es gibt mehrere frei verfügbare Implementierungen des NEAT-Algorithmus. Die Autoren des ursprünglichen Algorithmus lieferten auch eine Referenzimplementierung in der Programmiersprache C++. Neben Implementierungen in anderen Sprachen gibt es auch (mindestens) zwei Implementierungen in Java.

Während *ANJI* (Another NEAT Java Implementation) [JTH05] auf *JGAP* (Java Genetic Algorithms Package) [MMM+07] aufsetzt und den NEAT-Algorithmus mit JGAP-Mitteln implementiert, ist *JNEAT* [Vie02] eine direkte Konvertierung der originalen C++ Implementierung in Java. Da Letztgenannte kleiner und überschaubarer ist, wurde sie für dieses Projekt gewählt.

Der Autor von JNEAT, Ugo Vierucci, ist Italiener und schrieb unglücklicherweise die wenigen vorhandenen Kommentare im Quellcode zum überwiegenden Teil auf Italienisch. Da jedoch der Code angepasst und in Repast integriert bzw. mit Repast kooperieren sollte war es unbedingt notwendig, diesen in einen benutzbaren Zustand zu versetzen. Deshalb wurde der Java Code mit dem original C++ Code verglichen, die einzelnen Komponenten identifiziert und die englischen Kommentare in die Java Version übernommen. Erst danach konnte mit dieser gearbeitet werden.

Im Folgenden wird ein Überblick über die NEAT Implementierung JNEAT gegeben. Der Schwerpunkt wird dabei auf der Durchführung eines Evolutionsschritts liegen, da dessen Verständnis für die spätere Verwendung in Verbindung mit Repast unabdingbar ist. Außerdem wird dadurch klar, wie die theoretischen Ansätze aus Abschnitt 3.5 programmiertechnisch umgesetzt wurden. Die damit verbundenen Parameter werden ebenfalls erwähnt. Sie spielen für die später durchzuführenden Versuche eine wichtige Rolle.

3.6.1 Evolution/Experiment

Die Grundlage eines jeden JNEAT-Experiments bildet die Klasse *Evolution*. In dieser Klasse muss für jedes Experiment, das durchgeführt werden soll eine *evaluate*-Methode geschrieben werden. Dabei werden zunächst für den betroffenen Organismus Werte an die Eingabeneuronen seines Neuronalen Netzes gelegt. Dann wird die Ausgabe des Netzes berechnet und ausgelesen. Anschließend wird aus dem

zurückgelieferten Ergebnis ein neuer Fitness-Wert berechnet. Dazu werden die Ergebnisse üblicherweise mit einem Soll-Wert verglichen, die Bewertung kann jedoch beliebig erfolgen. Wichtig ist lediglich, dass dem Organismus am Ende der Evaluate-Methode ein neuer Fitness-Wert zugewiesen wird.

Da es häufig vorkommt, dass ein Neuronales Netz evolviert werden soll, dessen Zweck vorher genau bekannt ist (z.B. das Erkennen bestimmter Muster mit einer bestimmten Zuverlässigkeit) gibt es außerdem die Möglichkeit, ein *Winner*-Tag zu setzen. Es ist dazu gedacht, einen Organismus zu markieren, dessen Leistung als ausreichend gut betrachtet wird.

Neben *evaluate* gibt es zwei weitere wichtige Methoden. In der Methode *epoch* wird zunächst für jeden Organismus der Population die aktuelle Fitness festgestellt (*evaluate*-Methode). Danach kann für jede Spezies die durchschnittliche und die maximale Fitness festgestellt werden. Dies wird für den später folgenden Evolutionsschritt benötigt. Abschließend wird über die Methode *Population.epoch* eine neue Generation von Organismen erzeugt.

3.6.2 Population

Soll eine neue Generation erzeugt werden, so geschieht dies mit der Methode **Population.epoch**.

Zunächst werden die Fitness-Werte der einzelnen Organismen angepasst. Dies geschieht zum einen, um noch jungen Spezies Zeit zu geben sich zu entwickeln, andererseits werden Spezies bestraft, die lange keine Innovationen mehr hervorgebracht haben. Außerdem werden während dieses Vorgangs alle Organismen markiert, deren Fitness unterhalb eines Grenzwertes liegt (**p_survival_thresh** × Durchschnitts-Fitness) und somit dazu bestimmt sind, beim nächsten Evolutionsschritt zu sterben.

Schutz neuer Spezies Grundsätzlich soll eine Spezifizierung (Herausbildung mehrerer verschiedener Spezies) unterstützt werden. Da die Einteilung der einzelnen Individuen nach ihrer genetischen Verschiedenheit erfolgt, wird ein Individuum mit einer neuen Innovation wahrscheinlich in eine neue Spezies eingeordnet. Da jedoch eine neue Innovation (zum Beispiel eine neue Verbindung im KNN) nicht automatisch eine sofortige Verbesserung der Fitness des betroffenen Individuums zur Folge hat (oft müssen die Kantengewichtungen noch über mehrere Generationen angepasst werden) wird neu entstandenen Spezies für eine gewisse Zeit ein Fitness-Bonus zugestanden. Nach Ablauf dieser Zeit muss die Spezies aus eigener Kraft überleben oder stirbt aus.

Nun wird jedem Organismus eine Anzahl von Kindern zugestanden. Diese Zahl hängt von seiner Fitness in Relation zur Gesamtfitness ab und repräsentiert nur einen Verhältnisfaktor. Aus den Zahlen der einzelnen Organismen (jeweils einer Spezies) wird dann (durch einfache Summation) bestimmt, wie viele Kinder dieser Spezies als Ganzes zugestanden werden.

Abschließend wird noch geprüft, ob es Spezies gibt, die lange keine Innovationen mehr hervorgebracht haben. Ab einer bestimmten Stagnationsdauer **p_droppoff_age**

wird eine solche Spezies gelöscht. In einem solchen Fall müssen die Kinder, die der gelöschten Spezies zugestanden wurden, natürlich auf die verbleibenden Spezies aufgeteilt werden.

3.6.3 Network

In dieser Klasse ist vor allem die Methode *activate* wichtig. Sie führt die tatsächliche Simulation des KNN durch, indem aus den Werten, die an den Eingabeneuronen anliegen, mittels einer Sigmoiden Schaltfunktion die Werte der nächsten Netzwerkschicht bzw. der Ausgabeneuronen berechnet werden.

3.6.4 Species

In der Klasse *Species* ist vor allem die Methode *reproduce* interessant, sie erzeugt die jeweils nächste Generation. Da die Populationsgröße festgelegt ist, gibt es eine feste Zahl zu erwartender Kinder. Diese werden der Reihe nach erzeugt. Jede Spezies hat einen „Champion“. Ein Champion hat innerhalb seiner Spezies eine für ihn reservierte Anzahl von Nachkommen. Diese werden zuerst generiert. Dabei entstehen Champion-Kinder mit einer Wahrscheinlichkeit von 80% als Kopien des Champions. Lediglich die Kantengewichtungen der Netze dieser Kinder werden über eine Gaußfunktion leicht „verschmutzt“. Mit der verbleibenden Wahrscheinlichkeit von 20% wird eine Verbindung zwischen zwei zufällig gewählten Knoten in das Netz eingefügt. Der Champion wird als Klon ebenfalls in die neue Generation übernommen.

Nachdem die reservierte Anzahl an Champion-Kindern erzeugt wurde, werden nun die verbleibenden Kinder der Population erzeugt. Zunächst wird bestimmt, ob es eine einfache Klon-Mutation oder eine echte Paarung (Crossover) werden soll, eine initiale Wahrscheinlichkeit dafür ist als Parameter vorgegeben.

Falls nur ein Individuum im Spezies-Pool vorhanden ist, muss eine einfache Mutation verwendet werden. Gibt es mehrere, so besteht eine Wahrscheinlichkeit von **p_mutate_only_prob** dafür, dass eine einfache Mutation und kein Crossover ausgeführt wird. In diesem Fall wird entweder ein neuer *Node* (mit einer Wahrscheinlichkeit von **p_mutate_add_node_prob**) oder ein neuer *Link* hinzugefügt (mit der Wahrscheinlichkeit **p_mutate_add_link_prob**) oder es werden verschiedene andere Mutationen angewendet. Mögliche Mutationen sind:

mutate_link_weight Das Gewicht einer Verbindung wird verändert.

mutate_toggle_enable Schaltet ein Gen aus bzw. wieder ein.

mutate_gene_reenable Schaltet das erste auffindbare abgeschaltete Gen wieder ein.

Die aufgelisteten Mutationsarten haben jeweils ihre eigenen Wahrscheinlichkeiten und werden unabhängig voneinander angewendet, was bedeutet, dass es vorkommen kann, dass mehrere in einem einzigen Mutationsschritt Anwendung finden.

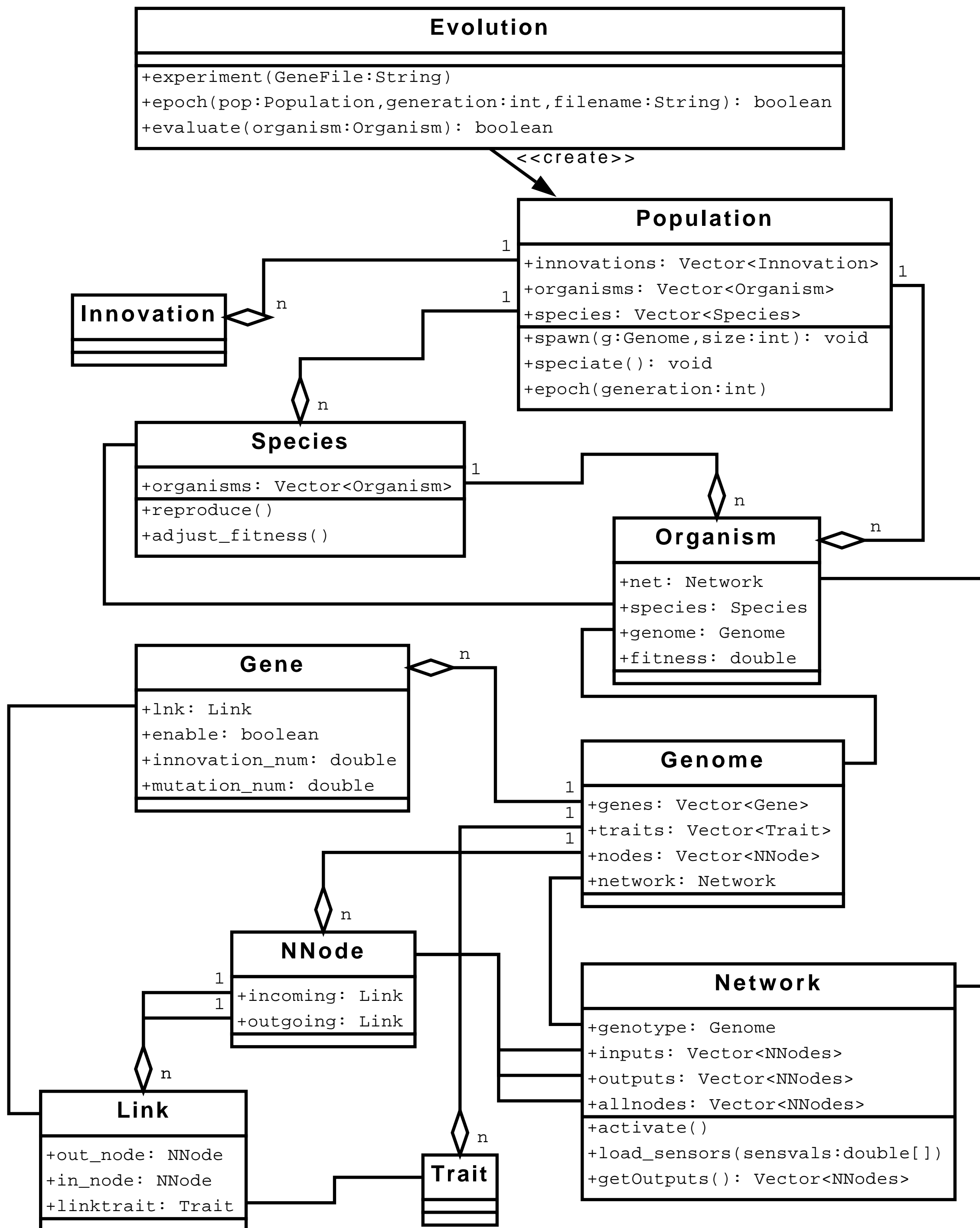


Abbildung 3.6: Überblick über die wichtigsten JNeat Klassen

Falls eine Paarung stattfinden soll, wird ein zufälliges Mitglied der Spezies gewählt (der erste Partner) und diesem ein weiteres zufälliges Mitglied zugeordnet. Dieses zweite Mitglied kann mit einer gewissen Wahrscheinlichkeit (**p_interspecies_mate_rate**) auch einer anderen Spezies angehören. Für das Crossover stehen die folgenden Möglichkeiten zur Verfügung:

mate_multipoint Bei dieser Methode werden die Genome verschmolzen, indem all jene Gene, die in nur einem Elternteil vorkommen (Innovationsnummern werden verglichen) auf das Kind vererbt werden. Ist ein Gen in beiden Elternteilen vorhanden, so wir zufällig bestimmt, welches der beiden in Frage kommenden Gene an das Kind vererbt wird.

mate_multipoint_avg Diese Methode funktioniert fast genauso, wie die Vorgenannte, der Unterschied besteht darin, dass im Fall eines doppelt vorkommenden Gens der Durchschnitt der beiden Einzelgewichtungen gebildet wird und dann auf das Kind übertragen wird.

mate_singlepoint Bei dieser Methode wird ein Punkt auf dem Genom eines Elternteils bestimmt. Alle Gene bis zu diesem Punkt werden vom ersten Elternteil auf das Kind übertragen, alle Gene ab diesem Punkt erbt das Kind vom zweiten Elternteil.

Das so neu erzeugte Kind wird nun wiederum mutiert, genau wie dies auch für geklonte Kinder der Fall ist.

Abschließend wird das neue Kind einer Spezies zugeordnet, die Zuordnung findet dabei aufgrund der Genomähnlichkeit statt. Sollte es zu keiner existierenden Spezies passen, so wird eine neue erzeugt.

3.7 Anwendung

Dass all die erwähnten Mechanismen nicht nur in der theoretischen Forschung und Simulation relevant sind, zeigen z.B. die Arbeiten [TGT+06] und [TND06] in denen ein Schwarm von Robotern lernt, sich als koordinierter Schwarm zu verhalten und dadurch z.B. Formationen zu bilden oder Hindernisse zu überwinden, die die einzelnen Roboter alleine nicht überwinden können.

Koenig et al. beschreiben in [KL01] verschiedene Experimente zu diesem Szenario. Unter anderem werden hier verschiedene Markierungsstrategien verglichen. Dabei wird geprüft, wie die Zahl der Agenten das Ergebnis beeinflusst und wie sich Störungen (z.B. defekte Agenten oder das Löschen von Markern) auf das System auswirken. Die Autoren stellen ebenfalls fest, dass ein markerbasiertes Vorgehen weit besser ist, als eine zufällige (unkoordinierte) Strategie. In [SK03] wird diese Theorie auf echte Roboter angewendet. Auch in [WLB96] werden Pheromonsimulationsexperimente durchgeführt, allerdings variieren hier nicht die Markierungs- sondern die Auswahlstrategien, also die Art, wie auf Markierungen reagiert wird. Dies dient der Optimierung der in ihrer Arbeit behandelten Wegewahlalgorithmen.

Kapitel 4

Design & Implementierung

The great tragedy of Science - the slaying of a beautiful hypothesis by an ugly fact.

Thomas Henry Huxley

Nachdem die Grundlagen dieser Arbeit in den vorangegangenen Kapiteln dargelegt wurden, wird nun eine mögliche Lösung für das eingangs vorgestellte Problem der Reinigung von Flächen skizziert. Als Lösungsansatz dient der in Kapitel 3.5 vorgestellte NEAT-Algorithmus.

Dieses Kapitel fasst die zu lösende Aufgabe noch einmal zusammen und gibt einen Überblick über das verwendete Modell. Danach behandelt es die Auswahl einer passenden Simulationsumgebung. Schließlich wird das implementierte Simulationstool *MyOwnWorld* vorgestellt und beschrieben. Den Abschluss dieses Kapitels bildet die Beschreibung der einzelnen Agenten, die während der Simulation zur Anwendung kommen.

4.1 Aufgabe

Die ursprüngliche Aufgabe der Arbeit bestand in der Untersuchung von Möglichkeiten der Koordinierung von Putzrobotern. Für das Szenario von mehreren kleinen Einheiten, die über eine beschränkte Sensorik und minimale Kommunikationsmöglichkeiten verfügen, sollte eine Methode gefunden werden, die es den Agenten ermöglicht, lokal zu lernen und ihr gelerntes Wissen untereinander auszutauschen. So könnte eine große, unbekannte Fläche, eventuell mit Hindernissen, effizient gereinigt werden.

Da diese Untersuchung theoretischer Natur ist, werden keine realen Einheiten benutzt. Stattdessen wird das Szenario im Rechner modelliert und nachgebildet. Die Simulation soll so realitätsnah sein, dass sich eine gefundene Methode ohne große Schwierigkeiten auf ein reales System übertragen lässt.

In einer Simulation sollen sich also Agenten über ein Spielfeld bewegen, das ein verschmutztes Büro repräsentiert. Sie sollen dabei von einem Künstlichen Neurona-

len Netz gesteuert werden, welches mit Hilfe des NEAT-Mechanismus über die Zeit verbessert wird. Am Ende der Arbeit soll entweder ein Agent stehen, dessen Einzelverhalten dazu führt, dass eine Gruppe von diesen Agenten das Spielfeld möglichst effizient und gleichmäßig reinigt, oder aber die Erkenntnis gewonnen werden, dass der gewählte NEAT-Algorithmus für die Aufgabe ungeeignet ist.

4.2 Modell

Da die Untersuchungen in einer simulierten Welt geschehen sollen, muss diese zunächst spezifiziert werden. Dies geschieht anhand eines Modells der realen Welt. Es hat die Aufgabe, all jene Faktoren der realen Welt abzubilden, die einen Einfluss auf das zu simulierende Verhalten haben sollen bzw. könnten und dabei dennoch die Komplexität der realen Welt zu verdecken, indem all jene Faktoren vernachlässigt werden, deren Einfluss auf das zu simulierende Verhalten als unerheblich eingestuft werden. Es entsteht also ein *idealisiertes* Abbild der realen Welt. Zum besseren Verständnis ist das beschriebene Modell auf Abbildung 4.1 graphisch veranschaulicht.

Grundsätzlich soll ein simulierter Roboter durch ein simuliertes Büro fahren und dieses dabei säubern. Der erste Idealisierungsschritt besteht darin, die Zeit zu diskretisieren. Sie läuft also nicht kontinuierlich, sondern in einzelnen Simulationsschritten, den *Ticks*, ab. Dementsprechend können Ereignisse, gleich welcher Art, nur zu diskreten Ticks stattfinden.

Der nächste Schritt besteht darin, die Roboter durch Softwarekomponenten zu ersetzen, den *Agenten*. Diese Agenten können eine ganze Reihe verschiedener Ausprägungen besitzen, die ihr Verhalten in der Welt repräsentieren. Auf diese Ausprägungen wird in Abschnitt 4.6 näher eingegangen.

Das Büro, durch das sich die Agenten bewegen, wird als eine rechteckige Ebene angesehen, deren Fläche in gleichgroße, quadratische Zellen unterteilt ist. Ein solches Konstrukt wird als *Grid-Layout* bezeichnet. Die Größe des in den späteren Experimenten verwendeten Grids kann für jedes Experiment separat definiert werden, in den meisten Fällen kommt ein 100×100 Zellen großes Grid zum Einsatz. Die genaue Konfiguration kann der jeweiligen Parameterliste (unter *worldXSize* bzw. *worldYSize*) des durchgeführten Versuchs entnommen werden, ein Beispiel hierfür stellt Tabelle 5.1 dar.

Da das Grid das Büro repräsentiert und in dieser Funktion auch mehrere Eigenschaften abbilden soll, hat es drei Ebenen. Auf der ersten Ebene, dem *Agent-Layer* werden „physische“ Objekte dargestellt. Die Agenten befinden sich ebenso auf dieser Ebene wie etwa Wände oder Hindernisse. Eine Zelle dieser Ebene kann also entweder leer sein oder genau einen Agenten bzw. ein Hindernis enthalten.

Da die Aufgabe das Reinigen des Büros ist, muss die Verschmutzung im Modell enthalten sein. Die zweite Ebene des Grid, das *Dirt-Layer*, wird deshalb dazu verwendet, die Verschmutzung der einzelnen Zellen abzubilden. Eine Zelle kann beliebig stark verschmutzt sein, der Verschmutzungsgrad wird durch eine Gleitkommazahl

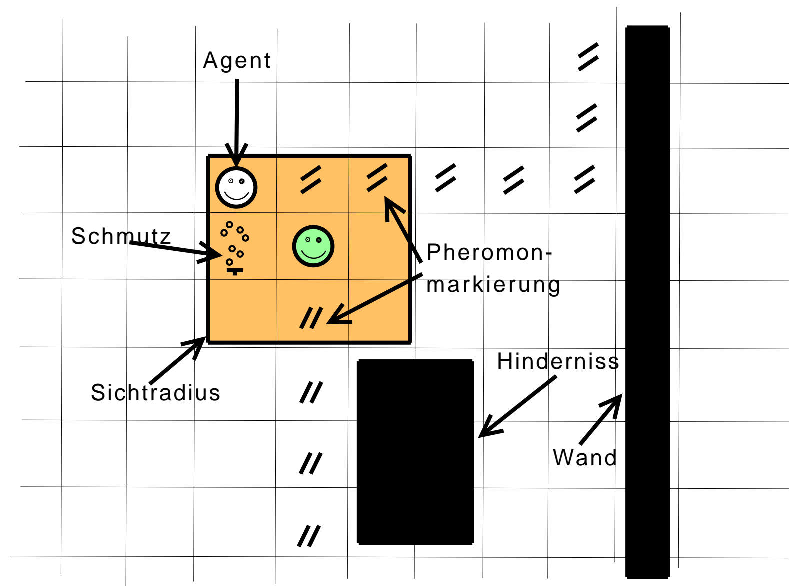


Abbildung 4.1: Szenario: Agenten mit einer Sichtweite von einem Feld, Hindernissen, Grenzwand, Pheromonspuren und Schmutz.

repräsentiert. Da in der realen Welt ein Stück Fußboden nie zu 100% sauber ist, wird auch im Modell ein Grenzwert eingeführt, ab dem eine Zelle als verschmutzt angesehen wird. Liegt der Verschmutzungsgrad unterhalb dieses Grenzwertes, so wird die Zelle als sauber angesehen. Auch dieser Parameter ist, unter dem Namen *worldCleanCellThreshold*, in der Parameterliste des durchgeführten Versuchs enthalten, er lag zumeist bei 1.0.

Das Dirt-Layer wird initial verschmutzt, zunächst mit einer zufälligen Streuung zwischen Null und dem Parameter *worldDirtMax*, in späteren Versuchen wurde auf die Streuung verzichtet und einfach der *worldDirtMax*-Wert als Initialverschmutzung genommen. Zusätzlich kommt mit jedem Simulationsschritt eine zusätzliche Verschmutzung hinzu. Diese Verschmutzung ist ebenfalls zufallsbedingt und verteilt sich gleichmäßig zwischen Null und dem Wert des Parameters *worldDirtRespread*.

Da es sich gezeigt hat, dass Strategien in Multiagentensystemen von einer Möglichkeit zur Kommunikation profitieren (vergl. [JG00]), soll den Agenten die Möglichkeit gegeben werden, über Marker miteinander zu kommunizieren. Diese Art der Kommunikation ähnelt denen von Ameisen, die ebenfalls Pheromonmarker in der Landschaft hinterlassen [BT00]. Im Modell wird also die dritte Ebene des Grid, das *Pheromon-Layer*, dazu verwendet, diese Marker darzustellen. Jeder Agent kann die Zelle, in der er sich gerade befindet, mit einer Marke beliebiger Stärke versehen. Diese wird den Markern, die bereits früher auf dieser Zelle hinterlassen wurden, hinzu addiert. Eine weitere Differenzierung der Marker, z.B. nach Erzeugern, findet nicht statt.

Mit jedem Simulationsschritt werden alle Marker des Pheromon-Layer um einen bestimmten Prozentsatz (*worldPheromoneFalloff*) abgeschwächt.

Um seine Aufgabe (das Reinigen von Zellen) zu erfüllen, stehen dem Agenten bei jedem Simulationsschritt fünf Handlungsoptionen zur Verfügung. Diese sind das Reinigen einer Zelle, die Bewegung in eine von vier Richtungen und das Markieren einer Zelle. Eine Bewegung kann dabei immer um genau eine Zelle erfolgen und der Agent kann *entweder* eine Zelle reinigen *oder* sich bewegen. Das Markieren einer Zelle kostet hingegen keine Zeit und kann deshalb bei jedem Zug erfolgen.

Der Agent wird zu jedem Simulationsschritt nach seinem Handlungswunsch befragt. Um eine qualifizierte Entscheidung treffen zu können, muss der Agent die Umgebung, in der er sich bewegt, wahrnehmen können. Seine Sichtweite beträgt ein Feld, was bedeutet, dass er Informationen über alle Felder seiner direkten (Moore-) Nachbarschaft erhält. Für das Agenten-Layer ist dies ein binärer Wert (frei oder besetzt, wobei es keinen Unterschied macht, ob eine Zelle von einer Wand, einem Hindernis oder einem anderen Agenten besetzt ist), für die beiden anderen Layer sind es jeweils Fließkommawerte.

Analog zu [JG00] kann ein Agent sich nur auf ein freies Feld bewegen. Sollte das gewählte Feld belegt sein, so bleibt er auf seinem alten Feld stehen, seine Bewegungsphase endet dann ohne dass er sich bewegt hat.

4.3 Begriffe

In diesem und im folgenden Kapitel werden immer wieder Begriffe verwendet, die dem Leser eventuell nicht klar bzw. die missverständlich sind. Deshalb gibt die folgende Zusammenstellung einen Überblick über solche Begriffe und erläutert kurz, wie sie im Kontext dieser Arbeit gebraucht werden.

Simulationsschritt Ein Simulationsschritt ist die kleinste Zeiteinheit in einem simulierten Experiment. Während eines solchen Schrittes (auch *tick* oder *step* genannt) führt jeder Agent einen Zug aus. Außerdem „altert“ die Welt am Ende jedes Schrittes.

Simulationslauf Ein Simulationslauf stellt das einmalige Ausführen einer bestimmten Anzahl von Simulationsschritten dar. Ein Lauf beginnt mit dem Einlesen der zu verwendenden Parameter und endet, nach einer vorbestimmten Anzahl von Schritten (*numberOfSteps*), mit dem Schreiben der statistischen Daten zu diesem Durchgang in ein Logfile.

Experiment Ein Experiment repräsentiert die (evtl. auch mehrmalige) Durchführung einer Simulation mit einer bestimmten Konfiguration, repräsentiert durch den Parameter *numberOfRuns*. Üblicherweise werden die Simulationen 20 Mal wiederholt, jeweils mit einem anderen Startwert für den Zufallsgenerator.[1] Am

[1] Der Java-Zufallsgenerator liefert keine wirklich zufälligen Zahlen, sondern eine Reihe quasi-zufälliger Werte. Da diese (in ihrer Reihenfolge) bei gleichem Anfangswert immer gleich sind, wird der Startwert variiert. So erhält man für jeden Startwert eine neue quasi-zufällige Folge von Zahlen.

Ende jedes Experiments werden die statistischen Daten der einzelnen Simulationsläufe gemittelt und gespeichert. So wird das „Rauschen“ der Werte verringert und die Ergebnisse werden aussagekräftiger. Eine solche Mittelung ist nur zulässig, wenn dadurch evtl. vorhandene, in verschiedenen Durchläufen verschobene, *Zyklen* nicht „herausgemittelt“ werden. Solche Zyklen konnten jedoch in der Erprobungsphase der Simulationsumgebung nicht entdeckt werden, weshalb die Mittelwertbildung an dieser Stelle verwendet werden kann.

4.4 Simulationsumgebung

Nachdem das beschriebene Szenario spezifiziert war, galt es, ein Simulationstool zu finden, welches in der Lage ist, das entwickelte Modell zu implementieren. Es gibt einige frei verfügbare Frameworks zur Realisierung von Agentensimulationen, die meisten davon als Open Source, so dass auch Anpassungen, die eventuell vorgenommen werden müssen, relativ schnell umgesetzt werden können.

Betrachtet man die verfügbaren Agentensimulationstools genauer, wird schnell erkennbar, wie weit der Begriff *Agent* gefasst ist (vergleiche Abschnitt 2.3). Viele dieser Tools sind für einen bestimmten Zweck bzw. ein bestimmtes Szenario entworfen worden. Das ***J****ava* ***A****gent* ***DE****velopment Framework – Jade* [BCR+08] z.B. ist darauf spezialisiert, Softwareagenten zu entwerfen, die über ein Peer-to-Peer Netzwerk Dienstleistungen erbringen. Das Swarm Projekt [Gro08] hingegen beschäftigt sich seit Jahren mit der Simulation von Multiagentensystemen. Es sind Implementierungen in Objectiv-C und in Java verfügbar. Ausgehend von den Erfahrungen, die damit über die Jahre gemacht wurden, wurde das *Repast* Framework entwickelt.

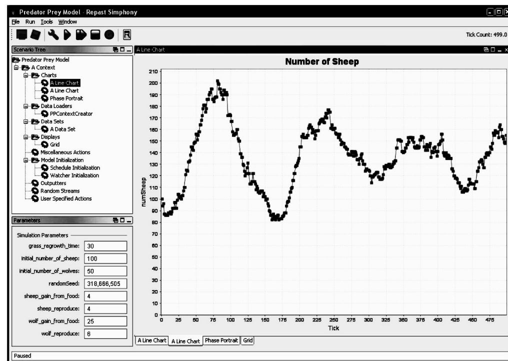
4.4.1 Repast

Am Beispiel von Repast soll hier eines der verfügbaren Frameworks näher betrachtet werden. In Repast können sowohl grid- als auch netzwerkbasierte Agenten simuliert werden. Es eignet sich somit nicht nur zu Experimenten mit zellulären Automaten, sondern es sind auch Untersuchungen zu sozialen Netzwerken oder wirtschaftlichen Zusammenhängen möglich.

Neben den älteren Versionen Repast-J (Repast für Java), Repast.Net (eine C# Implementierung) und Repast-Py (eine Implementierung für die Skriptsprache Python) heißt die aktuelle Version *Repast-Simphony*. Es handelt sich dabei wiederum um eine auf Java basierende Implementierung.

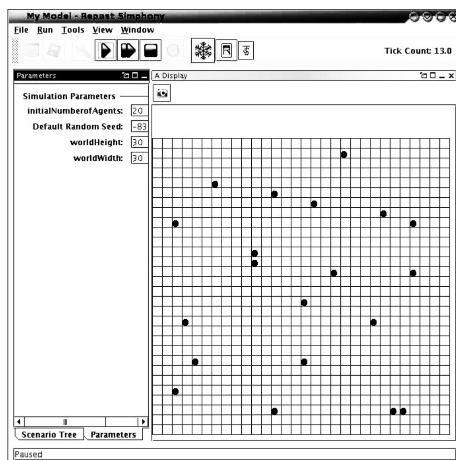
Sie umfasst nicht nur die grundlegenden Bibliotheken, die zur Programmierung einer Agentensimulation notwendig sind, sondern bietet darüber hinaus auch eine grafische Oberfläche. Diese bietet neben einigen Bedienelementen für den Simulationslauf (Start, Stop, Pause oder Schrittweise Vorwärts) auch verschiedene Methoden zur Visualisierung einer Simulation, angefangen bei einfachen Plots von einzelnen Werten über die schematische Darstellung des simulierten Modells in 2D bis hin zur 3D-Darstellung einer laufenden Simulation, siehe hierzu auch Abbildung 4.2.

Zusätzlich bietet sie ein Point-and-Click Interface zur Erstellung eigener Modelle und Agenten. Damit soll es auch Anwendern, die sich zwar für Agentensimulationen interessieren, jedoch über keine Programmierkenntnisse verfügen, ermöglicht werden, eigenständig Modelle zu implementieren.

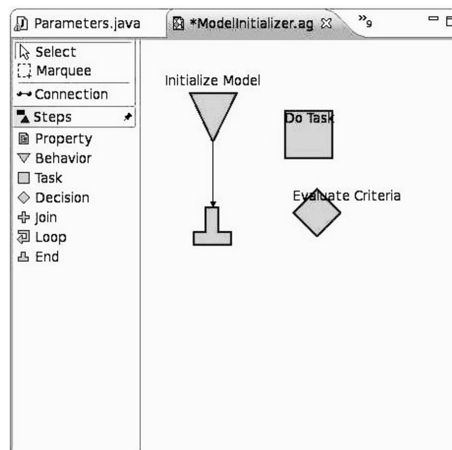


(a) Plot eines Einzelwertes

(b) 3D-Darstellung einer Simulation



(c) 2D-Darstellung einer Simulation



(d) Graphischer Agent-Editor

Abbildung 4.2: Das Repast Framework stellt mehrere Möglichkeiten zur Echtzeit-Visualisierung von Daten zur Verfügung, außerdem bietet es einen graphischen Editor zur Entwicklung von Agentenverhalten.

Die Struktur eines Repast Simulationsmodells wird nun näher betrachtet, soweit sie aus den verfügbaren Quellen [Koe04, NHCV05b, NHCV05a, TNH+06, LRH06, NSV+06, PNC+06, NTCO06, ONSP07] zu entnehmen ist. Dabei ist keine allgemeine, vollständige Beschreibung des Tools das Ziel, sondern es wird auf einzelne Aspekte eingegangen, die im Zusammenhang mit der zu lösenden Aufgabe interessant, oder die für die anschließende Bewertung des Tools relevant sind.

Interner Aufbau

Es gibt eine Vielzahl von Komponenten und Objekten, aus denen ein Modell bestehen kann. Zum einen gibt es natürlich die Agenten, zum anderen auch abstraktere Konstrukte wie *ValueLayer*, *Context* oder *Projection*. Diese Komponenten werden, je nach Modell und Zielsetzung, miteinander in Beziehung gesetzt. Diese Beziehungen sind – in gewissen Grenzen – frei definierbar und dynamisch, so dass sie sich auch zur Laufzeit ändern können.

Ein *Context* beinhaltet (ohne irgendeine Art von Ordnung) Agenten oder weitere Kontexte und stellt somit eine Art Container dar. In ihm werden logische Einheiten des Modells zusammengefasst.

Eine *Projektion* stellt eine Ordnung innerhalb eines bestimmten Kontextes dar. Zur Zeit sind vier verschiedene Arten von Projektionen verfügbar: Continuous Space Projections, GIS Projections, Grid Projections und Network Projections. Einige Beispiele sollen der Veranschaulichung des Nutzens der Projektion dienen.

So speichert die Grid-Projektion für jedes Objekt des zugehörigen Kontextes die kartesischen Koordinaten, an denen sich dieses befindet. In einem anderen Beispiel ordnet die Netzwerk-Projektion die Kontext-Mitglieder zu einem Graphen, indem sie die Verbindungen zwischen den einzelnen Mitgliedern speichert. Die in einer Projektion gespeicherten Informationen können von den Mitgliedern abgefragt werden. Außerdem können Projektionen zur Laufzeit angezeigt werden. Konsequenterweise kann in einer Projektion nur auf Agenten zugegriffen werden, die Mitglied des zugehörigen Kontextes sind. Wird ein Agent einem Kontext hinzugefügt, so wird er automatisch auch den dem Kontext zugeordneten Projektionen zugeordnet.

Dieses System ist sehr flexibel, da einem Kontext beliebig viele Projektionen hinzugefügt werden können.

Im Zusammenhang mit dem oben entwickelten Modell ist besonders die Grid-Projektion interessant. In Repast kann ein Grid eine beliebige Diminsionalität größer zwei haben. Es unterstützt *Von-Neumann* (4-Zellen) und *Moore* (8-Zellen) Nachbarschaften.

Außerdem kann die Begrenzung eines Grid Randbedingungen haben. Ein torusartiger Rand etwa lässt Agenten, die sich auf einer Seite aus dem Grid hinaus bewegen, auf der gegenüberliegenden Seite wieder erscheinen, reflektierende Wände stoßen den Agenten beim Versuch der Randüberschreitung zurück und ein fester Rand hindert Agenten am verlassen des Grid.

Da die Bedeutungen der verschiedenen Projektionen sehr verschieden sind, sind auch die Transformationsmethoden für jede Projektion spezifisch. Das Bewegen eines Agenten innerhalb eines Grid z.B. erfolgt mittels von der Grid-Projektion zur Verfügung gestellter Methoden z.B. *moveTo* (Bewegung zu einem festen Punkt) oder *moveByDisplacement* (Bewegung um einen bestimmten Verschiebevektor, relativ zur aktuellen Position).

Der Benutzer kann die Randeigenschaft (in einem nicht-torusartigen Raum) jedoch nur nutzen, wenn seine Agenten sich mit einem *moveByDisplacement(T object, int... displacement)*, also immer relativ zu ihrer aktuellen Position, bewegen. Ver-

sucht ein Agent, sich mit einem *moveTo*, also einer absoluten Bewegung, über den Rand hinauszubewegen, so wird eine *SpatialException* geworfen, der Benutzer muss also selbst für eine Randbeachtung sorgen.

Interessant ist auch die Methode *GridPoint.moveByVector(T object, double distance, double... anglesInRadians).* Mit ihr sind Bewegungen der Art „moveByVector(object, 1, Direction.NORTH)“ möglich. Die Winkelangaben werden hier in *Radiant* gemacht, die Orientierung läuft gegen den Uhrzeigersinn, so dass 0 Grad Osten und 90 Grad Norden entspricht. Bei Erfolg wird die neue Position als *GridPoint* zurückgeliefert, andernfalls *Null.* Dabei wird derjenige Grid-Punkt als Ziel der Bewegung gewählt, der dem mathematisch berechneten Punkt am nächsten kommt. Somit können Agenten auch Strategien verfolgen, die nicht unbedingt auf eine Grid-Umgebung ausgelegt sind, die Grid-Projektion setzt die Bewegungsanweisungen selbständig in Grid-Bewegungen um.

Zusätzlich können Grids weitere Eigenschaften besitzen, z.B. Mehrfachbesetzbarkeit, die bestimmt, ob mehrere Agenten gleichzeitig dieselbe Zelle belegen können.

Um in einem Modell Informationen zu speichern bzw. zu hinterlegen stehen *ValueLayer* zur Verfügung. Sie stellen einfache, beliebigdimensionale Arrays reeller Zahlen dar. Mit ihnen können z.B. Eigenschaften einer Fläche dargestellt werden. Die Autoren betonen wiederholt, dass Value-Layer auch für beliebige andere Zwecke verwendbar sind, ein Beispiel bleiben sie jedoch schuldig. Ein Value-Layer kann dabei einem Kontext hinzugefügt werden und mit beliebig vielen Projektionen verknüpft werden.

Die so über das Modell verteilten Informationen können von den einzelnen Komponenten gegenseitig abgefragt werden. Dazu dienen spezielle *Query-Objects*, die einer Anfrage die nötige, standardisierte Form geben. In [HCN+06] wird etwas näher darauf eingegangen, wie *Kontext*, *Projektion* und *Query-Objekte* zusammenhängen.

Ist ein Modell einmal entworfen und implementiert, so wird es in die Repast Simphony Runtime Engine geladen. Diese führt dann runden- bzw. tickbasiert die Simulation durch. Dabei wird in jeder Runde die Step-Methode eines jeden Agenten aufgerufen. In dieser ist das eigentliche Agentenverhalten spezifiziert.

4.4.2 Bewertung

Ursprünglich war geplant, das vorgestellte Modell mit Repast Simphony umzusetzen. Nach einigen Tests sprachen jedoch mehrere Gründe dagegen. So war lange Zeit nur eine unfertige Beta-Version verfügbar. Diese war nur nach erheblichem Debug-Aufwand, (siehe hierzu auch Anhang C) lauffähig. Zudem ist die verfügbare Dokumentation unzulänglich, was, in Kombination mit der hohen Komplexität des Frameworks, zu beträchtlichen Schwierigkeiten und Verzögerungen bei der Arbeit führte. Ebenfalls dieser Komplexität ist vermutlich die geringe Geschwindigkeit geschuldet, mit der eine Simulation letztendlich läuft. Selbst eine einfache Simulation mit sehr wenigen Agenten, die zudem sehr einfach gestrickt war (zufälliges Herumlaufen, keine weiteren Operationen), kommt nicht über ein Dutzend simulierter Ticks pro Sekunde hinaus. Zusätzlich ist die später verfügbare Version 1.0, obwohl in Ja-

va geschrieben, aufgrund von Softwarefehlern nur auf Windows Systemen lauffähig, Linux bleibt vorerst ebenso außen vor wie Mac OSX.

Aus den aufgeführten Gründen sprach vor allem die geringe Geschwindigkeit und die sehr schwierige Handhabung gegen eine Nutzung von Repast. Mit dem Wissen, das während des Studiums des Frameworks erlangt wurde, konnte jedoch in kurzer Zeit ein eigenes Simulationstool entwickelt werden. Dieses ist zwar bei weitem nicht so komplex, wie eines der genannten Tools, dafür aber sehr einfach in der Handhabung und vor allem sehr schnell. Es erhielt den Namen *MyOwnWorld*.

4.5 MyOwnWorld

Das für diese Arbeit entwickelte Simulationstool wurde so konzipiert, dass es genau die Merkmale des entworfenen Modells unterstützt. Dafür wurden vier Hauptklassen entwickelt, die im Folgenden kurz erklärt werden. Danach wird erläutert, wie die einzelnen Komponenten zusammenwirken und wie damit das besprochene Modell implementiert wurde.

4.5.1 Hauptklassen

Die Hauptklassen stellen den Kern der Simulationsumgebung dar. Es handelt sich um die Klassen *World*, *AbstractAgent*, *Scheduler* und *Statistics*. Zu diesen Hauptklassen kommen noch einige weitere Klassen, wie z.B. die Klasse *Log*, die verschiedene Methoden für die Ausgabe enthält, oder die Klasse *Parameters*, die die verwendeten Versuchsparameter und Methoden zum Laden und Speichern derselben enthält. Sie spielen jedoch eine untergeordnete Rolle und sollen hier keine weitere Beachtung finden. Die wichtigsten Klassen mit einer Auswahl ihrer jeweiligen Methoden sind auf Abbildung 4.3 zusammengefasst.

Interessanter und von größerer Bedeutung sind die Umsetzungen der einzelnen Agenten, auf die in Abschnitt 4.6 näher eingegangen wird.

World

Die Klasse World repräsentiert die Welt in der sich die Agenten bewegen. Sie besteht, wie auch das Modell, aus drei Ebenen, (*Agent-Layer*, *Dirt-Layer* und *Pheromon-Layer*). Dabei besteht das Agent-Layer aus einem Array aus Verweisen auf einzelne Agenten-Instanzen, während die beiden anderen Layer aus einfachen Fließkomma-Arrays bestehen.

Die World-Klasse stellt verschiedene Methoden zur Verfügung, die die Agenten für ihre jeweiligen Aktionen nutzen können, so zum Beispiel die Methoden *clean* (reinigen einer Zelle), *moveRelative* (bewegen, Angaben relativ zum aktuellen Standort), *mark* (markieren einer Zelle mit Pheromonen) oder *getSense* (liefert die aktuelle Umgebung eines Agenten).

Da ein sinnvolles Arbeiten mit einer Simulationsumgebung nur möglich ist, wenn man aussagekräftige Ergebnisse bzw. Werte der simulierten Gegebenheiten erhält,

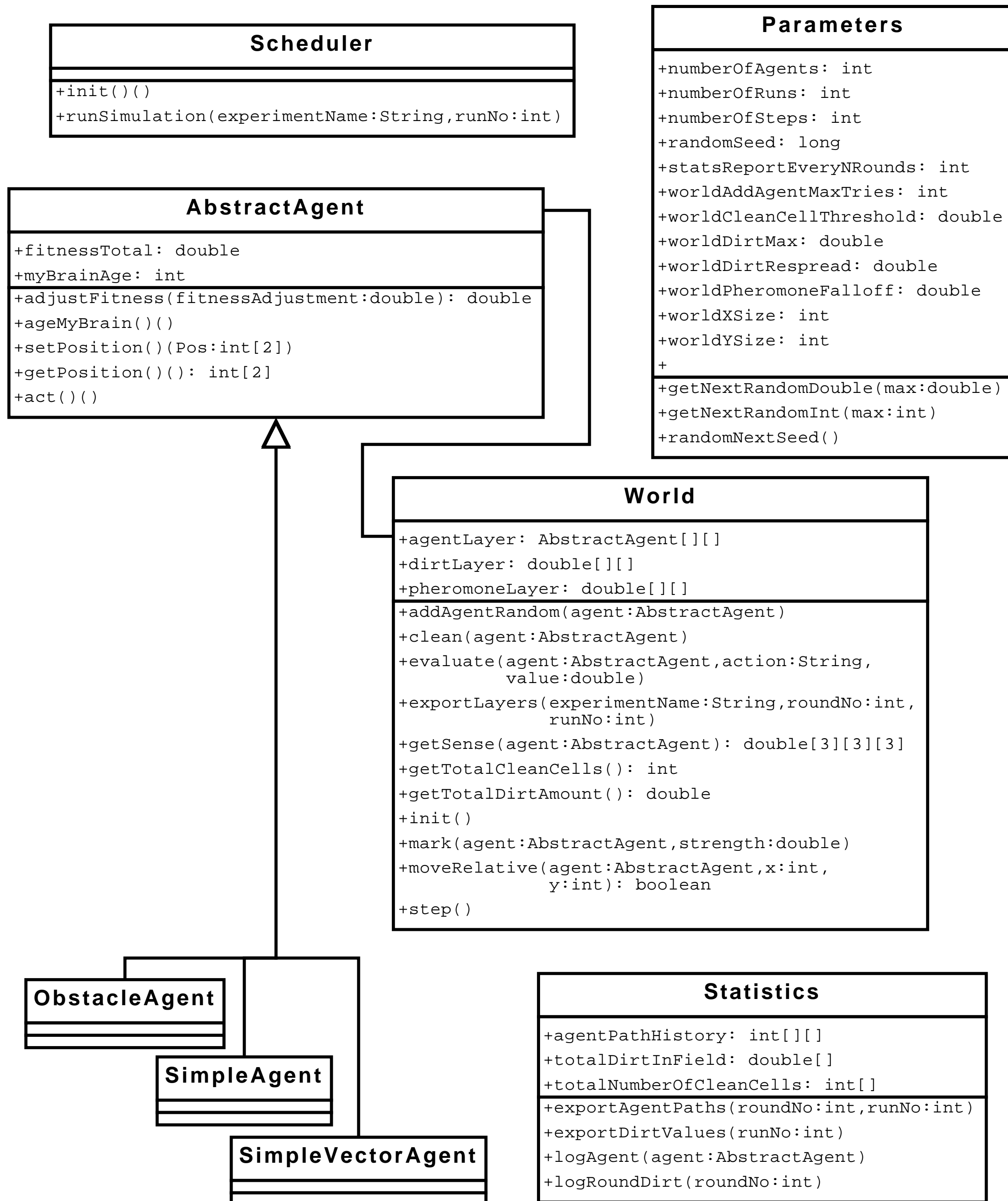


Abbildung 4.3: Eine Übersicht über die Hauptklassen der entwickelten Simulationsumgebung *MyOwnWorld* mit den wichtigsten Parametern und Funktionen.

ist es notwendig, diese zu erfassen. Dazu stellt die World-Klasse einige Methoden zur Verfügung. So können die aktuellen Belegungen des Dirt- und des Pheromon-Layers als Bitmap exportiert werden.

Außerdem enthält sie die Methode *evaluate*. Sie wird bei jeder Aktion eines Agenten automatisch aufgerufen und bewertet diese im Gesamtkontext. Somit kommt der evaluate-Methode eine zentrale Bedeutung in der Simulationsumgebung zu, stellt sie doch die Fitness-Funktion der Simulation dar. Die Bewertungsfunktion ordnet jeder Aktion eines Agenten eine kontextabhängige Punktzahl zu. Dabei werden im einzelnen die folgenden Ereignisse bewertet (die Bewertung wurde in einigen späteren Experimenten variiert, näher hierzu findet sich in Kapitel 5):

move Um eine möglichst große Fläche abzudecken, ist Bewegung unerlässlich. Deshalb gibt es für jeden erfolgreichen Schritt **0.1 Punkte**. Ein erfolgreicher Schritt ist ein Schritt, der ausgeführt wird, ohne dass es dabei zu einer Kollision mit einer Wand, einem anderen Agenten oder einem Hindernis kommt.

clean Das grundsätzliche Ziel ist eine saubere Fläche und der einzige Weg dorthin führt über das Reinigen. Deshalb wird der Reinigungsversuch als solcher belohnt.

Um aber Agenten nach tatsächlicher Leistung zu belohnen, ist es angebracht, ihnen zusätzliche Bonuspunkte proportional zur aufgesammelten Schmutzmenge zu geben. Je stärker dieser Bonus ausfällt, umso größer ist der evolutionäre Reiz, sich darum zu bemühen, möglichst viel Schmutz aufzusammeln.

An diesen Gedanken knüpft auch die Idee an, nur dann Punkte zu vergeben, wenn die Zelle tatsächlich auch schmutzig ist, ihr Verschmutzungsgrad also oberhalb des *worldCleanCellThreshold* liegt.

Alles in allem gibt es also für jeden Reinigungsversuch **0.001 Punkte** und einen zusätzlichen Bonus von [***Schmutz***] **Punkten**, abhängig von der tatsächlichen Menge aufgesammelten Schmutzes. Reinigt ein Agent also eine Zelle, deren Verschmutzungsgrad 2.5 beträgt, so erhält er dafür 2.501 Punkte.

leave uncleaned Bewegt sich ein Agent von einer Zelle weg, ohne sie gereinigt zu haben, so wird er für dieses Verhalten mit **-2 Punkten** bestraft.

collison Eine Kollision ist die denkbar ineffizienteste Aktion die ein Agent versuchen kann. Sie wird deshalb mit **-10 Punkten** bestraft.

set marker Die Kommunikation der Agenten über Pheromonspuren soll gefördert werden. Deshalb wird jedes Setzen einer Marke (unabhängig von dessen Stärke) mit **1 Punkt** belohnt.

Die erwähnten Punkte werden pro Aktion vergeben und aufaddiert. Ein Zähler registriert dabei, seit wie vielen Runden diese Summe bereits schon gebildet wird.

So ist es möglich, jederzeit die durchschnittliche Fitness einer bestimmten Strategie zu bewerten. Es ergibt sich also:

$$Fitness_{Durschschnitt} = \frac{\sum_{i=1}^{Zaehler} p_{Aktionsbewertung_i}}{Zaehler}$$

Durch einfaches Zurücksetzen des Zählers ist es möglich, dass ein Agent auch während einer laufenden Simulation seine Strategie ändert und der Fitnesswert trotzdem immer die Bewertung der aktuell genutzten Strategie repräsentiert, ohne noch die Bewertung vorangegangener Aktionen mit einfließen zu lassen. Dies wird später beim Evolvieren einer Strategie wichtig. Der so berechnete Fitnesswert kann für jeden Agenten abgefragt werden und dient dem Evolutionsalgorithmus als Bewertungsgrundlage. Dabei kann der betrachtete Bewertungszeitraum nahezu beliebig gewählt werden, da ein Gesamtverhalten – repräsentiert durch das Neuronale Netz – bewertet wird und nicht, wie bei anderen evolutionären Ansätzen, jeder einzelne Schritt.

Abschließend enthält die World-Klasse noch die *step*-Methode. Diese lässt die simulierte Welt einen Schritt altern. Dabei werden, je nach Konfiguration, die Felder des Dirt-Layer weiter verschmutzt und die Pheromonspuren abgeschwächt.

AbstractAgent

Die Klasse AbstractAgent ist als abstrakte Klasse implementiert. Sie enthält alle Funktionen, die ein Agent haben muss, um in der Simulationsumgebung berücksichtigt zu werden. Alle tatsächlichen Implementierungen von Agenten erben von dieser Klasse. Mehr zu den einzelnen Agenten findet sich unter Abschnitt 4.6.

Scheduler

Die Klasse Scheduler stellt die Hauptklasse unter den Hauptklassen dar. Sie initialisiert die Welt vor der Benutzung, sie sorgt dafür, dass in jeder Runde alle Agenten zum Zug kommen und sie sorgt dafür, dass alle statistischen Daten erfasst und am Ende der Simulation ausgegeben werden.

Statistics

Diese Klasse verwaltet alle anfallenden statistischen Daten. So werden z.B. alle n-Runden die Fitnesswerte aller Agenten protokolliert, der Verschmutzungsgrad der Welt wird registriert und die jeweilige Position der Agenten wird vermerkt.

Die so gespeicherten Daten werden am Ende eines Simulationslaufes aufbereitet (es werden die Maximal-, Minimal- und die Durchschnittswerte ermittelt) und in Dateien geschrieben. Die Pfadkarten der Agenten können alle n-Runden bzw. am Ende eines Simulationslaufes als Bitmap ausgeben werden.

4.6 Agenten

Die einzelnen Agenten stellen mit ihren verschiedenen Verhaltensweisen das Kernstück der Simulation dar. Für die durchgeführten Versuche wurden verschiedene Implementierungen der abstrakten Klasse *AbstractAgent* entwickelt. Sie sollen im Folgenden kurz vorgestellt werden.

4.6.1 ObstacleAgent

Dem Obstacle-Agent kommt in gewisser Weise eine Sonderrolle unter den Agenten zu. Er dient der Repräsentation von Hindernissen auf der Karte und hat nur die Aufgabe, eine Zelle zu besetzen und so für andere Agenten unpassierbar zu machen. Da er keine eigenen Handlungen ausführt, wird er vom Scheduler auch nicht berücksichtigt.

4.6.2 SimpleAgent

Der Simple-Agent dient zum Testen der Simulationsumgebung. Er hat keine Strategie im eigentlichen Sinne, sondern bewegt sich zufällig über die Karte. Dabei reinigt er bei jedem Zug seine aktuelle Zelle, markiert sie mit der Stärke 1 und bewegt sich dann einen Schritt in eine zufällige Richtung. Dabei ignoriert er seine Umgebung völlig und nimmt so auch Kollisionen in Kauf.

4.6.3 VectorAgent

Der Vector-Agent wurde entwickelt, um das Konzept der vektorisierten Werteübergabe, die später mit Einführung der Neuronalen Netze zum Einsatz kommt, zu testen. Dazu bestimmt er sechs Zufallswerte (den zukünftigen Ausgabevektor des Neuronalen Netzes) und ermittelt dann daraus die auszuführende Aktion. Auch der Vector-Agent nimmt seine Umwelt nicht wahr. Da er, genau wie auch die NEAT-Agenten, zu einer Zeit immer nur *entweder* reinigen *oder* sich bewegen kann, kann er als Vergleichsrefferenz für die Leistung einer rein zufälligen Verhaltensstrategie benutzt werden.

4.6.4 OwnAgent

Der Own-Agent hat eine feste Strategie, nach der er sich verhält. Das Ziel bei seiner Entwicklung war, einen Referenzagenten zu erstellen, an dem sich Agenten mit evolviertem Verhalten messen können. Das Verhalten dieses Agenten soll das Verhalten eines Menschen widerspiegeln, der sich so intelligent verhält, wie es bei der gegebenen begrenzten Wahrnehmung der Umwelt möglich ist.

Dazu überprüft er zunächst die Zelle, in der er sich gerade befindet, auf Sauberkeit. Falls sie als schmutzig eingestuft wird, so reinigt er sie und hinterlässt einen

Marker der Stärke 0.5. Sollte die Zelle jedoch nicht (bzw. nur geringfügig) verschmutzt sein, so wählt er diejenige Zelle aus seiner Von-Neumann-Nachbarschaft (vier angrenzende Felder) aus, die die höchste Verschmutzung aufweist. Sollte auch diese Verschmutzung noch unterhalb des globalen Sauberkeitsgrenzwertes (*CleanCellThreshold*) liegen, so beschließt er, dass er sich momentan in einer bereits gesäuberten Umgebung befindet. Deshalb ist es für ihn angebracht, sich in eine Richtung zu bewegen, die bisher wenig bearbeitet wurde. Dies ist ableitbar aus der Stärke der Pheromone in seiner Umgebung. Er wählt also (wiederum aus seiner Von-Neumann-Nachbarschaft) die Zelle mit der schwächsten Pheromonmarkierung.

Nachdem auf diese Weise ein neues Ziel festgelegt wurde, markiert er seine aktuelle (alte) Zelle mit einer Stärke von 0.5 und begibt sich dann auf seine neue Zelle. Bei der Wahl seiner Zellen vermeidet der Own-Agent eine Kollision mit Wänden, Hindernissen oder anderen Agenten.

Dieses Verhalten sollte nach menschlichem Ermessen eine relativ gleichmäßige Reinigung der Fläche gewährleisten, wobei der Schwerpunkt darauf liegt, möglichst viel Schmutz aufzusammeln.

4.6.5 NeatAgent

Der NEAT-Agent repräsentiert den lernenden Agenten. Sein Verhalten ist nicht vorbestimmt, sondern wird von einem neuronalen Netz erzeugt. Dieses Netz wird durch den in Kapitel 3.5 erwähnten NEAT-Algorithmus evolutionär optimiert.

Der Neat-Agent nimmt seine Umgebung wahr und leitet diese Wahrnehmung als Eingabevektor an sein neuronales Netz weiter. Diese Wahrnehmung besteht aus je einer 3×3 Matrix von Werten pro Umwelt-Layer, der resultierende Eingabevektor ist also 27-dimensional. Dann wird die Ausgabe des Netzes berechnet und als 6-dimensionaler Ausgabe- oder Handlungsvektor zurückgeliefert. Abbildung 4.4 veranschaulicht dies. Genau wie der Vector-Agent bestimmt der NEAT-Agent den größten Wert dieses Handlungsvektors und führt die so bestimmte Aktion aus.

Der NEAT-Agent handelt nach seiner, durch das KNN repräsentierten, Strategie für eine gewisse Anzahl von Schritten (im Folgenden auch Bewertungszeitraum oder Generationszeit genannt). In dieser Zeit summiert er alle Fitnesswerte, die er für die einzelnen Schritte erhält, auf. Am Ende eines Bewertungszeitraums wird diese Summe durch die Bewertungszeit geteilt, so dass man die durchschnittliche Bewertung pro Schritt erhält. Auf Basis dieses Fitnesswertes erzeugt der NEAT-Algoritmus dann eine neue Generation von neuronalen Netzen. Diese werden willkürlich auf die vorhandenen Agenten verteilt und ersetzen die dort vorhandenen KNN. Mit der Zuordnung eines neuen KNN wird der Fitnesswert des Agenten auf Null zurückgesetzt und die Bewertung beginnt von Neuem.

Während der NEAT-Agent ursprünglich auch die Stärke der zu setzenden Marker selbst bestimmen konnte, wurde dies später durch ein festes Verhalten ersetzt. Danach verhält sich der NEAT-Agent, was die Marker betrifft, genau wie der Own-Agent und markiert jede Zelle, die er reinigt, mit einer Stärke von 0.5 und jede Zelle, die er verlässt, ebenfalls mit einer Stärke von 0.5.

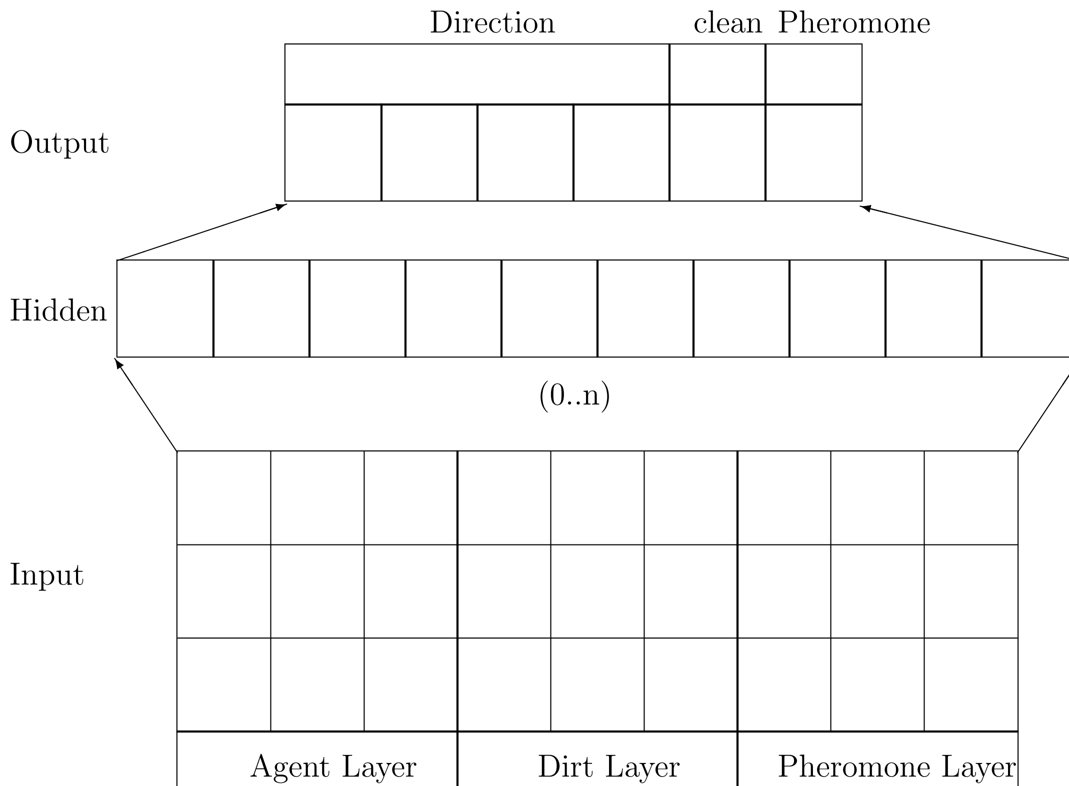


Abbildung 4.4: Das Neuronale Netz, welches den NEAT-Agenten steuert, besteht aus 27 (3×3×3) Eingabeneuronen (Sensoren) und 6 Ausgabeneuronen. Dazwischen können sich beliebig viele Neuronen im *Hidden-Layer* befinden. Die Verknüpfungen zwischen den einzelnen Neuronen können ebenfalls beliebig sein. Sie werden, genau wie die Neuronen des Hidden-Layers, vom NEAT-Algorithmus evolutionär entwickelt.

4.6.6 SmartNeatAgent

Im Laufe der Experimente stellte sich heraus, dass die Kodierung der ursprünglichen NEAT-Agenten nicht geeignet für eine Evolution durch den NEAT-Algorithmus ist (näheres hierzu in Abschnitt A.10). Deshalb wurde sein Design abgeändert, der neu entstandene Agent wird SmartNeatAgent genannt.

Der Hauptunterschied zum NEAT-Agent besteht darin, dass die Bewegungsrichtung nicht mehr fest an die Struktur des KNN gebunden ist, sondern eine Entscheidung unabhängig von der tatsächlichen Richtung trifft. Dies wird realisiert, indem dem KNN jede einzelne der vier Richtungen getrennt zur Bewertung vorgelegt wird (vgl. Abbildung 4.5). Dabei nimmt er jeweils auch nicht mehr die gesamte 3×3 Matrix wahr, sondern bei jeder Betrachtung nur eine 3×2 Matrix, er sieht also bei jeder Bewertung nur „nach vorn". Aus den vier Einzelentscheidungen wählt der SmartNeatAgent dann diejenige aus, die vom Netz am häufigsten gewählt wurde.

Vom SmartNeat-Agent wurde zudem noch eine Variante namens SmartSimpleNeatAgent entwickelt. Der Unterschied zum SmartNeatAgent besteht darin, dass der

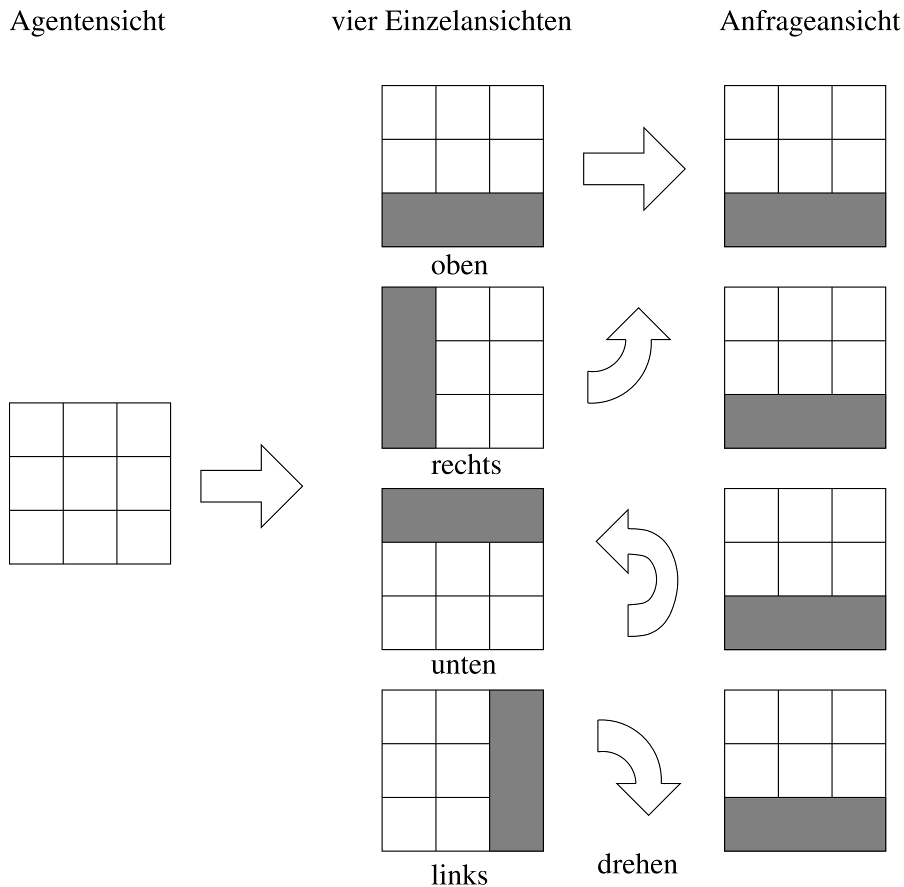


Abbildung 4.5: Der SmartNeat-Agent fällt vier einzelne Entscheidungen. Dabei wird seine Umgebung jeweils so gedreht, dass sein KNN die Situation objektiv und ohne Einfluss einer bestimmten Richtung oder Orientierung bewerten kann.

SmartSimpleNeatAgent nur das Dirt-Layer wahrnimmt, die anderen beiden Layer werden von ihm ignoriert. Ziel dieser Variante ist zu prüfen, ob die Pheromoninformationen einen tatsächlichen Vorteil erbringen oder eher ablenken bzw. verwirren und so den Lernprozess stören.

Kapitel 5

Experimente & Ergebnisse

Je vertrauter und alltäglicher eine Verhaltensweise ist, desto problematischer wird ihre Analyse.

Desmond Morris

Der beschriebene Simulator wurde benutzt, um eine ganze Reihe von Versuchen durchzuführen. Im folgenden Kapitel sollen einige davon näher betrachtet und dabei auch Besonderheiten der Ergebnisse geklärt werden. Eine größere Auswahl durchgeführter Versuche findet sich im Versuchslogbuch in Anhang A.

Eine Kernfrage bei der Durchführung der Versuche war die Klärung der Leistungsfähigkeit und Effizienz der NEAT-Agenten. Zunächst musste gezeigt werden, dass sie in der Lage sind, eine bessere Leistung als eine zufallsbestimmte Strategie, in diesem Fall die der SimpleVector-Agenten, zu erbringen. Auf der anderen Seite ist auch der Vergleich mit der nahezu perfekten Strategie der Own-Agenten von Interesse.

Bevor verglichen werden kann, müssen die Simulationsparameter auf das Modell abgestimmt werden. Die verfügbaren Parameter teilen sich in zwei Gruppen. Die erste Gruppe stellen die allgemeinen Simulationsparameter dar. Darin enthalten sind Eigenschaften wie die Anzahl der Agenten, die Feldgröße und der Verschmutzungsgrad. Dem gegenüber stehen die NEAT-Parameter. Sie enthalten z.B. die Wahrscheinlichkeiten der einzelnen Mutationen. In der folgenden Aufstellung sind die Parameter näher erklärt, von denen die größten Einflüsse auf die Simulationsergebnisse zu erwarten sind:

Spielfeldgröße Die Spielfeldgröße hat einen Einfluss auf viele Eigenschaften der Simulation. Zum einen erhöht sich die Rechenzeit und der Speicheraufwand der Simulation mit steigender Spielfeldgröße (in jeder Runde müssen alle Felder neu verschmutzt werden; die Pheromonspuren auf allen Feldern abgeschwächt werden). Zum anderen erhalten die Agenten mehr Bewegungsfreiraum. Die Wahrscheinlichkeit einer Kollision mit einer Wand sinkt bei einem größeren Spielfeld, ebenso die Wahrscheinlichkeit einer Begegnung zweier

Agenten. Dafür steigt die insgesamt zu verrichtende Arbeit, da die Verschmutzung zellenabhängig ist.

Zahl der Agenten Die Anzahl der Agenten, die sich gleichzeitig auf dem Spielfeld bewegen, hat einen großen Einfluss auf die Gesamtreinigungsleistung[1]. Werden zu wenig Agenten eingesetzt, können sie die Fläche nur ungenügend abdecken, sind es zu viele, so behindern sie sich gegenseitig.

Diese Zahl hängt natürlich auch von der Strategie der Agenten ab, je besser ihre Koordination ist, umso mehr können auch gleichzeitig eingesetzt werden.

Verschmutzungsgrad/Verschmutzungsrate Der anfängliche Verschmutzungsgrad kann das Verhalten der Agenten ebenso beeinflussen wie die Verschmutzungsrate, mit der die Zellen nach jeder Runde wieder neu verschmutzt werden. Ist die Wiederverschmutzung zu hoch, so haben die Agenten keine Chance, das Feld zu säubern bzw. es sauber zu halten, ist sie zu gering, so gibt es nach einiger Zeit keine schmutzigen Felder mehr und die Entwicklung der Agenten bzw. deren Verhalten stagniert.

Zahl der Bewertungsschritte Dieser Parameter ist nur dann relevant, wenn es sich um ein NEAT-Experiment handelt, also tatsächlich NEAT-Agenten und keine Agenten mit fester Strategie zum Einsatz kommen. Er bestimmt die Anzahl der Schritte, die ein Agent mit einer Strategie handelt, bevor es zu einem neuen Evolutionsschritt kommt. Ist diese Zeit zu kurz gewählt, so hat die betroffene Strategie keine Zeit, sich zu entfalten bzw. ihre Vorteile auszuspielen und somit durch Hinzugewinnen von Fitness-Punkten ihre Überlegenheit gegenüber anderen Strategien zu beweisen. Ist der Bewertungszeitraum hingegen lang, dann wird zum Einen die Simulation unnötig in die Länge gezogen und zum Anderen besteht die Gefahr, dass eine relativ gute Strategie durch wiederholtes Fehlverhalten sehr viele negative Bewertungen ansammelt und daraufhin beim nächsten Evolutionsschritt aussortiert wird.

Pheromone Falloff Dieser Parameter beschreibt, wie stark sich eine Pheromonspur verflüchtigt. Sie stellt dabei einen Faktor dar, mit dem jede Pheromonmarkierung am Ende jeder Runde multipliziert wird. Somit wäre es theoretisch auch möglich, die Spuren nach jeder Runde zu verstärken. Die Auswirkung der Veränderung in der Stärke der Markierungen lässt sich nicht im Voraus abschätzen, besonders nicht, was ihren Einfluss auf die Situationsbewertungen durch die Neuronalen Netze angeht.

Compat*ibility* Threshold Dieser Grenzwert bestimmt, wie gleich bzw. verschieden zwei Genome sein müssen, um noch der gleichen Spezies zugeordnet zu werden. Um verschiedene Lösungsstrategien erforschen zu können, ist eine größere Zahl von Spezies (Artenvielfalt) vorteilhaft. Allerdings hat jede Spezies

[1] die Summe der Putzleistung aller einzelnen Agenten

auch weniger Mitglieder, je mehr es von ihnen gibt, so dass der speziesinterne Wettkampf nicht mehr gewährleistet ist.

Mutate (add-node/add-link/link-weights) Prob*ability* Die Wahrscheinlichkeit für die Add-Node-/Add-Connection-Mutation bzw. die Wahrscheinlichkeit für die Änderung einer Verbindungsgewichtung. Grundsätzlich gilt: Je höher eine Mutationsrate, umso schneller kann man sich einer potentiell richtigen Lösung nähern. Allerdings sinkt mit steigender Mutationsrate auch die Stabilität der gefundenen Lösung bzw. Näherung, so dass gute Ansätze sehr schnell wieder beschädigt werden und verloren gehen.

Nachdem so eine Reihe von Versuchen durchgeführt wurde, stellt sich die Frage nach der Bewertung der Ergebnisse. Für das entwickelte Modell interessant und relevant ist letzten endes nur die Zahl der gesäuberten bzw. sauber gehaltenen Zellen. Dieses Maß ist absolut und gleichbleibend. Würde man z.B. die durchschnittliche Fitness aller Agenten[2] zu einer bestimmten Zeit als Maßstab ansetzen, so ergäben sich Verzerrungen, da z.B. die Menge aufgesammelten Schmutzes in die Bewertung der Strategien mit einfließt und somit die sich ergebenden Fitnesswerte auch bei gleichen Strategien unterschiedlich ausfallen würden, wenn die Verschmutzungsrate der Versuchskonfiguration unterschiedlich eingestellt ist.

Aufgrund der großen Anzahl von Versuchen und der damit verbundenen hohen Anforderungen, was die Rechenkapazität angeht, wurde das JOSCHKA-Tool (vgl. [BTS05]) am Institut für Angewandte Informatik und Formale Beschreibungsverfahren (AIFB) genutzt. Es verteilt Rechenjobs auf freie Rechner eines Rechnerpools und trägt die Ergebnisse nach deren Beendigung wieder zusammen. So ließ sich eine größere Zahl von Versuchen in relativ kurzer Zeit durchführen.

Im Rahmen der vorliegenden Arbeit wurden sehr viele Versuche durchgeführt. Sie alle im einzelnen und ausführlich zu beschreiben würde den Leser langweilen und diese Arbeit unnötig aufblähen. In Anhang A ist eine Auswahl der durchgeführten Versuche beschrieben. Sie sind zu Versuchsreihen gebündelt, die hier kurz erklärt und zusammengefasst werden. Details dazu finden sich in den jeweils referenzierten Abschnitten im Anhang. Außerdem finden sich auf der CD-ROM, die in Anhang B zu finden ist, sämtliche Daten zu den durchgeführten Versuchen, inklusive aller Logfiles und Bilddateien.

In den „ersten Versuchsreihen“ (Abschnitt A.2) wird die Tauglichkeit der entwickelten Simulationsumgebung getestet und geprüft, wie viele Agenten mit einer zufallsbasierten Strategie sich gleichzeitig auf dem Feld bewegen können, ohne sich gegenseitig zu behindern. Eine solche Behinderung war bereits bei 50 Agenten deutlich erkennbar.

Die Versuchsreihe „Statistiktests“ (Abschnitt A.3) diente der Erprobung des Werteexports in die Logfiles. Es wurde ebenfalls untersucht, wie viele Simulationsläufe notwendig sind, um einen relativ stabilen Mittelwert zu erhalten. Ausge-

[2]Die Fitness eines jeden Agenten ergibt sich aus der Fitnessfunktion, die jeden seiner Züge bewertet, siehe Abschnitt 4.5.1 *evaluate*.

hend von den Ergebnissen wurden von da an (fast) alle Versuche mit 20 Durchläufen simuliert. Außerdem wurde auch der Einfluss von verschiedenen Populationsgrößen auf die Fitness-Entwicklung der NEAT-Agenten untersucht. Es zeigte sich, dass die Fitnesskurven über den Lauf der Simulation mit steigender Anzahl von Agenten glatter wurde.

In der Versuchsreihe „r02“ (Abschnitt A.5) wurde getestet, wie sich verschieden lange *Bewertungszeiträume*[3] auf die Entwicklung auswirken.

Die Versuchreihe „r04“ (Abschnitt A.6) diente, analog zu den ersten Versuchsreihen, der Überprüfung der Skalierbarkeit der entwickelten Strategien. Es wurde getestet, wie viele Agenten der Typen Own-Agent und Vector-Agent gleichzeitig ein Feld sauberhalten können, ohne sich dabei gegenseitig zu behindern. Eine solche Behinderung wurde ab 800 gleichzeitig eingesetzten Agenten deutlich sichtbar.

Versuchsreihe 25 (Abschnitt A.7) diente der Überprüfung des Einflusses der Fitnessfunktion. Es wurden verschiedene vereinfachte Fitnessfunktionen getestet. Dabei wurde bewiesen, dass es zwar sehr wohl einen Einfluss der Fitnessfunktion auf das Lernverhalten gibt und somit eine zufallsbasierte Fitnessfunktion nicht zu annehmbaren Ergebnissen führt. Die verschiedenen Ausprägungen der tatsächlichen Bewertung haben jedoch einen relativ kleinen Einfluss.

In Versuchsreihe 26 (Abschnitt A.9) wurde mit einem selbst entworfenen KNN experimentiert. So wurde es z.B. als Startbelegung für den NEAT-Algorithmus benutzt. Es zeigte sich, dass der NEAT-Algorithmus dieses nicht etwa verbesserte, sondern im Gegenteil, es so beschädigte, dass es seine ursprünglich gute Leistung nicht mehr erbringen konnte.

Während der Versuchsreihe 28 (Abschnitt A.10) wurde darauf aufbauend der SmartNeat-Agent entwickelt. Damit sollte die Abhängigkeit von Symmetrien im zu erlernenden KNN verringert werden. Es zeigte sich, dass der SmartNeat-Agent tatsächlich bessere Resultate erbrachte, als der ursprüngliche NEAT-Agent. Außerdem wurde festgestellt, dass die Größe des im Laufe der Evolution erzeugten Genoms trotz relativ konstanter Leistung immer weiter wächst.

Schließlich wurde der neu entwickelte SmartNeat-Agent in Versuchsreihe 29 (Abschnitt A.12) dahingehend untersucht, ob das von ihm erlernte Verhalten nach einem Ende der Evolution auch davon losgelöst gute Resultate erbringen könnte. Es zeigte sich, dass dies nicht der Fall war. Sobald der Evolutionsprozess ausgesetzt wurde, sank die erbrachte Leistung auf einen Bruchteil der ursprünglichen Leistung ab.

5.1 Einleitende Versuchsreihen

Die ersten Versuchsreihen dienten dazu, das entwickelte System auszuloten, eventuelle Schwachstellen aufzudecken und ein Gefühl für die Zusammenhänge der einzelnen Parameter zu erhalten.

[3]Als Bewertungszeitraum, auch Generationszeit, wird die Anzahl von Schritten verstanden, in der eine Strategie erprobt wird, bevor sie von der nächsten Generation ersetzt wird.

So stellte sich schnell heraus, dass eine Spielfeldgröße von 100×100 Zellen für die durchzuführenden Versuche geeignet ist. Ebenso zeigte sich, dass eine verrauschte Initialverschmutzung schlechte Ergebnisse lieferte. Es wurde vermutet, dass die zufällige Verschmutzung zu viel Unruhe in den Lernprozess brachte bzw. aus zu chaotischen Werten keine gute Strategie abstrahiert werden konnte. Deshalb wurde bei späteren Versuchen das Spielfeld initial gleichmäßig verschmutzt.

Ebenfalls aufgegeben werden musste die Vorstellung, dass sowohl die Nutzung als auch das Setzen der Pheromonmarkierungen (innerhalb der gegebenen Zahl von Schritten bzw. mit der genutzten Fitnessfunktion) durch die NEAT-Agenten gelernt werden kann. Auch nach längeren Läufen (15.000 Schritte) war noch kein System in der Markierungsstrategie zu erkennen. Deshalb wurde die Markierungsstrategie der NEAT-Agenten auf die gleiche Weise wie die der Own-Agenten festgelegt.

Zur Zahl der Agenten wurden Versuche mit den SimpleVector-Agenten unternommen. Da sie weder Strategie noch Koordination besitzen, stellen sie ein Maß dafür dar, wie viele Agenten sich gleichzeitig über das Spielfeld bewegen können, ohne sich gegenseitig zu behindern. Wie sich zeigte, begann eine signifikante gegenseitige Störung bei über 50 Agenten. Deshalb wurden für die meisten nachfolgenden Versuchsreihen 50 Agenten benutzt.

Die Zahl der Bewertungsschritte festzulegen war in sofern schwierig, als dass sich hier verschiedene Faktoren gegenseitig überschnitten und so ein direktes Vergleichen nicht möglich war. Da eine Simulation immer eine festgelegte Anzahl von Schritten läuft, hängt die Anzahl der NEAT-Generationen vom Quotienten aus Simulationsschritten und Bewertungsschritten ab. Vergleicht man also beispielsweise Bewertungszeiträume von jeweils 10 Schritten und simuliert so 5000 Schritte, dann ergeben sich 500 Generationen. Lässt man nun das gleiche Experiment mit Bewertungszeiträumen von nur 2 Schritten laufen, so ergeben sich 2500 Generationen. Man kann erwarten, dass eine Strategie nach 2500 Generationen wesentlich ausgereifter ist als nach 500 Generationen (falls sie nicht bereits früher konvergiert). So müssen also die Vergleichsversuche unterschiedlich lange Gesamtlaufzeiten haben. Aus den gemachten Versuchen kristallisierte sich zunächst ein Bewertungszeitraum von 2-3 Schritten als günstig heraus.

Dabei war auch festzustellen, dass der größte Teil der Rechenzeit auf den Evolutionsteil der Simulation abfällt und somit Versuche mit kürzeren Bewertungszeiträumen wesentlich längere Laufzeiten hatten als solche mit längeren Bewertungsintervallen.

Eine weitere Entdeckung, die im Laufe der zahlreichen Versuche gemacht wurde, ist das einheitliche Verhalten der NEAT-Agenten. Ihr Verhalten scheint zu allergrößten Teilen von der aktuellen Konfiguration ihrer Neuronalen Netze abzuhängen und erst in zweiter Linie von ihrer lokalen Umgebung. Dies war daran zu erkennen, dass sich die 50 NEAT-Agenten in eine einheitliche Richtung bewegten, egal an welcher Stelle der Karte sie sich gerade befanden (wenn sie nicht gerade auf ein Hindernis stießen). Dieses Verhalten änderte sich vor allem immer dann, wenn den Agenten im Zuge einer neuen Generation neue neuronale Netze zugewiesen wurden. Verwunderlich hieran ist vor allem die Tatsache, dass sich die Agenten einheitlich

verhielten, obwohl die Population durchschnittlich in 7 Spezies unterteilt war, die sich (wie in Abschnitt 3.5.1 erläutert) eben genau durch ihre unterschiedlichen Netze unterscheiden.

5.2 OwnAgent

Nachdem in der Erprobungsphase eine ganze Reihe von Versuchsreihen mit den NEAT-Agenten durchgeführt wurden, in denen versucht wurde, die NEAT-Parameter zu optimieren, wird nun ein Vergleich zwischen NEAT-Agenten auf der einen Seite und Own-Agenten auf der anderen Seite durchgeführt. So soll festgestellt werden, ob NEAT-Agenten in ihrer Leistung an die Own-Agenten heranreichen können. Um dem Leser eine Vorstellung der zahlreich unternommenen Versuche zu vermitteln, wird diese Versuchsreihe hier detailliert erklärt.

5.2.1 Versuch Own-1500

Als erstes liefen 50 Own-Agenten in 20 Läufen jeweils 1500 Schritte lang über ein verschmutztes Feld. Dies sollte eine Referenz zum späteren Vergleichen der Ergebnisse liefern. Die genaue Konfiguration des Versuches ist der Tabelle 5.1 zu entnehmen. Es sei an dieser Stelle angemerkt, dass die *Namen* der Versuche (z.B. 24-Own-1500) lediglich zur Unterscheidung der anfallenden Daten dienen, sie stehen meist in keinem direkten Bezug zum Versuchsinhalt. Der Einfachheit halber wurden sie in den folgenden Beschreibungen beibehalten.

Tabelle 5.1: Simulationsparameter für Versuch 24-Own-1500

experimentName	24-Own-1500
numberOfSteps	1500
numberOfRuns	20
randomSeed	5
numberOfAgents	50
neatRoundsPerGeneration	5
isNeatExperiment	false
worldXSize	100
worldYSize	100
worldDirtMax	2.0
worldDirtRespread	0.1
worldCleanCellThreshold	1.0
worldPheromoneFalloff	0.0
statsReportImagesEveryNRounds	150
statsReportDataEveryNRounds	5

Ergebnisse

Nimmt man an, dass mögliche Hindernisse auf der Karte (sichtbar z.B. in der ersten Spalte der Abbildung 5.1) die Strategie der Own-Agenten nicht signifikant behindern, dann müssen die Ergebnisse aus Versuchen mit selbigen statistisch vorhersagbar sein. Im Folgenden werden deshalb zunächst die experimentellen Ergebnisse vorgestellt, danach werden diese anhand einiger einfacher Berechnungen relativiert. So wird die tatsächliche Vorhersagbarkeit gezeigt. Wie sich zeigt, ist die Reinigungsleistung eines einzelnen Own-Agenten nur von der Verschmutzungsrate und dem Sauberkeitsgrenzwert abhängig.

Wie bereits in Abschnitt 4.5.1 erwähnt, protokolliert die Simulationsumgebung *MyOwnWorld* nicht nur die reinen Messwerte, sondern kann auch Bilder der einzelnen Layer exportieren. Um dem Leser einen Eindruck von diesen Bildern zu vermitteln und um zu verdeutlichen, welchen Zusatzwert diese Bilder haben, ist auf den Abbildungen 5.1, 5.2 und 5.3 eine Auswahl dieser Bilder zu sehen. Ein Simulationslauf wird jeweils durch eine Zeile dargestellt, die einzelnen Bilder wurden im Abstand von je 150 Schritten aufgenommen. Auf den Verschmutzungskarten (Abb. 5.1) repräsentiert ein schwarzer Punkt eine saubere Zelle, ein weißer Punkt steht für die höchste Verschmutzung. Die Pfadkarten (Abb. 5.2) zeichnen auf jeweils einem Linienzug den Weg des betreffenden Agenten während der letzten 150 Schritte nach. Auf den Pheromonkarten (Abb. 5.3) schließlich bedeutet ein schwarzer Punkt ein unmarkiertes Feld, während ein weißer Punkt für ein Feld mit höchster Markierungsstärke steht. Die roten (Abb. 5.1 und 5.3) bzw. weißen (Abb. 5.2) Linien/Blöcke stellen Hindernisse dar.

Auf diesen Bilderserien kann man erkennen, warum es notwendig ist, die Experimente jeweils mehr als einmal zu simulieren. Bei jedem Lauf verhalten sich die Agenten etwas anders. Dies liegt an den unterschiedlichen Random-Seeds, die für jeden Lauf neu gewählt werden.

Die Plots der einzelnen Messwerte (z.B. Abb. 5.4) veranschaulichen das messbare Verhalten der Agenten, welches sich in drei Größen ausdrückt: *Fitness* (die durchschnittliche Fitness, die den Agenten durch dir Fitnessfunktion zugeordnete ist, z.B. Abb.5.4), *Saubere Zellen* (die Zahl der Zellen, die auf dem gesamten Feld als sauber gelten, z.B. Abb.5.5) und *Gesamtschmutz* (Die Summe aller Verschmutzungen über alle Zellen des Feldes, z.B. Abb.5.6).

Die dargestellten Messwerte sind immer die Durschschnittswerte über alle Simulationsläufe. Auf der x-Achse sind jeweils die Simulationsschritte, auf der y-Achse die Durschnittswerte aller Agenten über alle Simulationsläufe eines Versuchs abgebildet. Auf den Plots sind drei Kurven zu sehen. Aufgetragen wird jeweils sowohl der tatsächliche Durschnitt aller Durchschnittswerte über alle Läufe (einfache Linie, mitte) als auch die minimalen (gestrichelte Linie, unten) bzw. maximalen (dicke Linie, oben) Durchschnittswerte, die zu einem bestimmten Zeitpunkt während einem der betrachteten Läufe auftraten.

Bemerkenswert ist der glatte Verlauf der Fitnesskurve bei den Own-Agenten (Abb. 5.4). Da ihr Verhalten sich im Laufe der Simulation nicht ändert, kann sich ihr

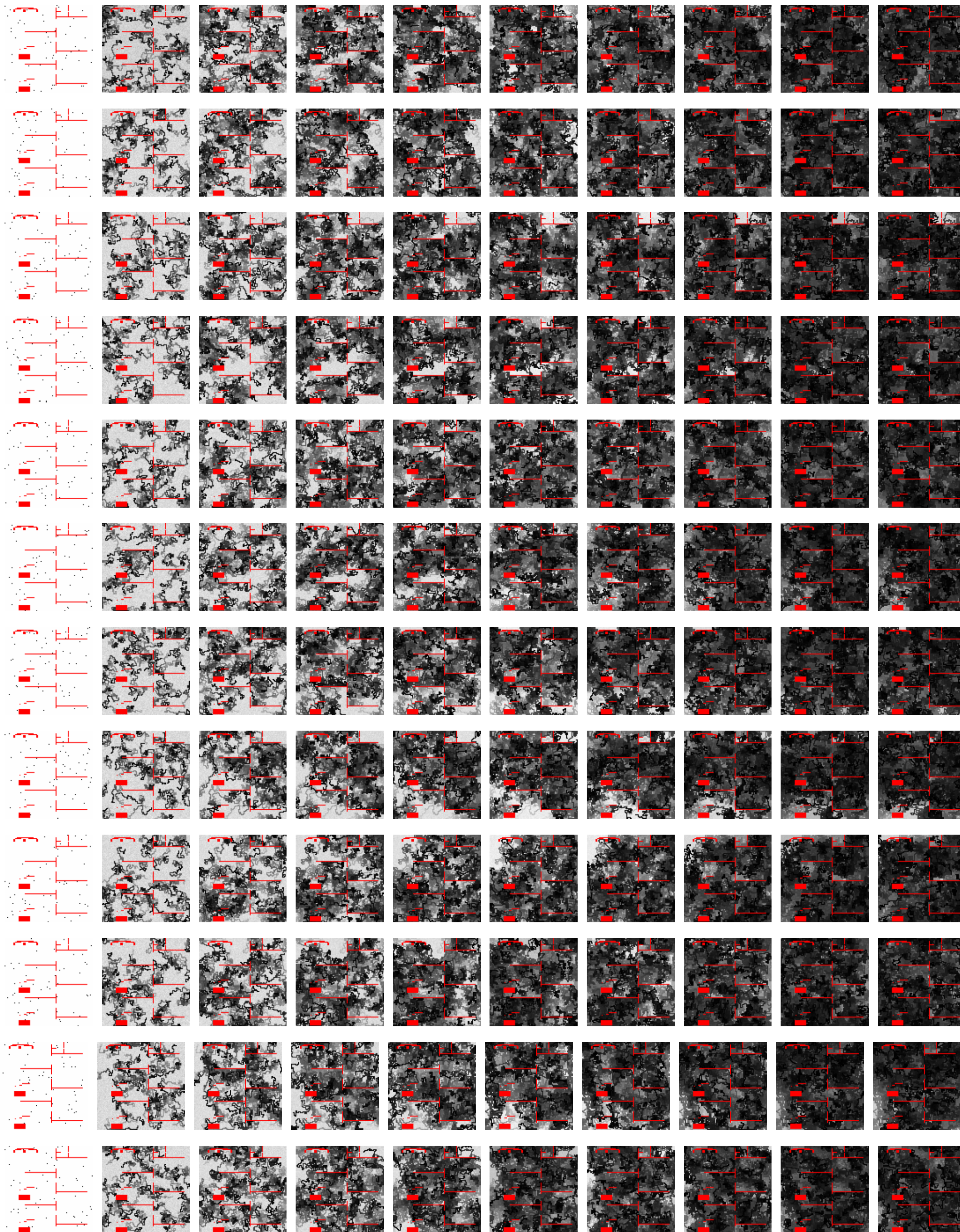


Abbildung 5.1: Experiment 24: 50 OwnAgents laufen 12×1500 Schritte lang über eine Karte mit Hindernissen. Man erkennt die leicht unterschiedlichen Verteilungen des Schmutzes, die durch die verschiedenen Random-Seeds bei jedem Lauf entstehen.

Abbildung 5.2: r02-Neat-2-5000: 50 NEAT-Agents laufen 12×5000 Schritte lang über eine Karte mit Hindernissen. Sichtbar sind die Spuren, der einzelnen Agenten. Die Bilder wurden jeweils im Abstand von 500 Schritten aufgenommen, eine Zeile mit ihren 10 Bildern repräsentiert folglich einen Lauf.
Es ist zu erkennen, wie sich bei manchen Läufen die Agenten auf einer Kartenseite konzentrieren, so z.B. in Zeile acht.

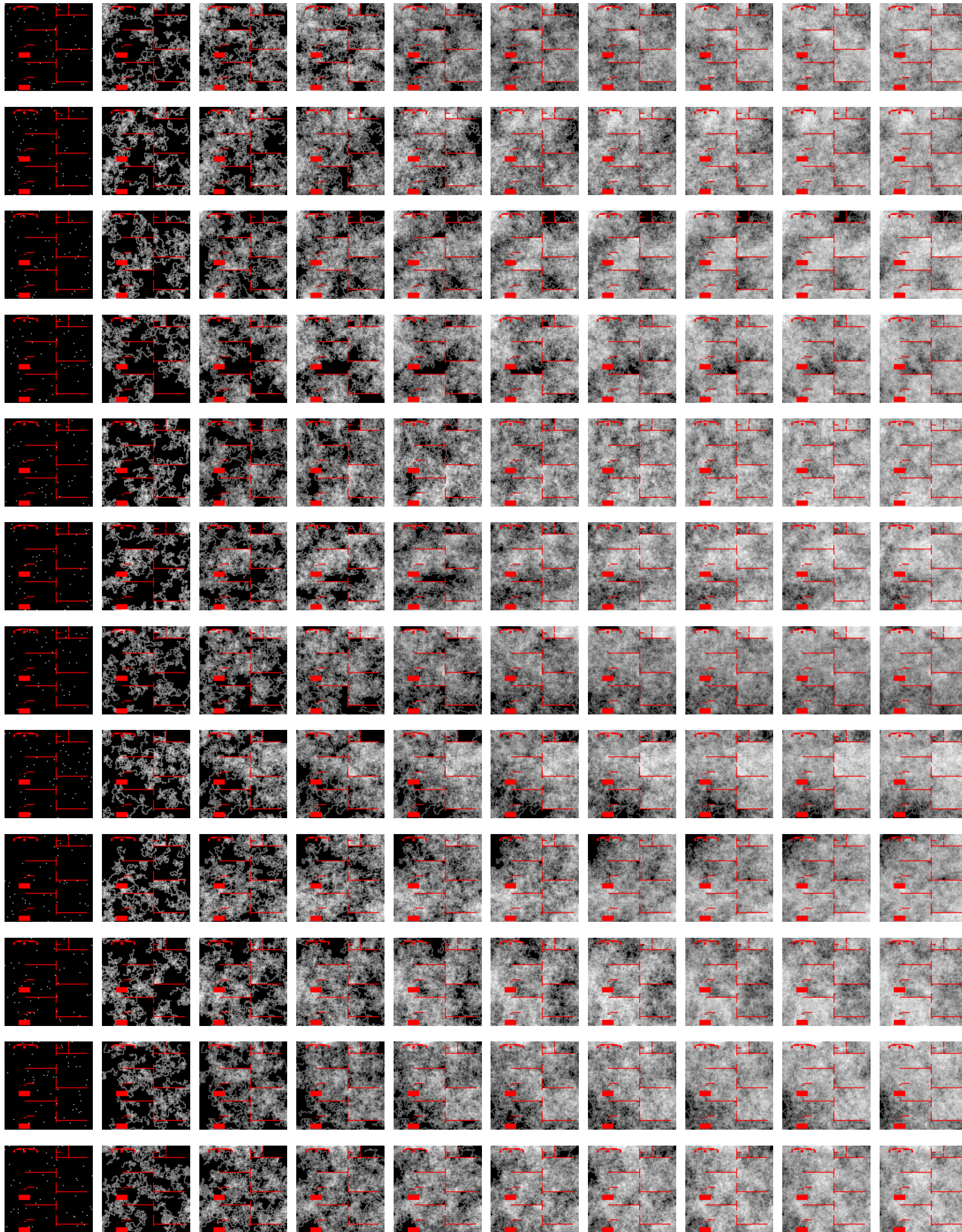


Abbildung 5.3: Experiment 24: 50 OwnAgents laufen 12×1500 Schritte lang über eine Karte mit Hindernissen. Erkennbar hier ist die unterschiedlichen Verteilungen der Pheromonspuren.

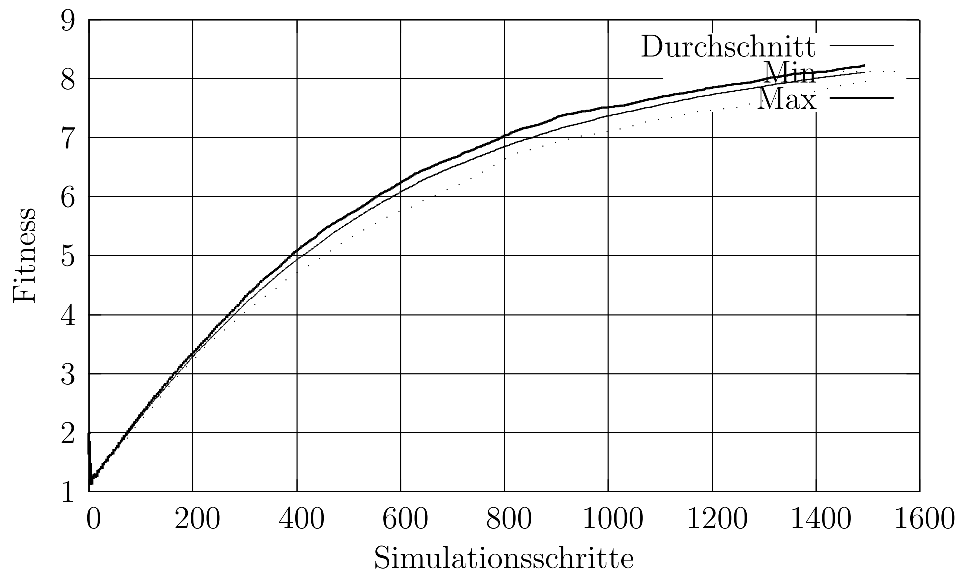


Abbildung 5.4: Experiment 24: Die Fitness der 50 OwnAgents konvergiert noch nicht innerhalb der ersten 1500 Simulationsschritte, es ist jedoch bereits absehbar, dass sie einen Wert von ca. 9 erreichen wird. In einem späteren Versuch wurde verifiziert, dass dies nach ca. 4000 Schritten tatsächlich der Fall ist.

Fitnesswert konstant entwickeln und enthält – im Gegensatz zu den NEAT-Agenten (z.B. Abb. A.10) – keine Sprünge.

Da der Own-Agent ein deterministisches Verhalten hat, pendelt sich die Anzahl an sauberen Zellen (Abb.5.5) auch bereits nach weniger als 25 Schritten auf einen Wert von ca. 520 ein. Dieser Wert stellt also die Zahl der Zellen dar, die (bei der gewählten Strategie mit dieser Anzahl von Agenten und dieser Verschmutzungsrate) von den Own-Agenten sauber gehalten werden kann.

Auch die Kurve 5.6, die die Gesamtschmutzmenge, die über die Karte verteilt ist, aufzeigt, konvergiert nach ca. 800 Schritten bei einem Wert von ca. 125.000.

Bei der verwendeten Wiederverschmutzungsrate von 0–0,1 und dem angewendeten Sauberkeitsgrenzwert von 1,0 dauert es im Schnitt

$$\frac{Sauberkeitsgrenzwert}{Wiederverschmutung_{max}/2} = \frac{1{,}0}{0{,}1/2} = \frac{1{,}0}{0{,}05} = 20$$

Schritte, bis eine Zelle wieder als schmutzig gewertet wird. Bei einer Gesamtanzahl von 100×100 Zellen werden also pro Zug

$$\frac{100 \times 100}{20} = 500$$

Zellen wieder schmutzig.

Geht man von den gemessenen 520 sauber gehaltenen Zellen aus, so hält ein einzelner Own-Agent im Schnitt

$$\frac{520\ Zellen}{50\ Agenten} = 10{,}4\ Zellen/Agent$$

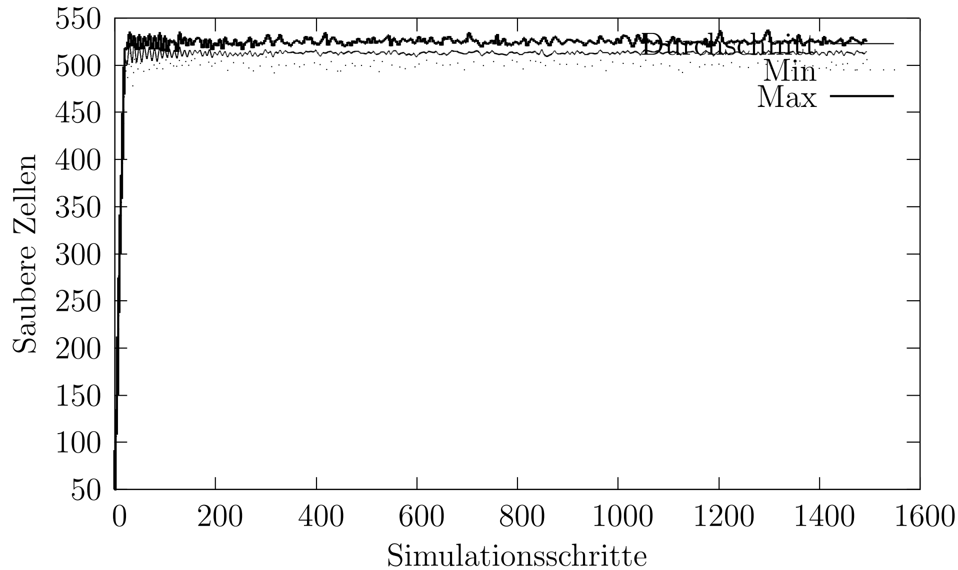


Abbildung 5.5: Experiment 24: Die Anzahl der von 50 OwnAgents gesäuberten Zellen konvergiert bereits nach ca. 25 Schritten bei einem Wert von ca. 520.

sauber. Da jede dieser Zellen 20 Schritte braucht, um wieder zu verschmutzen, kommen die Own-Agents also auf

$$\frac{10{,}4}{20} = 0{,}52$$

gereinigte Zellen pro Schritt.

Wenn man davon ausgeht, dass eine optimale Strategie darin besteht, immer eine schmutzige Zelle zu reinigen und sich dann zur nächsten schmutzigen Zelle zu bewegen, dann kommt man im Durchschnitt auf 0,5 gereinigte Zellen pro Schritt. Die tatsächliche Leistung des Own-Agent liegt somit leicht über dem theoretisch errechneten Wert.

Die Diskrepanz zwischen dem theoretisch errechneten Wert und der tatsächlichen Leistung wurde in weiteren Versuchen näher untersucht. Dabei stellte sich heraus, dass sie durch eine leichte Verzerrung während der Anfangsphase eines Experiments entsteht. Eine Zelle braucht in der genannten Versuchsanordnung (DirtRespread=0,1, CleanCellThreshold=1,0) mindestens 10 Runden, um wieder schmutzig zu werden. Tatsächlich dauert es jedoch im Schnitt nicht nur die Mindestanzahl von 10 Runden, bis eine Zelle wieder verschmutzt ist, sondern 20 Runden, da ja der Wiederverschmutzungswert ein Zufallswert zwischen 0 und 0,1 ist und somit im Schnitt bei 0,05 liegt. Somit kann der Own-Agent in den ersten 10 Runden einen „Vorsprung“ von 0,4 Zellen pro Agent herausarbeiten.

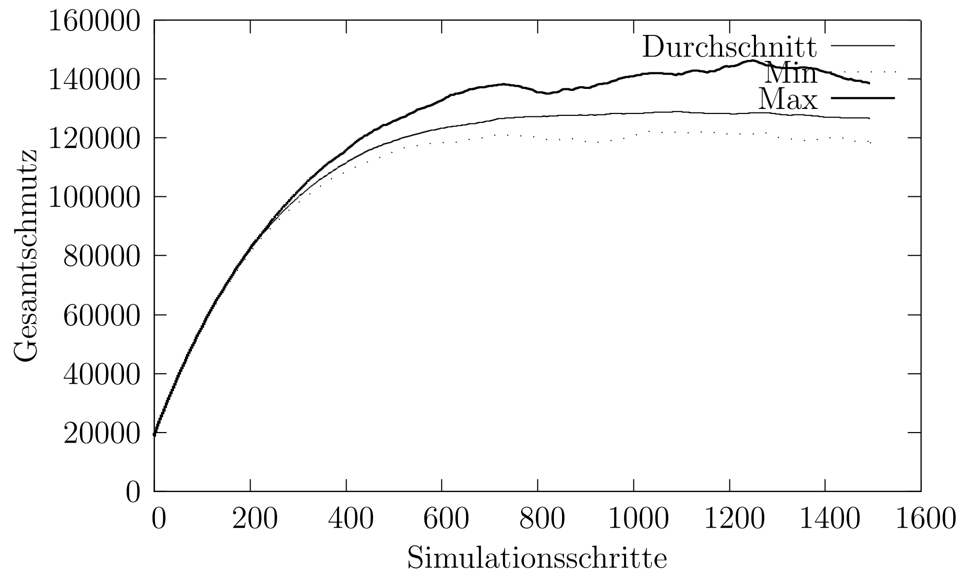


Abbildung 5.6: Experiment 24: Die Gesamtschmutzmenge, die auf dem Feld verteilt ist, stagniert nach ca. 800 Schritten bei einem Wert von ca. 125.000. Dies ist das Gleichgewicht, welches sich zwischen der konstanten Wiederverschmutzung und der ebenso konstanten Reinigungsleistung der Own-Agenten ergibt.

5.3 NEAT-Versuche

Wie bereits erwähnt, lässt sich das Verhalten des NEAT-Algorithmus über eine ganze Reihe von Parametern beeinflussen. Das Resultat der hierzu durchgeführten Versuche lässt sich wie folgt zusammenfassen:

Generationszeit Die Anzahl der Bewertungsschritte hat den größten Einfluss auf die Leistung der NEAT-Agenten. Ein Wert von 2 oder 3 liefert die besten Resultate, bei größeren Zeiträumen sinkt die durchschnittliche Leistung deutlich ab.

Mutationsraten Die Mutationsraten haben einen unerwartet kleinen Einfluss auf die erreichte Leistung. Liegen sie deutlich zu hoch, so driftet das Verhalten der Agenten ins Chaotische ab, andernfalls verändert sich das Ergebnis nur minimal. Die Art der Mutation (add Link oder add Node) spielt dabei keine signifikante Rolle.

Verschmutzungsrate Der Grad der Verschmutzung hat keinen Einfluss auf das Verhalten der Agenten (weder auf die NEAT-, noch auf die Own-Agenten). Lediglich Ergebniswerte wurden verändert, je höher die Wiederverschmutzungsrate war, um so geringer war auch die Zahl der sauber gehaltenen Zellen.

5.3.1 Vergleichsversuche

Nachdem für die NEAT-Parameter relativ gute Werte gefunden wurden, sollte die Leistung der NEAT-Agenten in Relation zu den anderen Strategien, namentlich dem SimpleVector-Agent (Zufall) und dem Own-Agent (Optimum) gesetzt werden. Unter besten Bedingungen (vgl. Tabellen A.5 und A.6) konnten 50 NEAT-Agenten ca. 130 Zellen sauber halten (vgl. Abb. A.12). Ein Vergleichsversuch (s. Abschnitt A.6) ergab jedoch, dass der SimpleVector-Agent unter gleichen Bedingungen 150 Zellen sauber halten kann, der Own-Agent gar 500. Es war nicht zu erwarten, dass die NEAT-Agenten die gleiche Leistung wie die optimalen Own-Agenten erbringen würden, es ist allerdings für einen Lernalgorithmus nicht hinnehmbar, dass er schlechter als eine zufallsbasierte Strategie ist.

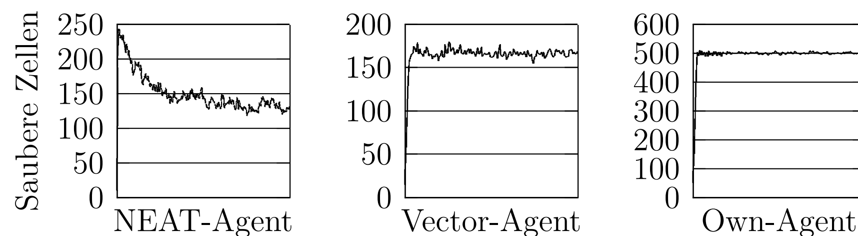


Abbildung 5.7: Vergleich verschiedener Agenten: Zu sehen ist, wie sich die Anzahl sauber gehaltener Zellen bei Einsatz unterschiedlicher Agenten entwickelt. Die Daten stammen aus den Versuchen r02-Neat-2-5000, r04-Vector-050-750 und r04-Own-050-750.

Da zunächst vermutet wurde, dass dies an der Bewertungsfunktion liegen könnte, wurden verschiedene Tests (Abschnitt A.7) mit verschiedenen Variationen derselben durchgeführt. Dabei stellte sich heraus, dass ihr Einfluss auf den Lernprozess minimal ist. Schließlich wurde nur noch mit der folgenden, stark vereinfachten Funktion gearbeitet: Für das Reinigen eines Feldes erhält der Agent 10 Punkte, für jede erfolgreiche Bewegung 0.1 Punkte und für alle weiteren Aktionen gar keine Punkte mehr. Die Versuche zeigten, dass die erreichten Ergebnisse dadurch weder besser noch schlechter wurden.

In der Versuchsreihe 26 (s. Abschnitt A.9) wurde mit von Hand kodierten neuronalen Netzen, die dem Algorithmus initial, anstelle der zufälligen Netze, übergeben wurden experimentiert, Abbildung 5.8 zeigt einen beispielhaften Verlauf der dabei entstandenen Fitnesskurve. Es ist erkennbar, wie der NEAT-Algorithmus ab Schritt 50 die Leitung verschlechtert.

Es kam die Vermutung auf, dass die Kodierung der Abfrage bei den NEAT-Agenten ungünstig gewählt ist: Bei jedem Schritt steht der Agent vor der Entscheidung, entweder sein aktuelles Feld zu putzen oder aber sich in eine von vier Richtungen zu bewegen. Dabei wurde im ursprünglichen Design nicht beachtet, dass die jeweilige Entscheidung für oder gegen eine bestimmte Richtung immer jeweils gleichartig ist und deshalb auch auf die gleiche Weise getroffen werden sollte. Da der

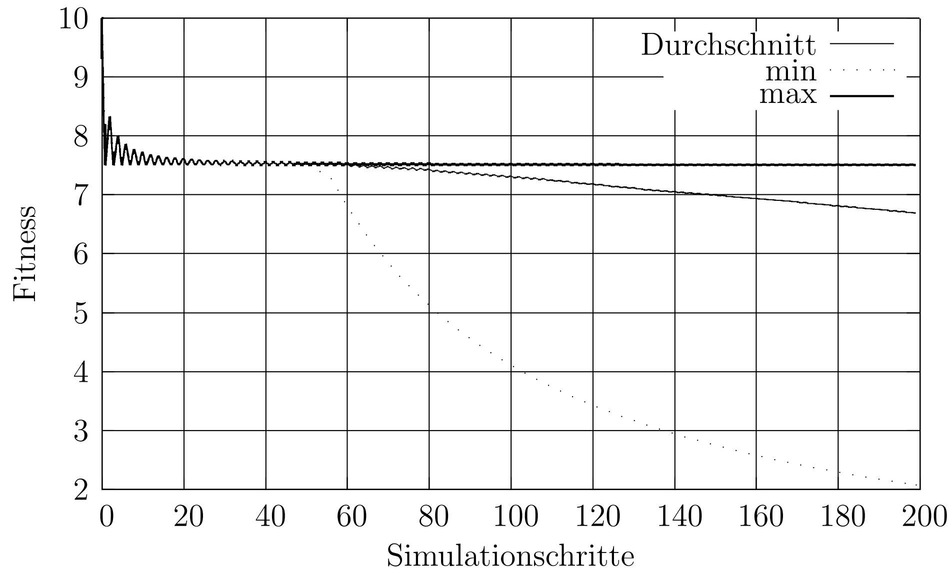


Abbildung 5.8: Versuch 26-PrefixedNeat-01-c: Ein von Hand kodiertes KNN wurde 50 Schritte lang unverändert genutzt und danach vom NEAT-Algorithmus verändert. Es ist deutlich erkennbar, dass mit Einsetzen der Evolution der Fitnesswert abnimmt.

NEAT-Agent jedoch z.B. sein nördliches Feld immer auf das gleiche Neuron abbildet und das südliche Feld auf ein *anderes* Neuron abgebildet wird (vgl. Abb. A.18), können sich sehr leicht Asymmetrien im Verhalten des Agenten ergeben. Wird z.B. durch eine Mutation eine Verbindung zum Ausgabeneuron NORD hinzugefügt, so wird der Agent solange eine Tendenz nach Norden aufweisen, bis die Verbindung durch eine gleichwertige auf das Ausgabeneuron SÜD ausgeglichen wird.

5.3.2 SmartNEAT Versuche

Zur Vermeidung dieser Ungleichheit wurde der SmartNEAT-Agent entworfen (vgl. Abschnitt 4.6.6). Das KNN dieses Agenten wird darauf trainiert, immer eine gleichartige Entscheidung zu treffen: Reinigen, nach vorn oder zu einer der beiden Seiten bewegen. Da auf diese Weise keine Präferenz zu einer bestimmten Seite erzeugt werden kann, ist zu erwarten, dass der SmartNEAT-Agent besser als der einfache NEAT-Agent abschneidet. Als weitere Vereinfachung kommt hinzu, dass das zu generierende KNN kleiner ist als beim NEAT-Agent, da der SmartNEAT-Agent nur je 2×3 Zellen als Eingabewerte berücksichtigen muss. Der später entwickelte SmartSimpleNEAT-Agent reduzierte diese Zahl durch alleiniges Berücksichtigen des Dirt-Layer auf 6 Eingabewerte.

Die SmartNEAT- und die SmartSimpleNEAT-Agenten wurden in der Versuchsreihe 28 untersucht (vgl. A.10). Dabei stellte sich heraus, dass ihre Leistung mit 200 sauber gehaltenen Zellen (vgl. Abb. A.20) sowohl über der der NEAT-Agenten (130 Zellen) als auch über der, der SimpleVector-Agenten (150 Zellen) lag.

Bei den nun folgenden Versuchsreihen r05 und r06 (Abschnitt A.11) zeigten sich zwei Dinge. Zum einen wurde festgestellt, dass die Genomgröße mit steigender Generationenzahl immer weiter zunimmt, obwohl die tatsächliche Leistung der Agenten bereits nach ca. 10 Generationen den Wertebereich erreicht, in dem sie für den Rest des Versuches bleibt (bis zu 8333 weitere Generationen). Dies widerspricht sich mit den Aussagen der NEAT-Autoren, der Algorithmus entwickle ein für das zu lösende Problem minimales KNN. Dieser Widerspruch deutet darauf hin, dass der NEAT-Algorithmus für dieses Szenario ungeeignet ist.

Zum anderen sind die Smart*Simple*NEAT-Agenten den SmartNEAT-Agenten überlegen, besonders bei längeren Läufen. So erreicht der SmartNEAT-Agent zwar innerhalb der ersten 1000 Simulationsschritte einen Wert von ca. 250 sauberen Zellen, dieser sinkt allerdings im weiteren Verlauf immer weiter ab und pendelte sich nach ca. 3000 Schritten um 200 ein. Diese Tendenz ist auch beim SmartSimpleNEAT-Agenten zu beobachten, wenn auch mit etwas besseren Werten. In den ersten 1000 Simulationsschritten pendelt der Wert hier noch um 270, sinkt aber im weiteren Verlauf ebenfalls ab und konvergiert schließlich gegen ca. 240 sauber gehaltene Zellen (vgl. Abb. A.21).

5.3.3 Abschließender Vergleich

Die Strategien der Own- und SimpleVector-Agenten sind fest und ändern sich im Verlauf der Simulation nicht. Es erscheint deshalb angebracht, einen Vergleich zwischen diesen beiden Strategien und einer Strategie zu ziehen, die durch den NEAT-Algorithmus generiert wurde.

Die Versuchsreihe 29 (vgl. Abschnitt A.12) hatte dies zur Aufgabe. Dazu bekommen die SmartSimpleNEAT-Agenten zunächst eine gewisse Zahl von Generationen Zeit, um eine Strategie zu entwickeln. Dann wird der Evolutionsmechanismus außer Kraft gesetzt, so dass die Agenten fortan mit dem bis zu diesem Zeitpunkt entwickelten KNN weiterarbeiten müssen.

Es zeigte sich, dass die Agenten offenbar gar keine Strategie erlernen. Unabhängig vom vorangegangenen Training stürzt die Zahl der sauber gehaltenen Zellen mit dem Beginn der zweiten Versuchsphase auf nur ca. 25 ab. Dabei spielte weder die Länge der Generationszeit in der vorangegangenen Trainingsphase noch die Anzahl der zuvor simulierten Generationen eine signifikante Rolle. Abbildung 5.9 veranschaulicht diesen Effekt deutlich.

5.4 Fazit

Der gewählte NEAT-Algorithmus liefert während des Evolutionsprozesses mittelmäßige Ergebnisse, die zwar signifikant über einer zufallsbasierten Strategie liegen, jedoch nur halb so gut sind, wie die der als optimal zu bezeichnenden festen, nicht lernenden Strategie.

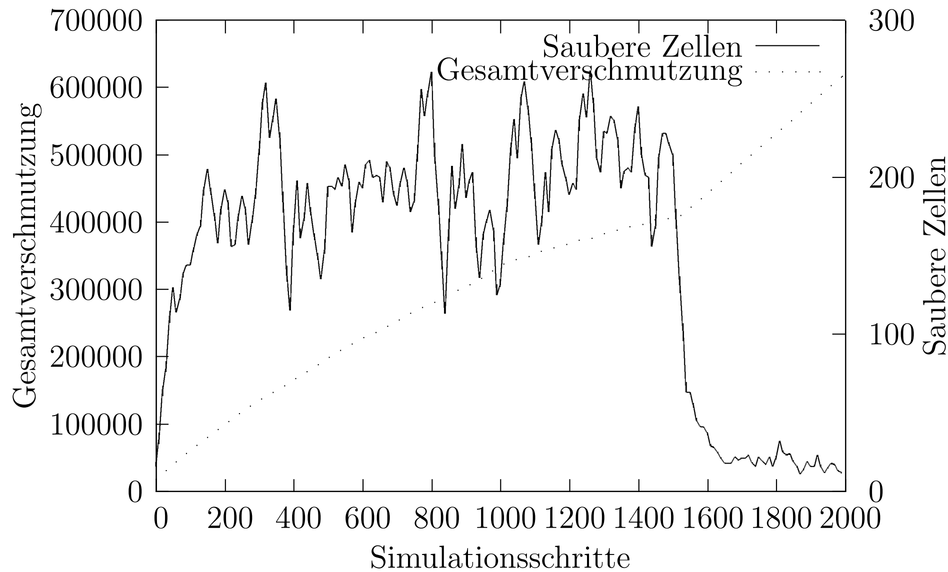


Abbildung 5.9: Versuch 29-SmartSimpleNEAT-02-dirtplot: Diese Grafik steht stellvertretend für die Versuchsreihe 29. Nach einer Trainingsphase von 1500 Schritten bzw. 100 Generationen werden die entwickelten KNN nur noch angewandt. Es wird erkennbar, dass die entwickelte Strategie nicht sehr leistungsfähig ist. Der tatsächliche Verlauf der Kurve ändert sich zwar mit verschiedenen Versuchsparametern, der charakteristische Leistungseinbruch nach Ende des Evolutionsprozesses ist jedoch in allen erkennbar. (Dieser Grafik liegt ein ausgewähltes Experiment mit nur einem Simulationslauf zugrunde, da es den darzustellenden Effekt besonders gut veranschaulicht.)

Sobald jedoch der Evolutionsprozess ausgesetzt wird, zeigt sich, dass keine Strategie erlernt wurde, die Ergebnisse liegen weit unter einer zufallsbasierten Strategie. Zudem sind die entwickelten Genome sehr groß, ein Mechanismus zur Begrenzung derselben ist im NEAT-Algorithmus nicht vorgesehen. Zu Vergleichszwecken wurde gezeigt, dass eine nahezu optimale Strategie in Form eines KNN ganz ohne versteckte Neuronen und mit nur fünf Verbindungen möglich ist (vgl. Abschnitt A.9). Diese Lösung wurde durch den NEAT-Algorithmus im Rahmen der in dieser Arbeit durchgeführten Versuche jedoch kein einziges Mal gefunden. Die Tabelle 5.2 fasst die Resultate der verschiedenen Versuche zusammen. Die aus den Versuchen gewonnene Erfahrung legt Nahe, dass dies daran liegen könnte, dass der NEAT-Algorithmus nicht geeignet ist, ein symmetrisches KNN zu entwickeln. Ein solches scheint jedoch für das im Rahmen dieser Arbeit verwendete Szenario am geeignetsten zu sein. Weitere mögliche Gründe werden in Abschnitt 6.2 diskutiert.

Daraus folgt, dass der gewählte Algorithmus für das beschriebene Problem nicht in der genannten Form angewendet werden kann.

Tabelle 5.2: Von je 50 Agenten sauber gehaltene Zellen

Own-Agent	500
eigenes KNN	490
SmartSimpleNeat-Agent	250
SmartNeat-Agent	200
Vector-Agent	155
NEAT-Agent	130
SmartSimpleNeat-Agent (nach Ende der Evolution)	25

Kapitel 6

Schlussfolgerungen & Ausblick

> *Die Entdeckung der Evolution schließt die Einsicht ein, dass unsere Gegenwart mit absoluter Sicherheit nicht das Ende (oder gar das Ziel) der Entwicklung sein kann.*
>
> Hoimar von Ditfurth, 1981

Die Frage, wie ein Schwarm von Agenten dezentral und kooperativ lernen könnte, eine bestimmte Aufgabe zu erfüllen und sich so organisieren, steuern und kontrollieren ließe, ist bisher nicht zufriedenstellend beantwortet. In dieser Arbeit wurde am Beispiel von Reinigungsrobotern das genannte Szenario in einer Simulation modelliert und der NEAT-Algorithmus angewendet.

Die bei den anschließenden Experimenten entstandenen Ergebnisse werden nun noch einmal kurz zusammengefasst und relativiert. Abschließend wird ein Ausblick auf mögliche zukünftige Vertiefungen und Erweiterungen der ursprünglichen Fragestellung gegeben.

6.1 Rückblick

Nach einer eingehenden Literaturrecherche wurde, ausgehend von der ursprünglichen Fragestellung, ein Szenario entwickelt, welches als Grundlage für eine Simulation dienen konnte. Anschließend wurde nach einer praktikablen Lernmethode gesucht, die ein dezentrales Lernen mit einem Austausch der beteiligten Akteure ermöglicht. Eine evolutionäre, genetische Herangehensweise wurde als geeignet identifiziert, da darin ja gerade der Austausch explizit vorgesehen ist. Ein Steuerungsansatz über *Künstliche Neuronale Netze* wurde gegenüber *Learning Classifier Systems* favorisiert, da Erstgenannte im evolutionären Rahmen bisher weniger erprobt, aber dennoch vielversprechend scheinen. Der NEAT-Algorithmus kombiniert die gewünschten Kriterien, er generiert genetisch-evolutionär KNN und ist noch relativ neu. Seine Erprobung im Rahmen dieser Arbeit sollte Aufschluss darüber geben,

in wieweit er sich für das beschriebene Szenario eignet.

Aus der Vielzahl verfügbarer Agentensimulationswerkzeugen galt es, eines zu wählen, das für das entwickelte Szenario geeignet war. Das ursprünglich in Betracht gezogene Agentensimulationsframework Repast Simphony bietet eine große Palette an Möglichkeiten und Funktionen, jedoch hatte es Unzulänglichkeiten, die es für eine Verwendung disqualifizierten.

Aus der Studie des Repast Frameworks konnte jedoch genug Wissen erlangt werden, um in kurzer Zeit ein eigenes einfaches Simulationswerkzeug zu entwickeln. Das Szenario wurde implementiert und mit der NEAT-Implementierung JNEAT verknüpft.

Im Anschluss wurden mit dem entwickelten Simulationswerkzeug eine Reihe von Versuchen durchgeführt. Das Ziel war es, die Agenten eine emergente Strategie erlernen zu lassen und so die Fähigkeiten des NEAT-Algorithmus zu evaluieren.

Im Laufe dieser Experimente zeigte sich jedoch, dass der Algorithmus in seiner Handhabung sehr schwierig ist. Die erreichten Ergebnisse, im Hinblick auf das ursprüngliche Ziel des kooperativen Lernens einer emergenten Strategie, sind leider enttäuschend.

6.2 Analyse

Das Maß für die erbrachte Leistung in dem gewählten Szenario wurde durch die Anzahl an sauber gehaltenen Zellen definiert, der maximal erreichbare Wert wurde errechnet und experimentell validiert. Es hat sich gezeigt, dass der NEAT-Algorithmus während des laufenden Evolutionsprozesses bestensfalls eine Leistung von weniger als der Hälfte dieses Maximalwertes erreichte. Dieser Wert ist besser als eine zufallsbasierte Strategie. Da das Ziel jedoch die Entwicklung einer Strategie war, die auch ohne weiteres Lernen anwendbar wäre, muss dieser Wert als belanglos betrachtet werden. Aussagekraft hat lediglich der Wert, der nach Abschluss des Evolutionsprozesses von der bis dahin erlernten Strategie erbracht wird. Dieser Wert lag jedoch nur bei einem Bruchteil des erreichbaren Wertes und außerdem deutlich unter einer rein zufallsbasierten Strategie. Hinzu kommt, dass mit Fortdauern des Evolutionsprozesses die Genomgröße stetig zunimmt.

Die ursprünglichen Erwartungen, die mit der Benutzung des NEAT-Algorithmus verknüpft waren, wurden demzufolge nicht erfüllt. Daraus lässt sich der Schluß ziehen, dass NEAT für das gewählte Szenario ungeeignet ist. Da NEAT jedoch, in Form des Computerspiels NERO, bereits erfolgreich eingesetzt wird, um einen Schwarm von Agenten zu steuern, kann nicht gesagt werden, dass er sich grundsätzlich nicht zur Steuerung von Agentenschwärmen eignet.

Die möglichen Gründe für den erfolglosen Versuch der Anwendung sind vielfältig. Eine mögliche Fehlerquelle stellt die Parametrisierung des Algorithmus dar. Zu den 34 Parametern des Algorithmus selbst kommen weitere 11 Simulationsparameter hinzu. Zwar ist der Standardimplementierung ein Parametersatz beigefügt (der auch getestet wurde), jedoch können diese Werte in Abhängigkeit vom Anwendungspro-

blem variieren. Trotz der großen Zahl an durchgeführten Versuchen kann also nicht sichergestellt werden, dass eine optimale Parameterkombination gefunden wurde.

Außerdem kann das beschriebene Szenario selbst auch eine mögliche Fehlerursache sein. Während die Agenten bei NERO sich kontinuierlich bewegen können, sind die Agenten im Rahmen dieser Arbeit auf ein Grid beschränkt. Dadurch ist sowohl ihre Bewegungsgeschwindigkeit vorgegeben, als auch ihre Bewegungsrichtungen diskret. Zwar ist auch das NEAT-Refferenzbeispiel, die XOR-Funktion, ein diskretes Problem, jedoch in einem sehr viel kleineren Suchraum. Möglicherweise ist NEAT für diskrete Entscheidungen in komplexen, realwertigen Suchräumen nur bedingt geeignet.

Die mit steigender Generationenzahl beständig wachsende Genomgröße ist ein weiteres Indiz für die schlechte Eignung. Nach Aussage der Autoren findet NEAT immer die kleinstmögliche Lösung für ein gegebenes Problem. Wird jedoch keine Lösung gefunden, so ist die logische Konsequenz, dass die in Betracht gezogenen Lösungen immer weiter wachsen. Dass eine kleine Lösung existiert, die auch sehr gut funktioniert, wurde in Versuchsreihe 26 (Abschnitt A.9) gezeigt.

6.3 Ausblick

Neben einigen alternativen Herangehensweisen an das beschriebene Szenario sind eine Reihe von Modifikationen und Erweiterungen desselben denkbar. Diese Arbeit schließt mit der Nennung einer Auswahl der interessantesten Möglichkeiten.

6.3.1 Modifikationen

Mit steigender Größe des in Betracht gezogenen Genoms steigt auch die Komplexität des kodierten KNN. Je komplexer ein solches jedoch ist, umso aufwendiger ist auch seine Simulation und somit die Berechnung seiner Ausgabeneuronen. Schon deshalb ist es erstrebenswert, ein für das jeweilige Problem minimales KNN zu erhalten. Es hat sich gezeigt, dass das Genomwachstum beim NEAT-Algorithmus keiner tatsächlichen Beschränkung unterliegt. Es ist lediglich wahrscheinlich, dass eine kleine Lösung aufgrund des kleineren Suchraumes vor einer Großen gefunden wird. Sollte es sich jedoch ergeben, dass eine große Lösung ebenso wie eine kleine gefunden wird und diese in ihrer Fitness übereinstimmen, dann gibt es keine weiteren Auswahlkriterien. An dieser Stelle wäre es denkbar, zusätzlich zur Fitness ein weiteres Maß in die Bewertung einer möglichen Lösung einfließen zu lassen. So könnte sich eine große Lösung negativ auf die Fitness auswirken. Dabei lässt sich die Größe entweder in einer großen Zahl von Neuronen, Verbindungen oder Beidem ausdrücken. Wichtig ist nur, dass dieses Maß proportional zur für die Simulation notwendigen Rechenzeit steht. So wäre sichergestellt, dass eine kleine Lösung einer Großen (bei gleicher Güte) vorgezogen wird.

Der NEAT-Algorithmus hat sich als wenig geeignet für das beschriebene Szenario gezeigt. Um dies zu validieren, wäre es angebracht, alternative Lernverfahren auf

das gleiche Szenario anzuwenden. Denkbar wären hier zum Beispiel *Learning Classifier Systeme* oder KNN mit klassichen nicht-evolutionären Lernverfahren. Allerdings wäre es im letztgenannten Fall schwierig, das Wunschkriterium des kollaborativen Lernens zu modellieren.

In [DBS07] wird der NEAT-Algorithmus mit einem neuartigen Verfahren namens ESN (Echo State Networks) verglichen. Die Autoren kommen dabei zu dem Schluss, dass in manchen Anwendungsfällen ESNs einer NEAT-basierten Lösung überlegen sind. Falls sich dieses Verfahren behaupten kann, wäre es interessant, es ebenfalls auf das behandelte Problem anzuwenden.

Schließlich wäre es denkbar, die Genotyp-Phänotyp-Abbildung zu variieren. Diese ist bei NEAT sehr intuitiv gewählt (vgl. Abschnitt 3.4). Es wurde jedoch bereits erwähnt, dass in der Biologie diese Abbildung nicht bijektiv ist. Bei einer geeigneteren Abbildung könnte man sich z.B. Symetrieeffekte zu Nutze machen, so wie dies auch in der Biologie geschieht.

Ebenso könnte die Crossover-Funktion dahingehend erweitert werden, dass das „stärkere“ Elternteil (ausgedrückt durch einen höheren Fitness-Wert) ein größeres Stimmgewicht bei der Verteilung der einzelnen Gene hat (beim NEAT-Algorithmus geschieht diese Aufteilung zufällig, vergl. Abschnitt 3.6.4). Dieser Mechanismus könnte analog zu dominanten/rezessiven Genen in der Vererbungslehre nach Gregor Mendel modelliert werden.

6.3.2 Erweiterungen

Hat man einmal ein Verfahren gefunden, welches für das gegebene Szenario geeignet ist, so lassen sich durch geeignete Erweiterungen desselben einige, evolutionär interessante, Experimente durchführen.

So könnte beispielsweise untersucht werden, ob sich auch dann stabile Strategien bilden können, wenn zwei Populationen *in Konkurrenz zueinander* eine Lösung zu einem gegebenen Problem finden müssen, oder ob diese sich mit ihren Verhaltensweisen gegenseitig zu immer neuen Innovationen antreiben, ohne jemals eine optimale Lösung zu finden. Im Fall der Konkurrenz müsste auch betrachtet werden, welchen Einfluss die Kommunikation (in diesem Fall in Form der Pheromonspuren) auf die Entwicklung hat. Die Pheromonspuren könnten dann z.B. als mehrdimensionales Array aufgefasst werden, so dass nachvollziehbar bleibt, *wer* sie hinterlegt hat, etwa ein Konkurrent oder ein Mitglied des eigenen Schwarms.

Eine weitere Variante des Konkurrenzansatzes wäre die Aufspaltung der Aufgabe. So könnte ein Team die Aufgabe bekommen, das Spielfeld zu verschmutzen, während das zweite Team diesen Schmutz wieder aufsammeln müsste. Eine denkbare Strategie für die putzenden Roboter wäre in einem solchen Fall, die „Verschmutzer“ zu verfolgen und den verteilten Schmutz direkt wieder zu entfernen.

Ebenso interessant wäre es, zu überprüfen, welche Vorteile eine Spezialisierung einer Spezies in bestimmte Kasten (Arbeiter, Kundschafter usw.) haben kann, so wie es z.B. in bestimmten Ameisenstaaten der Fall ist.

Das gewählte Szenario zielt darauf ab, dass jeder Agent für sich allein handelt, und sich daraus ein sinnvolles Ganzes ergibt. Es wäre interessant zu sehen, welche Strategien sich entwickelten, wenn sich aus der *Kooperation* mehrerer Agenten ein Vorteil ergäbe. So könnte man es beispielsweise einrichten, dass sich zwei Agenten zusammen schneller bewegen können als jeweils alleine, ähnlich wie das beim Brettspiel „Halma“ der Fall ist.

Eine weitere Variante wäre es, nur lokale Paarungen zuzulassen. Dies wäre näher am biologischen Vorbild, wo sich auch nur diejenigen Individuen vermehren, die sich auch tatsächlich treffen. Zwar wäre dafür ein stark vergrößertes Spielfeld notwendig, auf dem sich auch weit mehr als die in dieser Arbeit verwendeten 50 Agenten bewegen müssten. Es bestünde dann aber die Möglichkeit der Bildung lokaler Strategien und „echter“ Spezies.

6.3.3 Alternativen

Es hat sich gezeigt, dass die zweigeschlechtliche Fortpflanzung einige evolutionstechnisch bedeutsame Mechanismen mit sich bringt, etwa das Crossover. Andererseits vermehren sich z.B. Viren und Bakterien seit langem erfolgreich eingeschlechtlich. Dabei gelingt es ihnen trotzdem immer wieder, bestimmte Eigenschaften oder Fähigkeiten (etwa bestimmte Resistenzen), auch über Artengrenzen hinaus, auszutauschen. Eine Nachbildung dieses Prozesses könnte eine alternative Herangehensweise an lokales, kollaboratives Lernen sein.

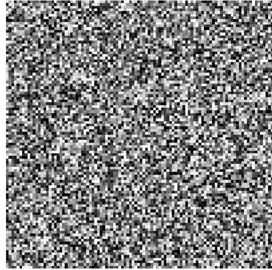
Anhang A

Versuchslogbuch

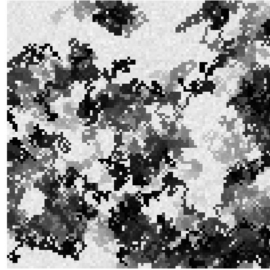
Im Folgenden wird ein Überblick über einige der durchgeführten Versuche gegeben. Die Auflistung ist nicht vollständig, vielmehr wurden einige Versuche ausgewählt, aus denen besondere Erkenntnisse gewonnen wurden, oder aus denen das Vorgehen während der Versuchsdurchführung deutlich wird.

Der Leser sollte sich insbesondere nicht durch die Namensgebung der Versuche verwirren lassen. Die Namen der Versuche setzen sich aus einer Vielzahl von technisch bedingten Komponenten zusammen. Der Einfachheit halber wurden sie hier beibehalten.

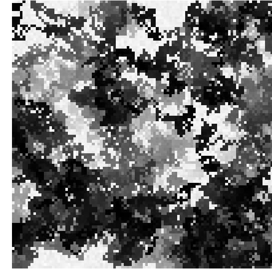
A.1 Erste Gehversuche



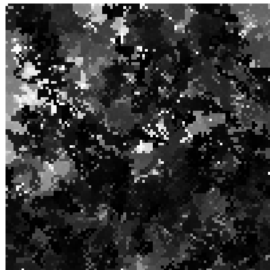
(a) Zu Beginn,



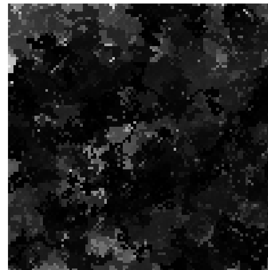
(b) nach 500,



(c) 1000,



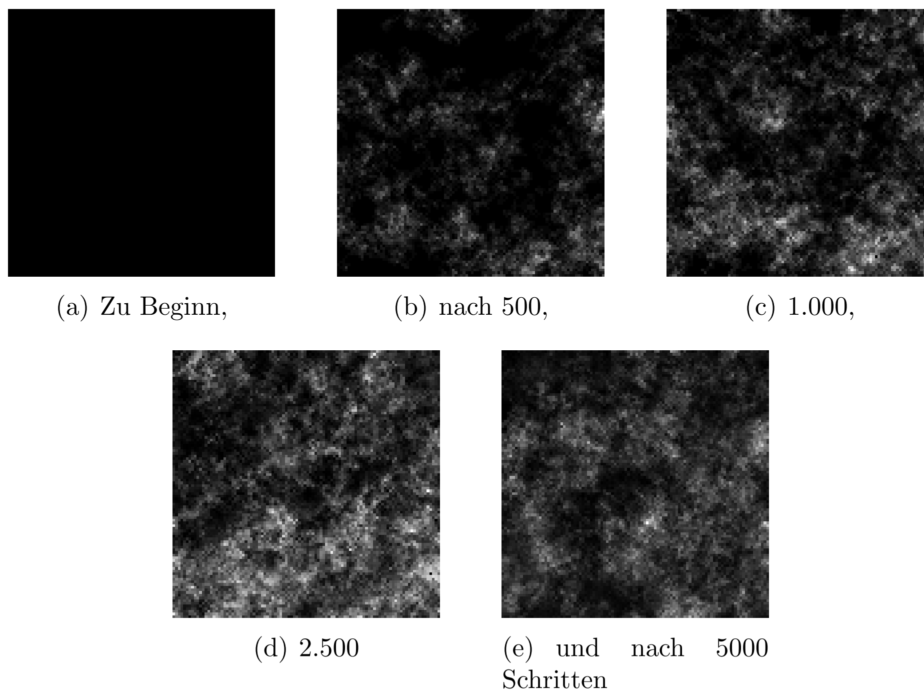
(d) 2500



(e) und nach 5000 Schritten

Abbildung A.1: Testexperiment 1: Die sich ergebende Verschmutzungskarte zeigt, dass auch planloses Herumlaufen über lange Zeit zu einer relativ sauberen Karte führt.[a]

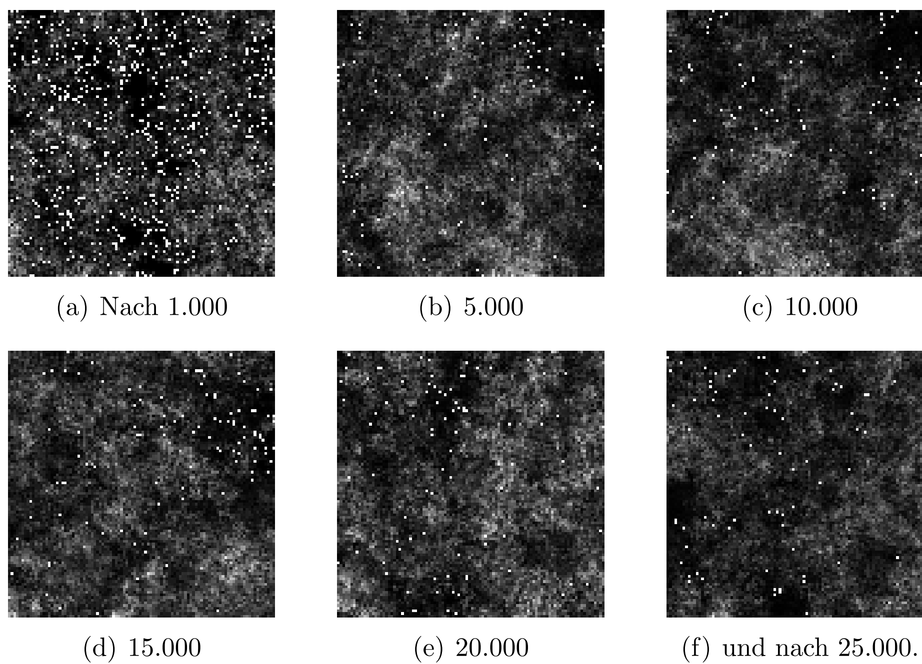
[a]Schwarz ist hierbei eine saubere Zelle. Je heller die Zelle, desto stärker ist sie verschmutzt. Die Skalierung der Grauwerte erfolgt jeweils relativ zu den aktuell auf der Karte verwendeten Werten und ist nicht über die Bildfolge einheitlich.



(a) Zu Beginn, (b) nach 500, (c) 1.000, (d) 2.500 (e) und nach 5000 Schritten

Abbildung A.2: Testexperiment 1: Die zugehörige Pheromonkarte zeigt jedoch deutlich, dass die Flächenabdeckung sehr unregelmäßig ist.[a]

[a]Ähnlich wie bei den Verschmutzungsbildern bedeutet hier eine schwarze Zelle eine unberührte Zelle, weiß ist die stärkste Pheromonmarkierung der Karte. Die Skalierung der Grauwerte erfolgt jeweils relativ zu den aktuell auf der Karte verwendeten Werten und ist nicht über die Bildfolge einheitlich.



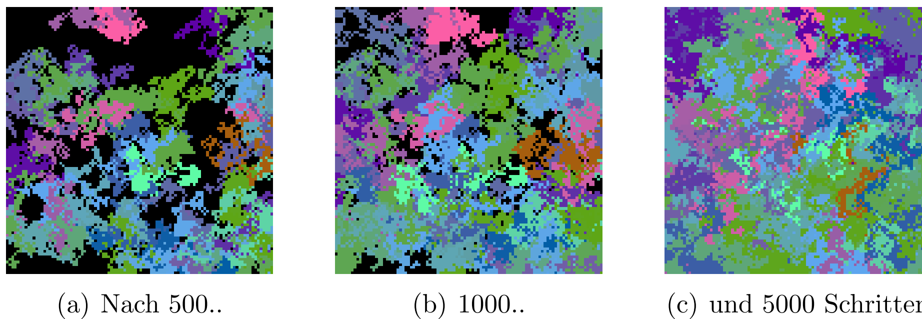
(a) Nach 1.000 (b) 5.000 (c) 10.000

(d) 15.000 (e) 20.000 (f) und nach 25.000.

Abbildung A.3: Testexperiment 2: Insgesamt 100 SimpleVectorAgents laufen 25.000 Schritte lang über eine Karte. Zu sehen sind die Pheromonkarten. Je heller ein Bereich gefärbt ist, desto höher ist die Pheromonkonzentration an dieser Stelle, die weißen Punkte repräsentieren die Agenten.

A.2 Erste Versuchsreihen

Zunächst musste der Simulator auf Benutzbarkeit geprüft werden. Dabei ging es vornehmlich um die Ausgabe der Bilder und der Logfiles für die Plots. Abbildung A.4 zeigt ein Beispiel für die Ausgabe der Pfadkarten. Darauf wird erkennbar, welche Teile der Karte von einem Agenten besucht wurden (farbig) und welche nicht (schwarz).



(a) Nach 500.. (b) 1000.. (c) und 5000 Schritten

Abbildung A.4: Testexperiment 1: 50 Simple-Agents laufen 5000 Schritte lang über eine Karte

Im Großen und ganzen wurde die Umgebung mit zwei Testexperimenten getestet. In Testexperiment 1 traten 50 *SimpleAgent*s auf einem 100×100 Zellen großen Feld an. Die maximale Anfangsverschmutzung lag bei 2, der maximale Verschmutzungsgrad pro Runde bei 0.1. Als Grenze, ab wann eine Fläche als verschmutzt gilt wurde 1 gewählt. Somit ist zu Beginn des Experiments ungefähr jede zweite Zelle (ca. 5000) verschmutzt. Der Verlauf der Gesamtverschmutzung und die Anzahl der verschmutzen Zellen über den Verlauf der Simulation sind auf Abbildung A.5 zu sehen.

Ein Simple-Agent hat eine sehr einfache „Strategie“. Er reinigt bei jedem Zug seine aktuelle Zelle und läuft danach in eine zufällige Richtung[1]. Jedes von ihm betretene Feld markiert er mit einem Pheromonmarker der Stärke 1.

Der Versuchsaufbau von Testexperiment 2 gleicht dem Versuchsaufbau von Testexperiment 1 mit der Ausnahme, dass hier *SimpleVector-Agenten* zum Einsatz kommen. Die ihnen zu Verfügung stehenden Aktionsmöglichkeiten gleichen denen, die schließlich in den intelligenten Agenten zum Einsatz kommen sollen. Sie können sich also *entweder* in eine von vier Richtungen bewegen *oder* das Feld reinigen, auf dem sie zur Zeit stehen. Zusätzlich können sie in jeder Runde einen Pheromonmarker auf ihr aktuelles Feld setzten. Dessen Stärke kann zwischen -1 und 1 liegen und wird zufällig bestimmt. Näheres zu den verwendeten Agenten findet sich in Abschnitt 4.6.

Die Ergebnisse aus Testexperiment 1 sind nicht weiter von Interesse. Wie auf Abbildung A.6 zu sehen ist, pendelt sich der Fitnesswert der Agenten bereits nach

[1] Der Simple-Agent ist (im Gegensatz zu allen anderen Agenten) in der Lage, eine von *acht* Richtungen zu wählen. Er hat dadurch einen Geschwindigkeitsvorteil von $\sqrt{2}$.

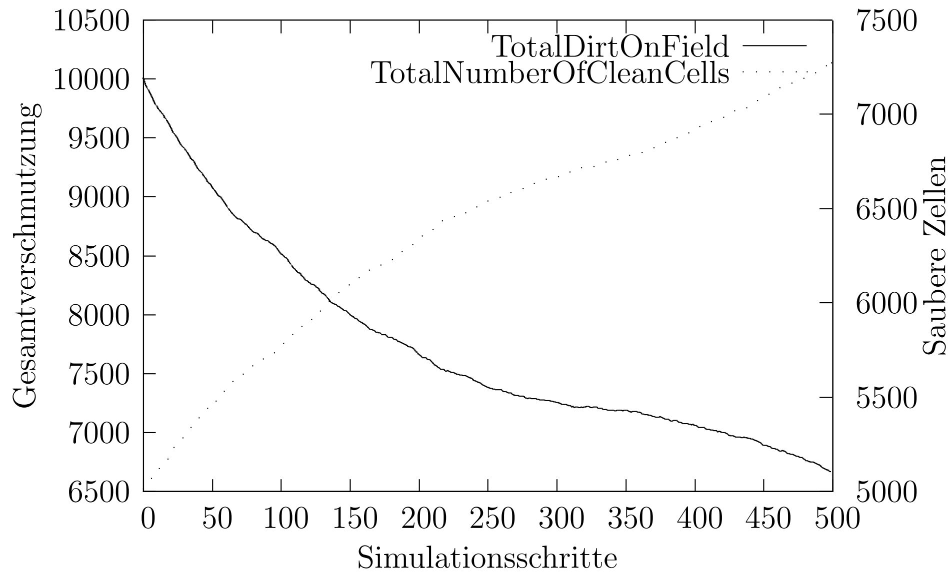


Abbildung A.5: Testexperiment 1: Bei einer maximalen Anfangsverschmutzung von 2, einem maximalen Verschmutzungsgrad von 0.002 und einer Sauberkeitsgrenze ergeben sich diese Verteilungen.

ca. 75 Schritten auf einen Wert ein (in diesem Fall knapp über zehn) und bleibt dann relativ konstant. Dies begründet sich darin, dass die Strategie des Simple-Agent sich nicht ändert. Die Wahrscheinlichkeit, durch eine zufällig gewählte Aktion einen Fitnesspunkt zu gewinnen ist immer gleich und mittelt sich somit über die Zeit.

Interessanter sind hingegen schon die Ergebnisse des Testexperiments 2, sichtbar auf Abbildung A.7. Auch die Strategie der hier verwendeten SimpleVector-Agenten ist zufallsbedingt, so dass sich der Fitnesswert ebenfalls nach kurzer Zeit auf einen festen Wert einpendelt. Auch die Tatsache, dass dieser Wert niedriger ist als beim Simple-Agent kann nicht verwundern, kann doch der Simple-Agent gleichzeitig putzen *und* sich bewegen (und somit sehr viel mehr Punkte pro Zug gewinnen).

Erstaunlich ist hingegen die Tatsache, dass die Anzahl der sauberen Zellen um 4.500 pendelt (Abb. A.8). Auch die Schmutzmenge, die über das gesamte Feld verteilt ist stagniert bei ca. 18.000. Da es sich hier jedoch um grundlegende Testversuche handelt, werden den konkreten Werten an dieser Stelle keine Bedeutung beigemessen.

Wird das gleiche Experiment mit 12, 25, 50 und 100 Agenten gemacht, so sieht man auf Abbildung A.9, dass sich beim Schritt von 12 auf 25 Agenten die Zahl der sauberen Zellen noch nahezu verdoppelt. Der Zuwachs beim Schritt von 25 auf 50 Agenten fällt mit ca. 4500 sauberen Zellen hingegen geringer aus als eigentlich zu erwarten wäre, noch deutlicher ist dieser Effekt bei 100 Agenten zu beobachten. Daraus kann man schließen, dass bereits 50 gleichzeitig eingesetzte Agenten gegenseitig behindern, sofern sie sich nach einem rein zufälligen Prinzip bewegen.

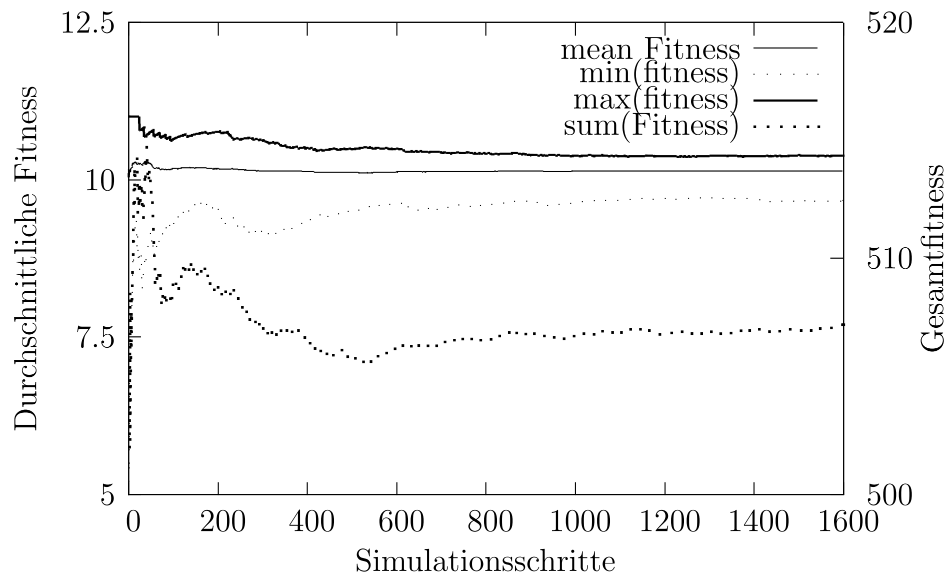


Abbildung A.6: Testexperiment 1: Die Fitness pendelt sich recht schnell bei einem Wert knapp über zehn ein.

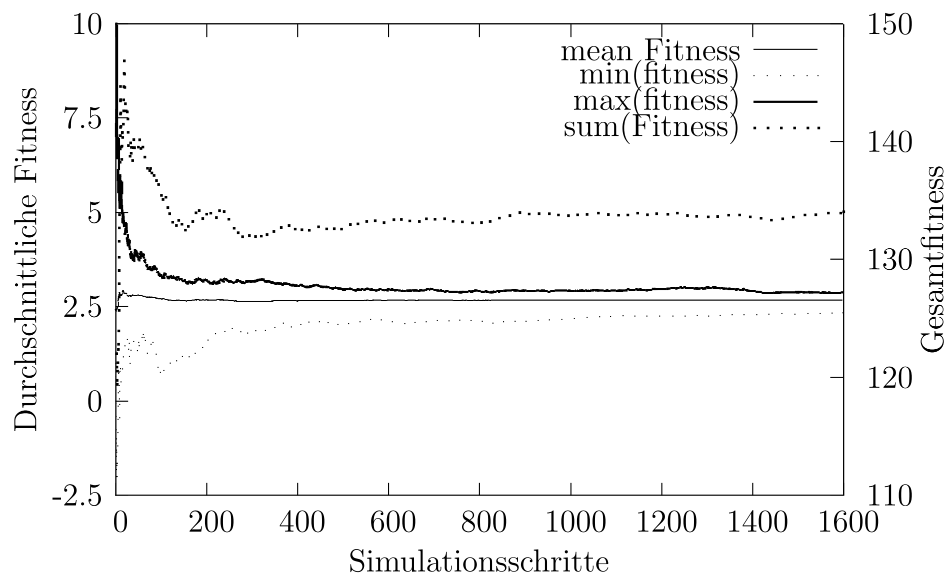


Abbildung A.7: Testexperiment 2: Wie zu erwarten war, sinkt der Fitnesswert der *SimpleVector-Agenten* im Vergleich zu den *SimpleAgenten* stark ab. Dies liegt wohl vor allem daran, dass sie nicht mehr in der Lage sind, gleichzeitig zu putzen und sich zu bewegen.

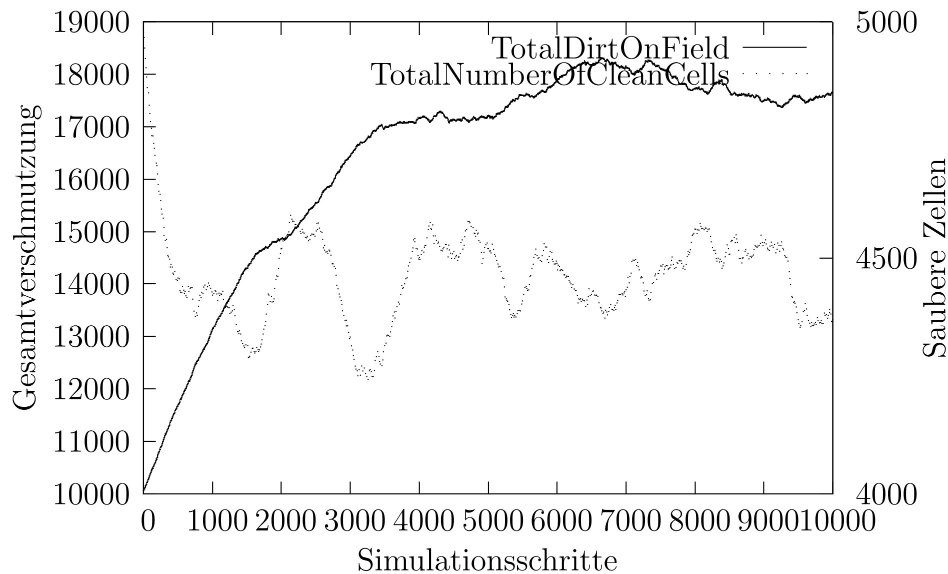


Abbildung A.8: Testexperiment 2: Trotz gleicher Randbedingungen liefert der SimpleVector-Agent ein ganz anderes Resultat. Dies liegt vor allem daran, dass er (im Gegensatz zum Simple-Agent) nicht mehr gleichzeitig reinigen und sich bewegen kann. Dadurch steigt die Gesamtverschmutzung kontinuierlich an. Erstaunlicherweise pendelt sich die Anzahl der sauberen Zellen bei ca. 4500 ein.

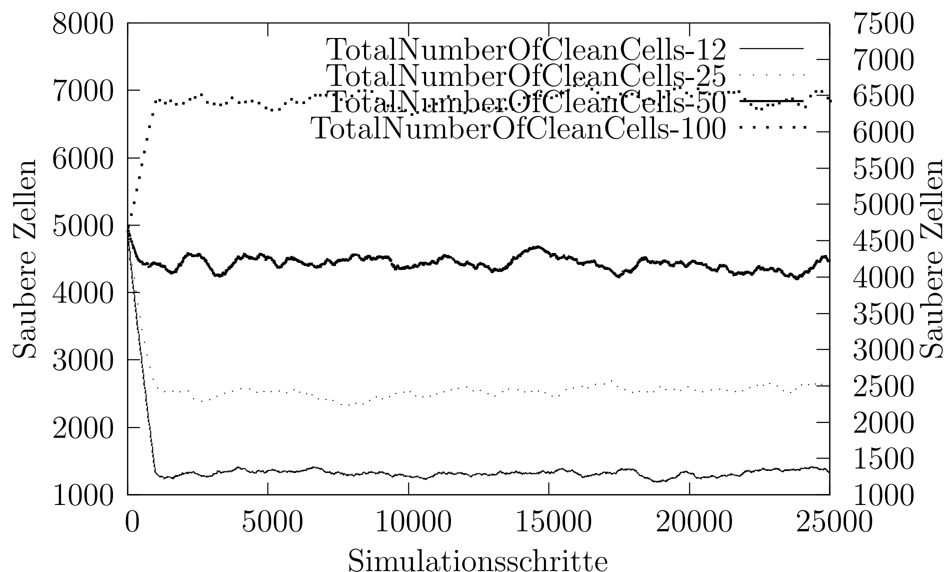


Abbildung A.9: Testexperiment 2: Hier ist zu sehen, wie sich die Anzahl der sauberen Felder erhöht, wenn mehr Agenten zum Einsatz kommen.

A.3 Versuchsreihe Statistiktests

Um die Tauglichkeit der Simulationsumgebung auf eine größere Anzahl von Versuchen mit jeweils mehreren Simulationsläufen zu testen, werden hier eine Reihe von Testversuchen durchgeführt.

A.3.1 Versuch Nr. 14

Im Zuge einer späteren Automatisierung wird getestet, ob die Daten der verschiedenen Läufe in die richtigen Verzeichnisse geschrieben werden. Außerdem soll geprüft werden, ob es zwischen den einzelnen Läufen große Unterschiede gibt.

Tabelle A.1: Simulationsparameter für Versuch 14

experimentName	14-SimpleVectorAgent-50-15000-noInputs
numberOfSteps	1500
numberOfRuns	3
randomSeed	1
numberOfAgents	50
neatRoundsPerGeneration	15
worldXSize	100
worldYSize	100
worldDirtMax	2.0
worldDirtRespread	0.1
worldCleanCellThreshold	1.0
worldPheromoneFalloff	0.0
statsReportImagesEveryNRounds	150
statsReportDataEveryNRounds	25

Ergebnisse

Die Daten landen in den richtigen Verzeichnissen, die Weltparameter werden nach jedem Lauf korrekt resetet.

Es fällt auf, dass die Agenten in allen Läufen einen Hang zur rechten Seite entwickeln. Dies ist sowohl auf den Verschmutzungs- als auch auf den Pfadbildern zu erkennen.

Schlussfolgerungen, Ideen

In einem weiteren Versuch sollte geklärt werden, ob der Hang zur rechten Seite sich auch über eine größere Anzahl von Runden hält.

Tabelle A.2: NEAT-Parameter für Versuchsreihe *Statistiktests*

p_trait_param_mut_prob	10.5
p_trait_mutation_power	1.0
p_linktrait_mut_sig	1.0
p_nodetrait_mut_sig	0.5
p_weight_mut_power	1.0
p_recur_prob	0.05
p_disjoint_coeff	1.0
p_excess_coeff	1.0
p_mutdiff_coeff	7.0
p_compat_threshold	9.0
p_age_significance	1.0
p_survival_thresh	0.4
p_mutate_only_prob	0.25
p_mutate_random_trait_prob	0.1
p_mutate_link_trait_prob	0.1
p_mutate_node_trait_prob	0.1
p_mutate_link_weights_prob	0.8
p_mutate_toggle_enable_prob	0.01
p_mutate_gene_reenable_prob	0.0010
p_mutate_add_node_prob	0.01
p_mutate_add_link_prob	0.3
p_interspecies_mate_rate	0.01
p_mate_multipoint_prob	0.6
p_mate_multipoint_avg_prob	0.4
p_mate_singlepoint_prob	0.0010
p_mate_only_prob	0.2
p_recur_only_prob	0.2
p_pop_size	50
p_dropoff_age	15
p_newlink_tries	20
p_print_every	60
p_babies_stolen	0
p_num_runs	1
p_num_trait_params	8

A.3.2 Versuch Nr. 16

Unter der Versuchsnummer 16 wurde eine ganze Reihe von Versuchen durchgeführt. Dabei wurde die Anzahl der Gesamtläufe variiert um herauszufinden, ab wann die gemittelte Fitness über alle Läufe einen relativ glatten Verlauf nimmt.

Tabelle A.3: Simulationsparameter für Versuch 16

experimentName	16-StatTest01
numberOfSteps	500
numberOfRuns	1, 5, 10, 15, 20
randomSeed	1
numberOfAgents	50
neatRoundsPerGeneration	15
worldXSize	100
worldYSize	100
worldDirtMax	2.0
worldDirtRespread	0.1
worldCleanCellThreshold	1.0
worldPheromoneFalloff	0.0
statsReportImagesEveryNRounds	15
statsReportDataEveryNRounds	3

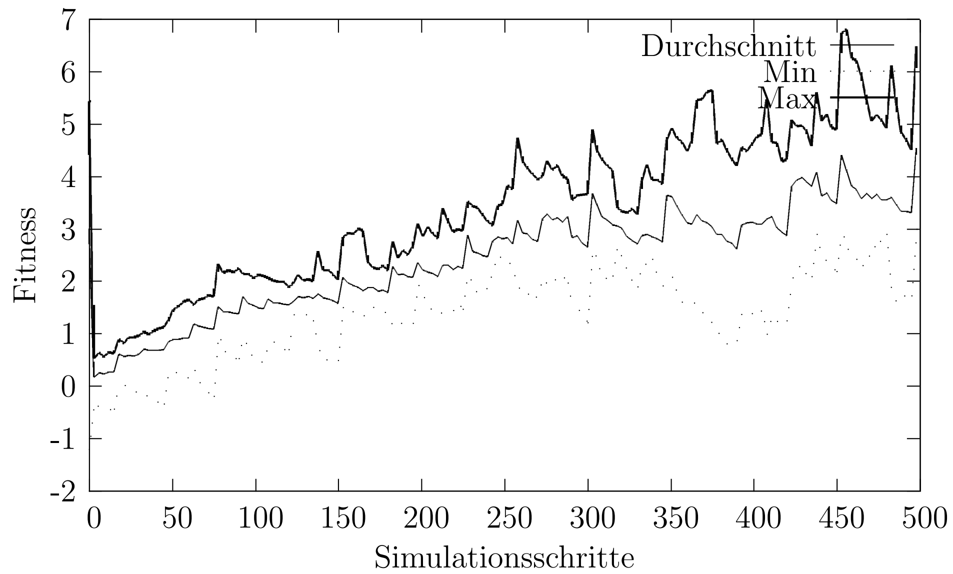
Ergebnis (20) Bei der Durchsicht der Ergebnisse fällt auf, dass die NEAT-Agenten in den ersten vier Läufen eine Tendenz dazu entwickelten, die rechte Hälfte des Feldes deutlich besser zu reinigen als die Linke. Dieses Phänomen kehrte sich in den darauf folgenden 15 Läufen um, so dass nun die linke Hälfte des Feldes besser gereinigt wurde, als die Rechte.

Ein Vergleich mit den Pfadkarten bestätigt die Vermutung, dass die von den Agenten in diesem Versuch entwickelte Strategie hauptsächlich im zur-Seite-bewegen besteht. Bewegungen und der Vertikalen kommen sehr selten vor.

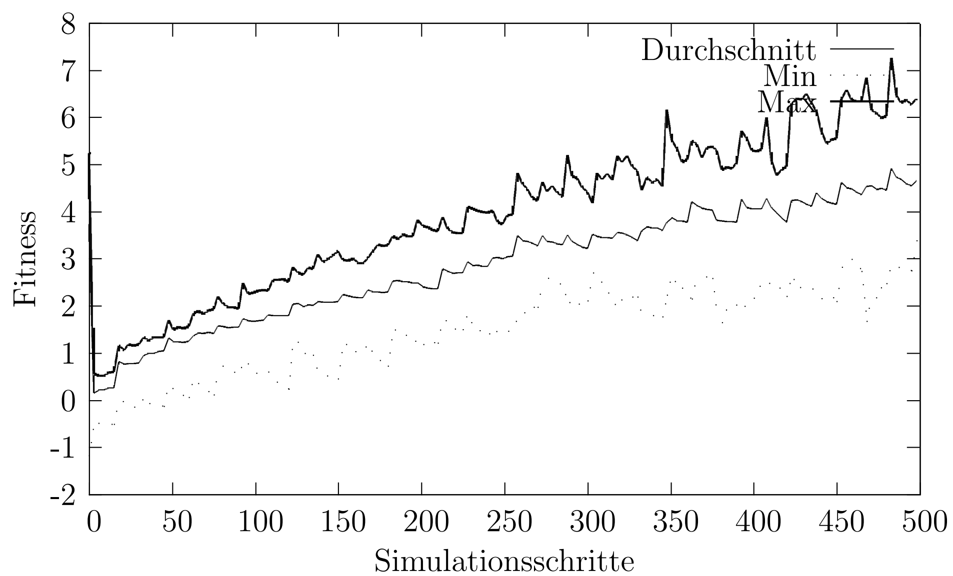
Nebenbei bemerkt erzeugt dieses Experiment mit seinen 20 Durchläufen, mit den genannten Parametern, über 2.500 Dateien mit insgesamt ca. 74 MB an Daten.

A.3.3 Versuch Nr. 16-02

Der NEAT-Algorithmus braucht eine gewisse Mindestanzahl von Organismen oder mit anderen Worten eine Mindestgröße der Population um einen sinnvollen evolutionären Auswahlprozess simulieren zu könne. Theoretisch ist hier das zu erwartende Ergebnis besser, je größer die Population ist. In den praktischen Versuchen unterliegt die Populationsgröße jedoch zwei Beschränkungen. Zum einen ist das Spielfeld nur 100×100 Felder groß. Wie schon die Experimente mit den Zufallsagenten (A.8)



(a) 5 Läufe



(b) 20 Läufe

Abbildung A.10: Testexperiment 16/20: Die Kurve der durchschnittlichen Fitnesswerte über 20 Versuche ist bereits deutlich glatter als die Kurve, die bei gleicher Versuchskonfiguration und nur 5 Durchläufen entsteht.

gezeigt haben, stagniert der Mehrwert von mehr Agenten ab einer gewissen Anzahl von Agenten auf dem Feld, da sie sich gegenseitig behindern.

Die zweite Beschränkung besteht in der zur Verfügung stehenden Rechenzeit. Diese Beschränkung fällt bei den verwendeten Versuchskonfigurationen jedoch nicht weiter ins Gewicht, ein Durchlauf mit 5 Agenten läuft nicht wesentlich schneller als ein Durchlauf mit 100 Agenten.

Tabelle A.4: Simulationsparameter für Versuch 16-02-[a-d]

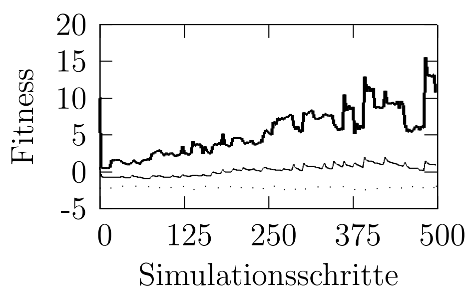
experimentName	16-StatTest02-5-[a-d]
numberOfSteps	500
numberOfRuns	20
randomSeed	5
numberOfAgents	5(a), 50(b), 50(c), 100(d)
neatRoundsPerGeneration	15(a,b,d), 50(c)
worldXSize	100
worldYSize	100
worldDirtMax	2.0
worldDirtRespread	0.1
worldCleanCellThreshold	1.0
worldPheromoneFalloff	0.0
statsReportImagesEveryNRounds	100
statsReportDataEveryNRounds	3

Fitness-Entwicklung Wenn man sich die Fitness-Graphen A.11 der Versuche ansieht und diese miteinander vergleicht, dann fallen mehrere Dinge auf.

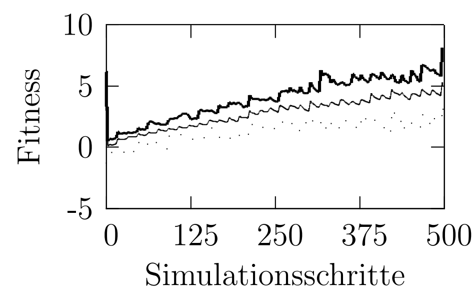
Zunächst einmal kann man beobachten, wie die Fitness mit jeder NEAT-Generation höher wird. Dementsprechend steigt die Fitness-Kurve der Versuche a,b, und d weiter an, als bei Versuch c, da bei letzterem eine Generation 50 Schritte und nicht 15 Schritte dauert. Da so im Laufe der 500 simulierten Schritte nur 10 Generationen entstehen (und nicht 33, wie bei den Anderen) erreichen die Agenten, die sich so entwickeln auch nur eine geringe Fitness.

Betrachtet man jedoch gerade die Kurve aus Versuch c, so fällt auf, dass die Kurve der durchschnittlichen Fitness eine Sägezahnstruktur aufweist. Die einzelnen Zacken der Sägezähne liegen immer 50 Schritte auseinander. Dies entspricht genau einer NEAT-Generation. Mit anderen Worten: mit jeder neuen Generation steigt die Fitness zunächst sprunghaft an und fällt dann kontinuierlich ab, bis sie mit der nächsten Generation wieder steigt. Warum das so ist, muss durch weitere Versuche festgestellt werden.

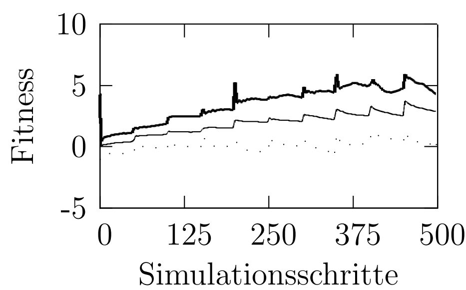
Schlussfolgerungen, Ideen Die Sägezahnstruktur der Fitnesskurve sollte in einem späteren Versuch genauer untersucht werden.



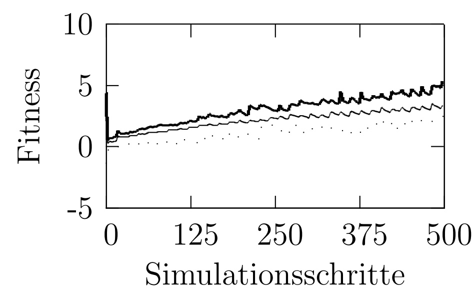
(a) 5 Agenten, 15 Schritte/Generation



(b) 50 Agenten, 15 Schritte/Generation



(c) 50 Agenten, 50 Schritte/Generation



(d) 100 Agenten, 15 Schritte/Generation

Abbildung A.11: Testexperiment 16-02-5(a-d): Die Kurve der durchschnittlichen Fitnesswerte, jeweils über 20 Versuche. Variiert wurden die Anzahl der Agenten, bzw. die Dauer einer NEAT-Generation.

A.4 Versuchsreihe 24

Die Beschreibung dieser Versuchsreihe findet sich im Hauptteil der Arbeit in Abschnitt 5.2.

A.5 Versuchsreihe r02

Das Ziel dieser Versuchsreihe ist es, herauszufinden, wie lange die KNN erprobt und bewertet werden sollten, bevor sie durch eine neue Generation ersetzt werden. Verglichen wurden zunächst die Bewertungszeitraum 2 und 3.

Außerdem soll der Einfluss der add-Node-Mutation (Parameter *add-node-prob*) ausprobiert werden.

A.5.1 Versuch r02-Neat-2-5000

Bei diesem Versuch wurden NEAT-Agenten 5000 Schritte lang evaluiert. Der Bewertungszeitraum (Anzahl von Runden bis zur nächsten Generation) betrug dabei 2 Schritte. Die genauen Versuchsparameter sind den Tabellen A.5 und A.6 zu entnehmen.

Tabelle A.5: Simulationsparameter für Versuch r02-Neat-2-5000

experimentName	r02-NEAT-2-5000
numberOfSteps	5000
numberOfRuns	20
randomSeed	5
numberOfAgents	50
neatRoundsPerGeneration	2
isNeatExperiment	true
worldXSize	100
worldYSize	100
worldDirtMax	2.0
worldDirtRespread	0.1
worldCleanCellThreshold	1.0
worldPheromoneFalloff	0.0
statsReportImagesEveryNRounds	1000
statsReportDataEveryNRounds	10

Abbildung A.12 zeigt die resultierenden Kurven der sauber gehaltenen Zellen über die 5000 Schritte. Der durchschnittliche Wert fällt von Anfangs fast 250 auf ca. 140 und bleibt dort.

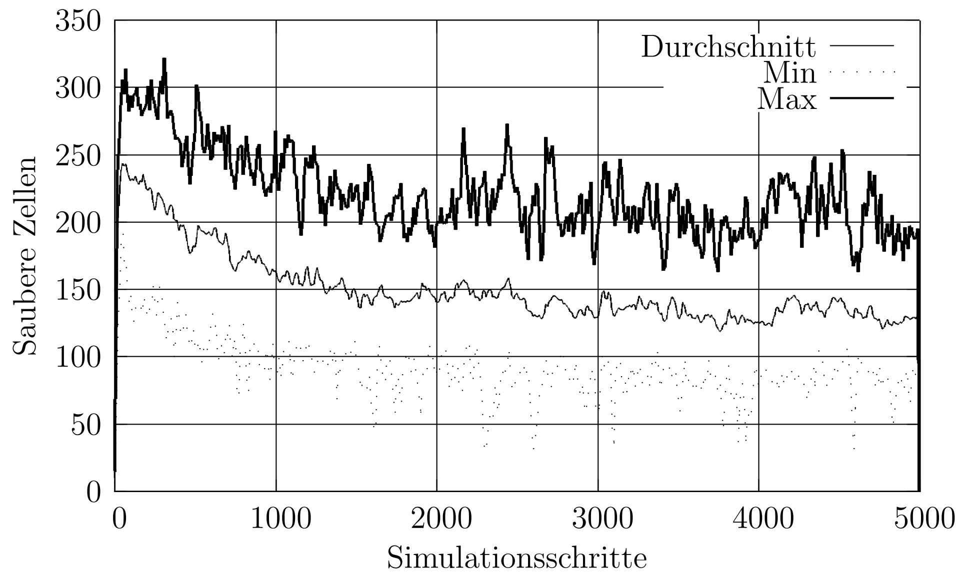


Abbildung A.12: Experiment r02-NEAT-2-5000: Die Zahl der Zellen, die 50 NEAT-Agenten bei einer Bewertungszeit von zwei Schritten sauber halten schwankt zwischen 130 und 150.

A.5.2 Versuch r02-Neat-3-5000

Als Vergleich zu Versuch r02-Neat-2-5000 wurden in diesem Versuch alle Parameter beibehalten, lediglich der Bewertungszeitraum wurde auf 3 Schritte verlängert.

Vergleicht man Abbildung A.12 mit Abbildung A.14, so zeigt sich, dass der durchschnittliche Wert, der sich nach ca. 2500 Schritten einstellt von etwa 140 (bei 2 Schritten) auf etwa 125 (bei drei Schritten) gefallen ist. Daraus lässt sich schließen, dass kürzere Generationen ein besseres Ergebnis liefern.

Bei der Durchsicht der Pfad-, Pheromon- und Verschmutzungskarten fällt weiterhin auf, dass sich in beiden Fällen (2 und 3 Schritte) bei ca. 75% der Läufe Schwerpunkte auf einer der Spielfeldseiten oder in einer Ecke bilden. Dies ist auf Abbildung A.13 zu erkennen.

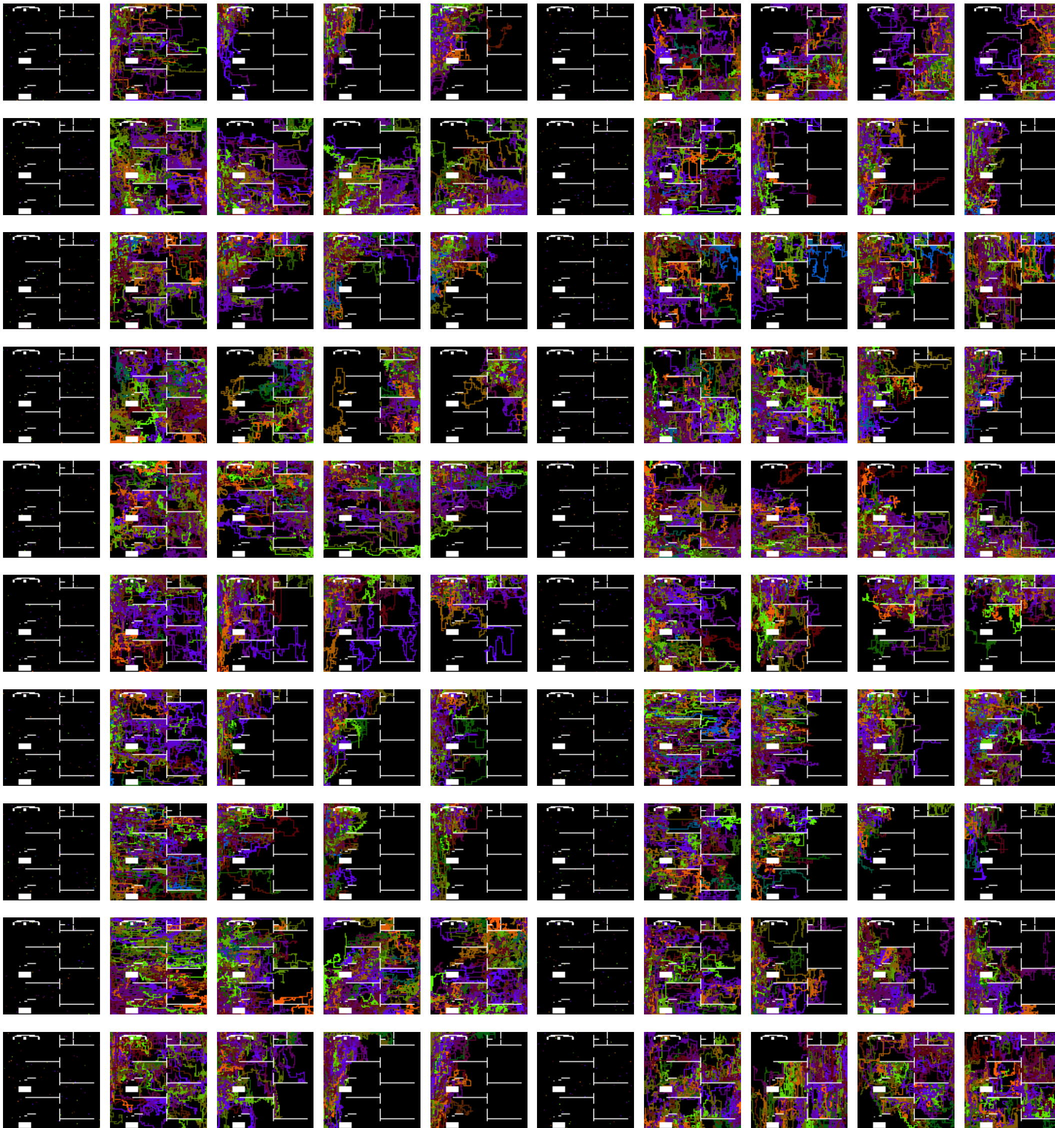


Abbildung A.13: Experiment r02-NEAT-2-5000: 50 NeatAgents laufen 20×5000 Schritte lang über eine Karte mit Hindernissen. Sichtbar sind die Spuren, die einzelnen Agenten. Es fällt auf, dass der Schwarm sich oft auf einer Seite oder in einer Ecke sammelt.

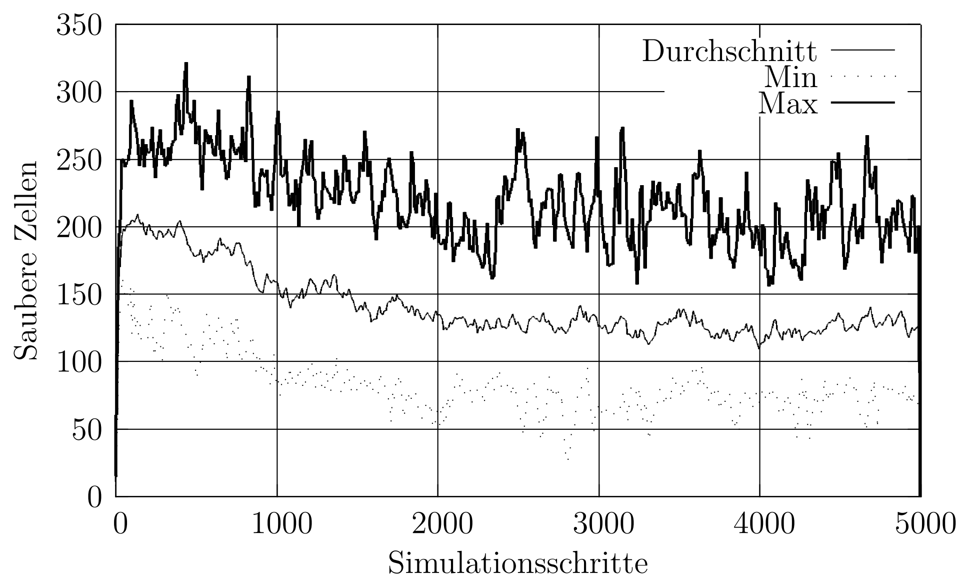


Abbildung A.14: Experiment r02-NEAT-3-5000: Die Zahl der Zellen, die 50 NEAT-Agenten bei einer Bewertungszeit von drei Schritten sauber halten schwankt um 125.

Tabelle A.6: NEAT-Parameter für Versuch

p_trait_param_mut_prob	0.5
p_trait_mutation_power	1.0
p_linktrait_mut_sig	1.0
p_nodetrait_mut_sig	0.5
p_weight_mut_power	1.0
p_recur_prob	0.05
p_disjoint_coeff	1.0
p_excess_coeff	1.0
p_mutdiff_coeff	7.0
p_compat_threshold	9.0
p_age_significance	1.0
p_survival_thresh	0.4
p_mutate_only_prob	0.25
p_mutate_random_trait_prob	0.1
p_mutate_link_trait_prob	0.1
p_mutate_node_trait_prob	0.1
p_mutate_link_weights_prob	0.8
p_mutate_toggle_enable_prob	0.01
p_mutate_gene_reenable_prob	0.0010
p_mutate_add_node_prob	0.01
p_mutate_add_link_prob	0.3
p_interspecies_mate_rate	0.01
p_mate_multipoint_prob	0.6
p_mate_multipoint_avg_prob	0.4
p_mate_singlepoint_prob	0.0010
p_mate_only_prob	0.2
p_recur_only_prob	0.2
p_pop_size	50
p_dropoff_age	15
p_newlink_tries	20
p_print_every	60
p_babies_stolen	0
p_num_runs	1
p_num_trait_params	8

A.6 Versuchsreihe r04

In Testexperiment 2 wurde bereits ermittelt, wie viele Agenten mit einer zufallsgesteuerten Strategie sie gleichzeitig über die Karte bewegen können, ohne sich gegenseitig zu behindern. Es zeigte sich, dass die Behinderung bereits bei 50 Agenten einsetzt.

Von einer Strategie, die, mit Hilfe der Pheromonkommunikation, die eingesetzten Agenten untereinander koordiniert, sollte man erwarten können, dass deutlich mehr Agenten gleichzeitig auf einer Karte eingesetzt werden können, ohne dass sie sich gegenseitig behindern.

Das zu überprüfen war das Ziel dieser Versuchsreihe. Dazu wurden (bei ansonsten gleicher Konfiguration) Versuche mit 12,25,50,100,200,400 und 800 Agenten durchgeführt.

A.6.1 Versuch Nr. r04-Own-*

Es wurden sieben Versuche mit unterschiedlich vielen Own-Agenten durchgeführt. Die genaue Versuchskonfiguration kann Tabelle A.7 entnommen werden. Verglichen wurde jeweils die Zahl der gereinigten Zellen.

Tabelle A.7: Simulationsparameter für die Versuche r04-Own-*

experimentName	r04-Own-*
numberOfSteps	750 (400)
numberOfRuns	20
randomSeed	25
numberOfAgents	12, 25, 50, 100, 200, 400, 800
neatRoundsPerGeneration	3
isNeatExperiment	false
worldXSize	100
worldYSize	100
worldDirtMax	2.0
worldDirtRespread	0.1
worldCleanCellThreshold	1.0
worldPheromoneFalloff	0.0
statsReportImagesEveryNRounds	75
statsReportDataEveryNRounds	5

A.6.2 Ergebnis

Wie auf Abbildung A.15 zu erkennen ist, skaliert der Own-Agent (nahezu) perfekt, der in Versuch 24 errechnete Wert 5.2.1 von ca. 10 sauber gehaltenen Zellen pro Agent wird bei 200 Agenten noch eingehalten, bei 400 sinkt er auf ca. 9,375 und

bei 800 Agenten sinkt er im Verlauf von Anfangs ca. 8,5 auf unter 7,25. Bedacht werden sollte hierbei, dass pro Runde im Schnitt nur 500 Felder verschmutzt werden. Somit steigt bei weiter steigender Agentenzahl der Aufwand ein verschmutztes Feld zu finden immer weiter. Rein rechnerisch müssten bei 1000 Agenten das Feld perfekt sauber gehalten werden können. Dies kann jedoch nicht erreicht werden, da ein Agent (mindestens) 20 Schritte benötigt um die von ihm im Schnitt sauber gehaltenen Felder abzufahren.

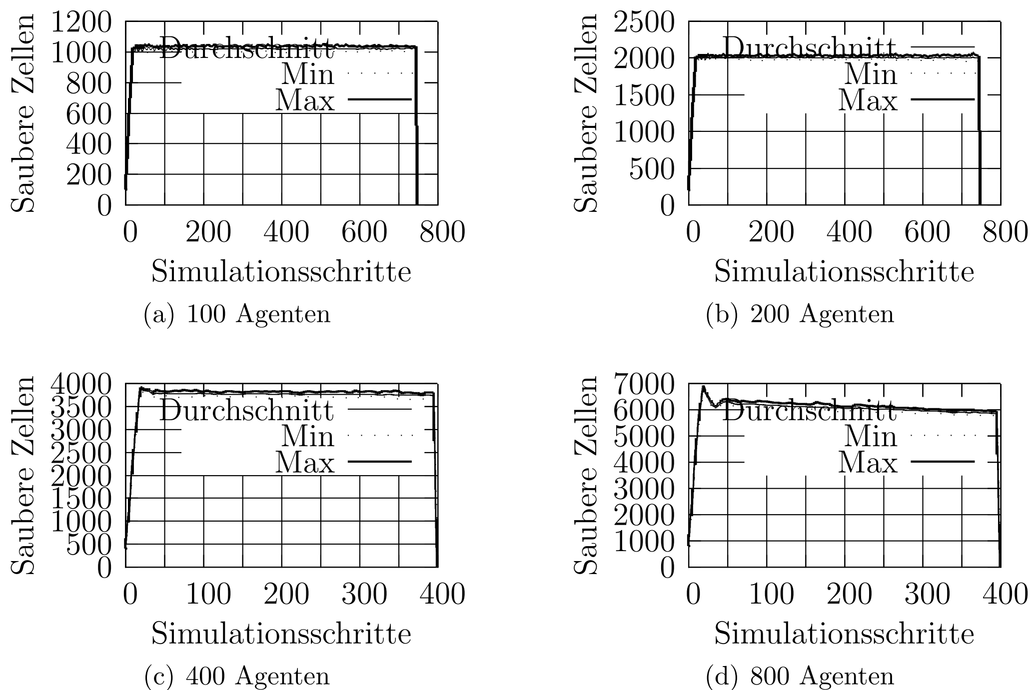


Abbildung A.15: Experiment r04-Own(a-d): Die Kurven der durchschnittlichen Anzahl gesäuberter Zellen, jeweils über 20 Versuche gemittelt. Variiert wurde die Anzahl der Agenten um zu testen, wie gut sich die benutzte Strategie skalieren lässt.

A.6.3 Schlussfolgerung

Soweit aus den angesetzten Bewertungskriterien erkennbar, kann die entwickelte Strategie des Own-Agenten in Bezug auf die Kooperation als gut angesehen werden.

A.6.4 Versuch Nr. r04-Vector*

Die Versuchsreihe r04-Own soll mit der Zufallsstrategie des SimpleVector-Agent verglichen werden. Dazu werden die r04-Own-Versuche noch einmal wiederholt, einziger Unterschied ist, dass der SimpleVector-Agent anstelle des Own-Agents zum Einsatz kommt.

Ergebnis

Der SimpleVector-Agent skaliert zwar ebenso gut, bei 800 Agenten sogar etwas besser als der Own-Agent. Allerdings bleibt seine tatsächliche Leistung weit hinter der des Own-Agent zurück. Er kommt im Schnitt auf 3 (sauber gehaltene Zelle pro Agent (anstelle von 10 beim Own-Agent), was einer Rate von 0,15 gereinigten Zellen pro Runde bzw. einer durchschnittlichen Reinigungsdauer von 6,66 Runden pro Zelle entspricht.

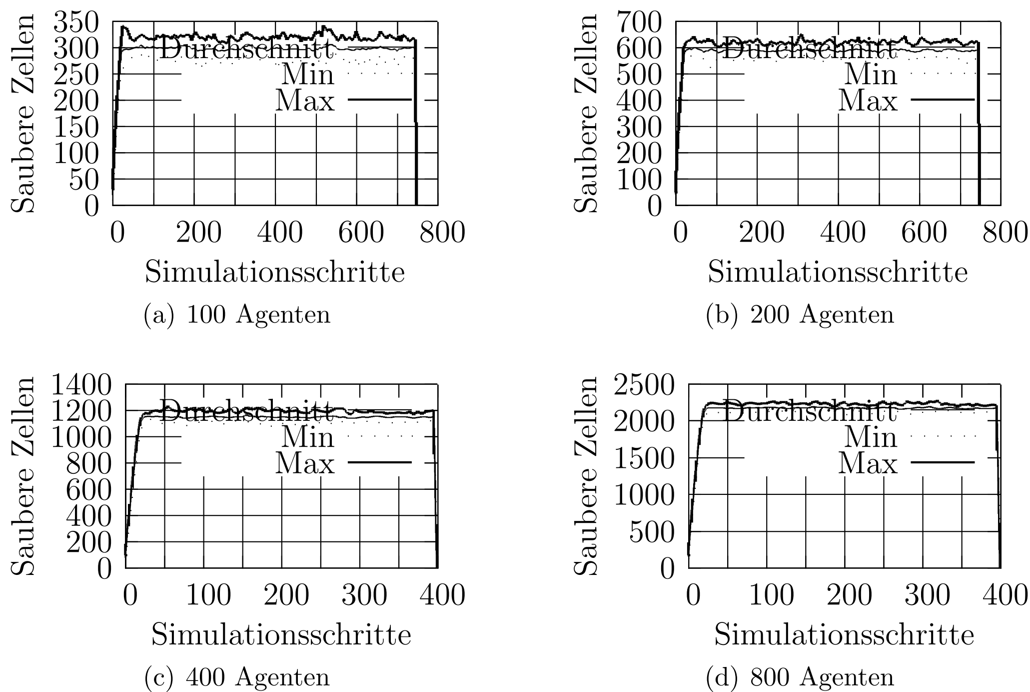


Abbildung A.16: Experiment r04-Vector(a-d): Die Kurven der durchschnittlichen Anzahl gesäuberter Zellen, jeweils über 20 Versuche gemittelt. Variiert wurde die Anzahl der Agenten um zu testen, wie gut sich die benutzte Strategie skalieren lässt.

A.7 Versuchsreihe 25

Nachdem die Leistung der NEAT-Agenten signifikant und dauerhaft sehr weit unter der Leistung der Own-Agenten und auch deutlich unter der der zufallsbasierten SimpleVector-Agenten lag, wurde die Fitnessfunktion geändert bzw. radikal gekürzt. Es gibt nun 10 Punkte für ein gereinigtes Feld, das schmutziger ist als der CleanCellThresshold, für alle anderen Aktionen gibt es keine Punkte.

Dahinter steckt die Idee, die Fitness-Funktion an die CleanCell Zahl zu koppeln.

A.7.1 Versuch Nr. 25-Neat-simple-01

Ein erster Versuch um zu sehen, ob diese Herangehensweise überhaupt Erfolg hat, wurde mit den in den Tabellen A.8 und A.9 genannten Werten durchgeführt.

Tabelle A.8: Simulationsparameter für Versuch 25-Neat-simple-01

experimentName	25-Neat-simple
numberOfSteps	1000
numberOfRuns	10
randomSeed	5
numberOfAgents	50
neatRoundsPerGeneration	5
isNeatExperiment	true
worldXSize	100
worldYSize	100
worldDirtMax	2.0
worldDirtRespread	0.1
worldCleanCellThreshold	1.0
worldPheromoneFalloff	0.0
statsReportImagesEveryNRounds	100
statsReportDataEveryNRounds	5

Ergebnis

Die Leistung ist zwar noch immer nicht annähernd ausreichend, aber auch nicht deutlich schlechter als die andern NEAT-Versuche.

Die Zahl der sauberen Zellen liegt nach 1000 Schritten bei 50 Agenten zwischen **100** und **150**.

Schlussfolgerungen, Ideen

Eventuell sind 1000 Schritte nicht ausreichend, um einen Unterschied zu bemerken. Der Versuch sollte mit einer längeren Laufzeit wiederholt werden.

A.7.2 Versuch Nr. 25-Neat-simple-02

Den Schlussfolgerungen aus Versuch 25-Neat-simple-01 folgend, wurde der Versuch wiederholt, jedoch diesmal über 5000 Runden und mit 100 Agenten. Die höhere Agentenzahl sollte zu einer höheren Genom-Vielfalt führen. Um Rechenzeit zu sparen, wurden dieses mal nur 2 Läufe simuliert. Eine Tendenz (falls vorhanden) sollte sich auch so zeigen.

Außerdem wurde die Wahrscheinlichkeit für eine add-node-Mutation um den Faktor 10 herabgesetzt. Dadurch sollte mehr Stabilität in die Topologie kommen und so mehr Zeit für die Ausformung, das „Feintuning“, der KNN zur Verfügung stehen.

Ergebnis

JNEAT führt sehr oft ein so genannten „DeltaCoding“ durch. Dies geschieht immer dann, wenn eine Spezies sich über einen längeren Zeitraum nicht mehr verbessert hat.

Außerdem ist die Zahl der sauberen Zellen nicht signifikant gestiegen, obwohl die Zahl der Agenten verdoppelt wurde.

Die Überprüfung der Verschmutzungskarten (Dirtmaps) zeigt, dass sich die Agenten auf (je) einer Seite konzentriert haben und dort auch geblieben sind.

Die Zahl der sauberen Zellen liegt innerhalb der ersten 1000 Schritte zwischen 300 und 150, pendelt sich dann aber für die folgenden 4000 Schritte bei ca. **150** ein.

Schlussfolgerungen, Ideen

Um das häufige DeltaCoding zu umgehen sollte der Versuch mit einem erhöhten *dropoff-age* wiederholt werden.

A.7.3 Versuch Nr. 25-Neat-simple-03

Dieser Versuch gleicht dem Versuch 25-Neat-simple-02, nur wurde das droppoff-age von 15 auf 50 erhöht. Dadurch werden Spezies langsamer gelöscht, auch wenn sie für längere Zeit keine Innovationen mehr hervorgebracht haben.

Ergebnis

Erstaunlicherweise steigt die Fluktuation in der Zahl sauberer Zellen. Dies könnte evtl. daran liegen, dass zu wenig Läufe simuliert wurden.

Ebenfalls ist (auf den Path-Maps) wieder eine Konzentration der Population auf einer Kartenseite erkennbar.

Außerdem steigt die Anzahl der hinzugewonnenen *hidden Nodes* der entwickelten KNN. Offenbar wird dies dadurch begünstigt, dass durch das höhere *droppoff-age* auch „unsinnige“ Mutationen länger leben und nicht aussortiert werden.

Die Zahl der sauberen Zellen liegt bei 100 verwendeten Agenten anfangs zwischen 350 und 200 (erste 1000 Schritte), pendelt dann aber zwischen **150** und **200**.

A.7.4 Versuch Nr. 25-Neat-simple-04

Dieser Versuch gleicht wieder dem Versuch 25-Neat-simple-01. Die einzige Änderung zu diesem besteht in der längeren Laufzeit. Alle anderen Änderungen an den NEAT-Parametern wurden rückgängig gemacht.

Ergebnis

Die Zahl der sauberen Zellen liegt für die verwendeten 50 Agenten anfangs zwischen 150 und 100 (erste 1000 Schritte), pendelt sich dann aber um **80** ein.

A.7.5 Versuch Nr. 25-Neat-simple-06

Es wurde vermutet, dass die Bewertungsfunktion in ihrer verwendeten Form (lediglich das Reinigen einer Zelle wird belohnt) *zu* einfach war. Die Agenten hätten vielleicht zu wenig Anreiz gehabt, sich zu bewegen. Deshalb wurde die Bewertung so geändert, dass das Reinigen einer Zelle weiterhin mit 10 Punkten belohnt wurde, eine erfolgreiche Bewegung wurde mit 0.1 Punkt belohnt.

Die steigerte zwar, wie zu erwarten war, den erreichten Fitness-Wert, die Anzahl der sauber gehaltenen Zellen änderte sich jedoch nicht.

A.8 SimpleNeat

Da eine Änderung der Bewertungsfunktion keine Verbesserung erbrachte, musste nach einem anderen Grund für das schlechte Abschneiden der NEAT-Agenten gesucht werden.

Es wurde vermutet, dass die ursprünglich verwendeten 27 Eingabeneuronen einen zu großen Suchraum darstellten. Deshalb wurde die Zahl dieser auf 9 verringert. Es wurde nun nur noch das Dirt-Layer betrachtet, Pheromon-Layer und Agent-Layer blieben unbeachtet.

A.8.1 Versuch Nr. 25-SimpleNeat-01

Es soll festgestellt werden, ob NEAT bei kleinerer Anzahl von Eingaben besser funktioniert. Deshalb haben die SimpleNeat-Agenten nur 9 Eingabeneuronen, die mit dem Dirt-Layer verbunden sind. Auf diese Weise ist das zu entwickelnde Netz viel kleiner und konvergiert evtl. schneller.

Auch dieser Ansatz brachte keine erkennbaren Verbesserungen.

A.8.2 Versuch Nr. 25-Neat-04-GRID

Es soll überprüft werden, ob ein häufiger Wandkontakt das Lernen verbessert. Hintergedanke ist die Überlegung, dass die Agenten nur dann lernen können, wie sie auf eine Wand reagieren, wenn diese auch häufig genug angetroffen wird. Dazu wurde ein Versuch mit 50 NEAT-Agenten durchgeführt, die sich über eine Karte mit einem so genannten *GRID-Obstacle* bewegten.

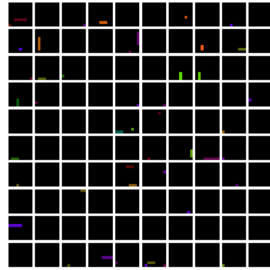


Abbildung A.17: Versuch 25-Neat-04-GRID: Um die Auswirkung von häufigerem Wandkontakt zu überprüfen, wurde die Karte mit einem *Grid-Obstacle* belegt.

Auch dieser Versuch blieb erfolglos.

A.8.3 Versuch Nr. 25-Neat-04-GRID-randomEvaluate

Da verschiedene Methoden, über die zugeordnete Fitness Einfluss auf das Lernverhalten zu nehmen, fehlschlugen, wurde getestet, ob die Fitness überhaupt einen Einfluss auf die erbrachte Leistung hat. Dazu wurde den NEAT-Agenten für jede Aktion ein komplett zufälliger Fitnesswert zugeordnet. Sollte es tatsächlich keinen Zusammenhang zwischen Fitnesswert und Lernverhalten geben, so müsste sich jetzt ein Ergebnis zeigen, welches sich mit den Vorangegangenen vergleichen lässt.

Ergebnis

Die Leistung ist sehr schlecht, die Zahl sauber gehaltener Zellen geht gegen Null.

Schlussfolgerungen, Ideen

Die Fitnessfunktion hat also tatsächlich einen Einfluss auf das Lernverhalten. Evtl. kann durch „tuning“ der Funktion das Verhalten verbessert werden.

A.8.4 Versuch Nr. 25-Neat-05

Bisher wurden den NEAT-Agenten in einer vereinfachten Form der ursprünglichen Bewertungsfunktion 10 Punkte zugeordnet, wenn sie versuchten, eine Zelle zu reinigen. Dies geschah unabhängig davon, ob diese Zelle tatsächlich verschmutzt war. Dies wird nun geändert, so dass ein Agent nun einen Punkt bekommt, wenn er eine Zelle reinigt, die auch tatsächlich schmutzig ist, für eine erfolgreiche Bewegung

erhält er weiterhin 0.1 Punkte, für alle anderen Aktion keine Punkte. Außerdem wurde wieder eine Karte mit GRID-Obstacle verwendet.

Auch dieser Versuch erbrachte keine signifikante Verbesserung.

Tabelle A.9: NEAT-Parameter für Versuch 25-Neat-simple-01

p_trait_param_mut_prob	0.5
p_trait_mutation_power	1.0
p_linktrait_mut_sig	1.0
p_nodetrait_mut_sig	0.5
p_weight_mut_power	1.0
p_recur_prob	0.05
p_disjoint_coeff	1.0
p_excess_coeff	1.0
p_mutdiff_coeff	7.0
p_compat_threshold	9.0
p_age_significance	1.0
p_survival_thresh	0.4
p_mutate_only_prob	0.25
p_mutate_random_trait_prob	0.1
p_mutate_link_trait_prob	0.1
p_mutate_node_trait_prob	0.1
p_mutate_link_weights_prob	0.8
p_mutate_toggle_enable_prob	0.01
p_mutate_gene_reenable_prob	0.0010
p_mutate_add_node_prob	0.01
p_mutate_add_link_prob	0.3
p_max_hidden_nodes	0
p_interspecies_mate_rate	0.01
p_mate_multipoint_prob	0.6
p_mate_multipoint_avg_prob	0.4
p_mate_singlepoint_prob	0.0010
p_mate_only_prob	0.2
p_recur_only_prob	0.2
p_pop_size	50
p_dropoff_age	15
p_newlink_tries	20
p_print_every	60
p_babies_stolen	0
p_num_runs	1
p_num_trait_params	8

A.9 Versuchsreihe 26

Der NEAT-Algorithmus scheint nicht wie erwartet zu funktionieren. Deshalb soll zunächst geprüft werden, ob es möglich ist, ein KNN zu entwerfen, welches die gestellte Aufgabe löst. Dieses KNN wird dem NEAT-Algorithmus dann anstelle von einem zufälligen KNN als Startwert übergeben. NEAT sollte dann im besten Fall die gegebene Lösung durch minimale Änderungen noch weiter verbessern.

Für diese Versuchsreihe wird eine vereinfachte Fitness-Funktion verwendet: das reinigen einer schmutzigen Zelle gibt 10 Punkte, eine Bewegung gibt 5 Punkte, alles andere gibt keine Punkte.

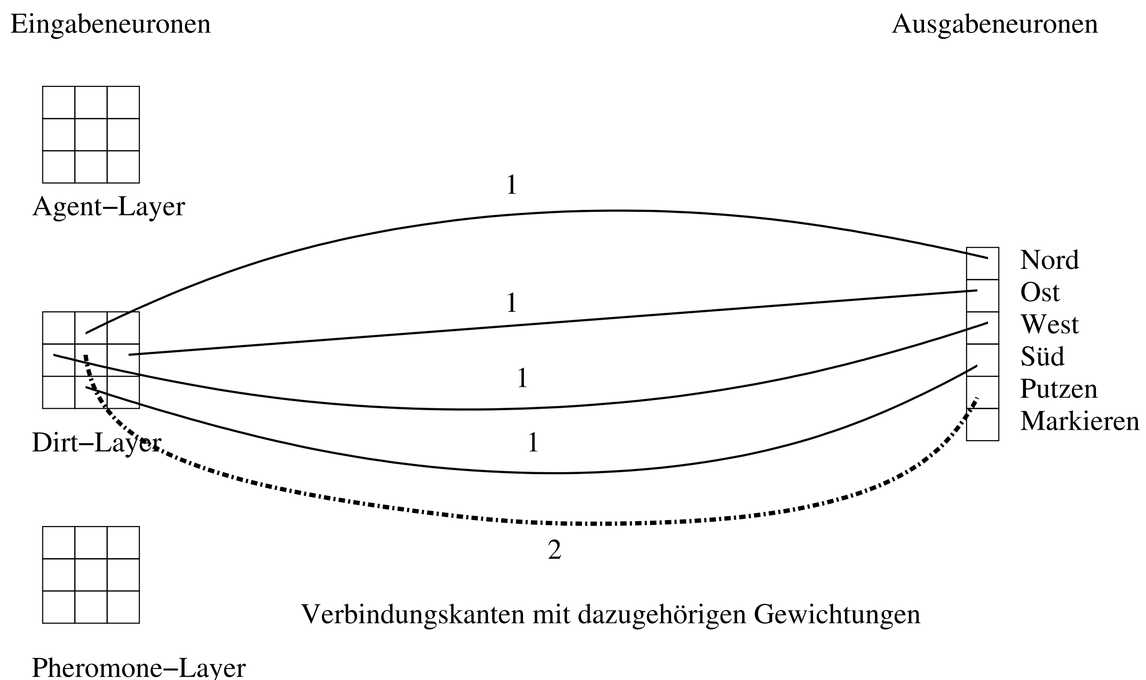


Abbildung A.18: Hier ist das KNN zu sehen, welches verwendet wurde, um in seinem Verhalten dem Own-Agenten nahe zu kommen. Es verknüpft lediglich die Eingangsneuronen einer bestimmten Richtung des Dirt-Layer mit den Ausgangsneuronen der entsprechenden Richtung. Das aktuelle Feld bekommt eine Gewichtung, die doppelt so hoch ist, wie die der anderen Verbindungen, und wird mit dem Putzen-Neuron verbunden. So erhält das Putzen Priorität über dem Weiterbewegen.

A.9.1 Versuch Nr. 26-PrefixedNeat-01-c-test

Es wurde ein sehr einfaches KNN entwickelt, welches in seinem Verhalten dem Own-Agenten ähnelt. Es besteht lediglich aus 5 Verbindungen und kommt ganz ohne

hidden Nodes aus und ist auf Abbildung A.18 zu sehen.

Zunächst wurde das KNN ohne Modifikationen verwendet, der NEAT-Algorithmus war also abgeschaltet. Erste Versuche zeigten, dass das KNN während der ersten 240 Schritte gut funktioniert, danach jedoch „aufgibt“ und die Agenten sich nicht mehr richtig bewegen, sondern nur noch in eine Richtung laufen und dann stehen bleiben.

Dieses Verhalten soll hier untersucht werden. Dazu werden im ersten Versuch 300 Schritte simuliert.

Ergebnis

Alles entwickelt sich linear, ein Absacken ist nicht erkennbar.

Auf den Path-Maps ist erkennbar, dass die Agenten sehr gut laufen (ähnlich wie die Own-Agenten), bis sie in eine saubere Ecke kommen. Dort bleiben sie stehen.

Schlußfolgerungen, Ideen

Der *dirt-respread* Wert war versehentlich auf Null gesetzt worden. Das Stehen bleiben ist bei der Netzkonfiguration zu erwarten, da ohne eine Eingabe (alle umliegenden Felder des Dirt-Layer Null) auch kein Ausgabewert erzeugt wird.

A.9.2 Versuch Nr. 26-PrefixedNeat-01-c-test02

Der Versuch 26-PrefixedNeat-01-c-test wird wiederholt, diesmal wird der dirt-respread Wert auf 0.1 gesetzt.

Ergebnis

Wie ursprünglich beobachtet sackt der CleanCells Wert nach ca. 240 Schritten rapide ab.

Schlussfolgerungen, Ideen

Es wird ein Zusammenhang mit dem respread Wert vermutet.

A.9.3 Versuch Nr. 26-PrefixedNeat-01-c-test03

Versuch 26-PrefixedNeat-01-c-test02 wird wiederholt, der respread Wert wird auf 0.2 verdoppelt.

Ergebnis

Der erwartete Effekt (absinken der Leistung) tritt diesmal schon nach ca. 130 Schritten auf, also doppelt so schnell wie im Vorgängerversuch.

Schlussfolgerungen, Ideen

Da das Fehlverhalten scheinbar in direktem Zusammenhang mit der Wiederverschmutzung steht, wird vermutet, dass das KNN keine vernünftigen Werte mehr liefert, wenn Zellen einen bestimmten Schmutzwert überschreiten.

Dies muss durch debuging untersucht werden.

A.9.4 Versuch Nr. 26-PrefixedNeat-01-c-test04

Unter diesem Versuchsnamen wurden mehrere Debug-Durchläufe durchgeführt.

Fazit: Der Skalierungsfaktor der Sigmoid-Funktion die als Schaltfunktion für die Neuronen-Aktivierung benutzt wurde (vgl. Abschnitt 3.4), war (willkürlich oder aus alten Experimenten der Original-Autors) auf einen Wert von 4,924273 festgelegt. Wurden nun die Verschmutzungswerte der einzelnen Zellen zu groß (größer als ca. 15) so lieferte die Sigmoid-Funktion einen Wert, der so nah an 1 lag, dass er (von Java) auf 1 aufgerundet wurde. Dadurch konnten die Ausgaben des KNN nicht mehr miteinander verglichen werden und es kam zur Blockierung[2].

Der Skalierungsfaktor wurde nun auf 0.1 geändert. So werden auch Schmutzwerte von 60 und mehr noch in den Bereich von 0.99 abgebildet und es kommt zu keiner Blockade.

Ergebnis

Die Agenten laufen wie erwartet mit konstanter Leistung.

Schlussfolgerungen, Ideen

Evtl. könnte man in diesem Beispiel die Sigmoid-Funktion durch eine Lineare Funktion ersetzen, da letzten Endes nur Vergleiche gemacht werden und die Proportionen durch eine Skalierung nicht geändert werden. Ob sich dies jedoch auch auf mehrschichtige Netze übertragen ließe ist fraglich.

A.9.5 Versuch Nr. 26-PrefixedNeat-01-d bis -05

In neun weiteren Versuchen wurde geprüft, ob der NEAT-Algorithmus mit dem erwähnten, selbst entworfenen KNN als Startbelegung bessere Ergebnisse erzielt. Im Laufe der Versuche wurde sowohl das GRID-Obstacle als auch die bisher bekannte Hinderniskarte verwendet.

Abbildung A.19 zeigt einen typischen Verlauf einer Kurve aus diesen Experimenten.

[2] zur Erinnerung: Es werden alle Ausgaben miteinander verglichen und die größtwertige Ausgabe wird als Handlungswunsch interpretiert (vgl. Abschnitt 4.6.3).

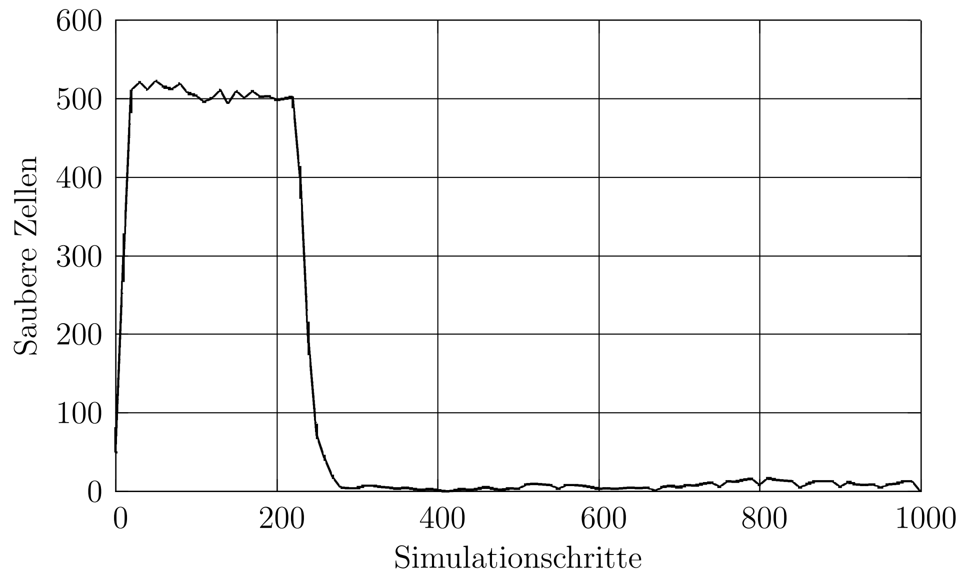


Abbildung A.19: Experiment 26-PrefixedNeat-01-f: Das selbst entwickelte Netz läuft zunächst 200 Schritte lang unverändert, danach beginnt der NEAT-Algorithmus es zu verändern.

A.9.6 Fazit

Ein selbstgestricktes KNN ist in der Lage, gute Ergebnisse zu liefern. Es wird jedoch durch den NEAT-Algorithmus asymmetrisch verändert und verliert deshalb seine Funktion. Die begründet sich darin, dass NEAT die Werte einzelner Verbindungen ändert, bzw. einzelne Verbindungen löscht oder neu einfügt. Basiert die Funktion eines KNN, wie in diesem Fall, auf Symmetrie, so wird dies von NEAT mit hoher Wahrscheinlichkeit zerstört. Aus dem gleichen Grund kann vermutet werden, dass ein symmetrisches KNN von NEAT nur sehr schwer entwickelt werden kann.

A.10 Versuchsreihe 28

Ausgehend von den Ergebnissen der vorangegangenen Versuche wurde der Neat-Agent zum SmartNeat-Agent abgeändert.

Da die Vermutung aufkam, dass eine fehlende Symmetrie der Grund für die schlechte Leistung sein könnte, war das Ziel bei der Entwicklung diesmal, das KNN in einer Weise trainieren zu lassen, die es ihm gestattet, sich zu entwickeln ohne dabei einer bestimmten Richtung den Vorzug zu geben.

Deshalb betrachtet der SmartNeat-Agent jede Himmelsrichtung getrennt, ganz so, als ob er jeweils in diese Richtung blickte. Diese vier Blicke werden dem KNN einzeln vorgelegt, so dass es sich jeweils dafür entscheiden kann, entweder zu reinigen oder aber in eine von *drei* Richtungen zu gehen: vorwärts, links oder rechts. Die Ergebnisse der vier Anfragen werden als Stimmen gewertet, die Aktion, die am Ende die meisten Stimmen hat wird ausgeführt. (Im Falle eines Gleichstands wird immer die erste Variante genommen, die Reihenfolge ist: Nord, Süd, West, Ost, Reinigen. Um zu verhindern, dass (im Fall einer gleichmäßigen Feldverschmutzung) der Agent nur läuft, haben die Stimmen für *clean* doppeltes Gewicht.

Dabei bekommt der SmartNeatAgent für jede Einzelentscheidung nicht mehr die ganze 3×3 Matrix zu sehen, sondern nur noch eine 3×2 Matrix (er kann also nicht „hinter“ sich sehen).

In einer zweiten Version wird der SmartNeat-Agent zum SmartSimpleNeat-Agent vereinfacht. Im Unterschied zum SmartNeat-Agent beachtet er nur das Dirt-Layer; Agent-Layer und Pheromone-Layer werden unbeachtet gelassen. Dies geschieht mit dem Ziel, zu überprüfen, ob die Beachtung der Pheromone einen tatsächlichen Vorteil bringt oder ob sie die Agenten beim Erlernen einer Strategie nur verwirren.

Auch für diese Versuche wurde wiederum die vereinfachte Fitness-Funktion benutzt.

A.10.1 Versuch Nr. 28-SmartNeat-01

Ein erster, kurzer Testlauf (100 Schritte) mit dem SmartSimpleNeat-Agent sollte zeigen, ob er annehmbare Resultate liefert, die Versuchsparameter sind in Tabelle A.10 zu sehen.

Ergebnis

Der CleanCells Wert steigt (nahezu) kontinuierlich auf 250 an. Das ist ein überaus erfreuliches Ergebnis, denn damit wäre erstmals ein NEAT-basierter Agent besser als die Zufallsstrategie des SimpleVector-Agents, die sich bei ca. 150 CleanCells eingependelt hatte. Der Vergleich mit dem Own-Agent, der unter gleichen Bedingungen 500 saubere Zellen erreicht zeigt jedoch, dass die Strategie noch nicht optimal ist.

Tabelle A.10: Simulationsparameter für Versuch 28-SmartNeat-01

experimentName	28-SmartNeat-01
numberOfSteps	100
numberOfRuns	1
randomSeed	5
numberOfAgents	50
neatRoundsPerGeneration	10
isNeatExperiment	true
worldXSize	100
worldYSize	100
worldDirtMax	2.0
worldDirtRespread	0.1
worldCleanCellThreshold	1.0
worldPheromoneFalloff	0.0
statsReportImagesEveryNRounds	1
statsReportDataEveryNRounds	1

Schlussfolgerungen, Ideen

Der Verwendete Agent hatte nur 100 Schritte bzw. 10 Generationen Zeit, um aus einer zufälligen Anfangskonfiguration der KNN eine Strategie zu entwickeln, die besser als der Zufall ist.

In einem weiteren Experiment muss überprüft werden, ob diese Leistung über eine Längere Zeit gehalten werden kann und ob sie vielleicht sogar noch weiter steigt.

Außerdem sollte in einem weiteren Versuch geklärt werden, ob die Nutzung der Pheromone einen tatsächlichen Vorteil bringt.

A.10.2 Versuch Nr. 28-SmartNeat-03

Unter gleicher Konfiguration wurde der SmartSimpleNeat-Agent 1500 Schritte lang getestet, um die Langzeitleistung zu bestimmen. Dabei wurde die Bewertungszeit auf 3 Schritte verkürzt.

Ergebnis

Der Zahl der sauber gehaltenen Zellen pendelt um ca. 280.

Schlussfolgerungen, Ideen

In einem Experiment mit mehreren Läufen muss dieser Wert stabilisiert und verifiziert werden.

A.10.3 Versuch Nr. 28-SmartNeat-04

Bei gleicher Konfiguration wie Versuch 28-SmartNeat-03 kam diesmal ein SmartNeat-Agent zum Einsatz, also ein Agent, der die Pheromonkommunikation nutzt.

Ergebnis

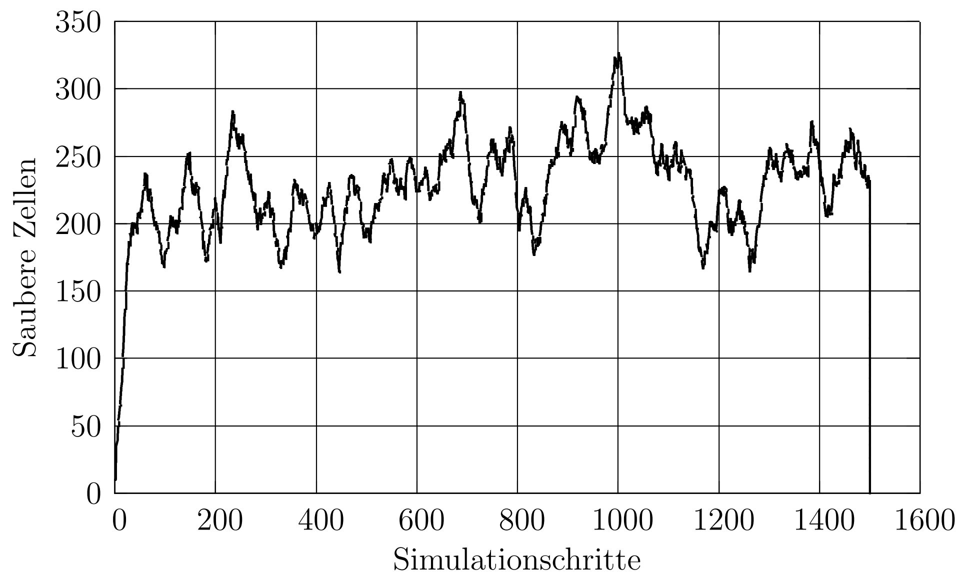


Abbildung A.20: Experiment 28-SmartNeat-04: Die Leistung des SmartNeat-Agenten pendelt zunächst um 200, nach ca. 600 Schritten wird die Varianz der Pendelbewegung jedoch größer.

Wie auf Abbildung A.20 zu sehen ist, pendelt der CleanCells Wert zunächst um 200, nach ca. 600 Schritten wird die Varianz der Pendelbewegung größer und der (geschätzte) Mittelwert steigt auf ca. 220.

Schlussfolgerungen, Ideen

Es ist schwieriger aus je 12 Eingabewerten eine schlüssige Entscheidung abzuleiten als aus 6 Eingabewerten. Deshalb war zu erwarten, dass der SmartNeat-Agent im Gegensatz zum SmartSimpleNeat-Agent langsamer lernt.

Die Streuung ist jedoch viel zu groß um eine belastbare Aussage über seine Leistung zu treffen. Zudem sind 1500 Schritte bzw. 500 Generationen evtl. nicht ausreichend um das schwierigere Verhalten auf Basis von 12 Eingabewerten zu erlernen.

Es muss also ein weiteres Experiment durchgeführt werden, mit erhöhter Laufzeit und mehr Durchläufen.

A.10.4 Versuch Nr. 28-SmartSimpleNeat-05

Um eine Tendenzabschätzung zum Entwicklungspotential abgeben zu können wird der SmartSimpleNeat-Agent für einen Lauf 5000 Schritte bei einer Generationszeit (Bewertungszeit) von drei Schritten simuliert.

Ergebnis

Der CleanCells Wert pendelt im großen und ganzen zwischen 200 und 300. Betrachtet man die Path-Maps, so fällt auf, dass die Agenten sich insgesamt noch immer sehr gleichförmig bewegen. Es ist nicht erkennbar, dass dies gegen Ende des Experiments (wo ein verbessertes Verhalten angenommen werden kann) zugunsten einer individuelleren Bewegungsrichtung abnimmt.

Vergleicht man die Größe der entstehenden Genome, so fällt auf, dass diese nach 5000 Schritten deutlich größer (fast vier mal so viele Gene) sind als nach 1500 Schritten, obwohl sich die Leistung über die Zeit nicht verbessert hat. Dies scheint ein Widerspruch zur Aussage der Autoren zu sein, der NEAT-Algorithmus stelle sicher, dass eine minimale Topologie entwickelt wird.

Schlussfolgerungen, Ideen

5000 Schritte sind eine ausreichend lange Zeit um die Werte zu bewerten, allerdings müssen mehrere Läufe durchgeführt werden, um die Werte durch Mittelung zu stabilisieren.

Außerdem sollte noch ein weiterer Langzeitversuch mit 20.000 Schritten durchgeführt werden, um die Gleichförmigkeit der Bewegung der einzelnen Agenten genauer zu untersuchen.

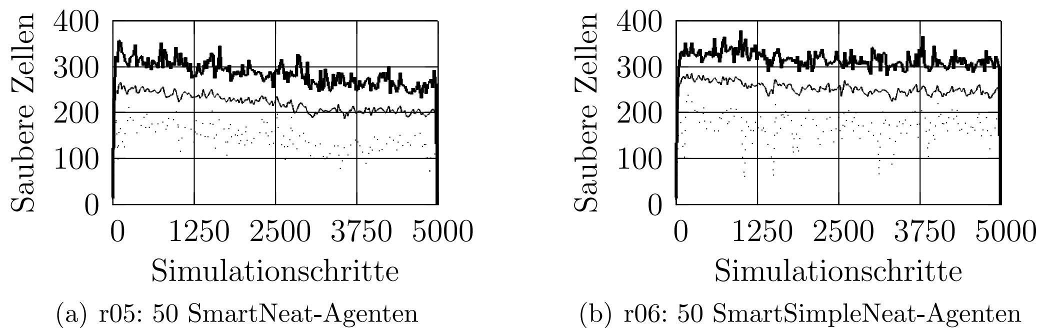
A.10.5 Fazit

Auch wenn die Leistung des SmartNeat-Agenten noch nicht an die optimale Strategie heranreicht, so ist er doch deutlich besser als der Neat-Agent. Somit war diese Versuchsreihe ein Erfolg, das Ziel, den Neat-Agenten durch die Einführung von Symmetrien zu verbessern, wurde erreicht.

A.11 Versuchsreihen r05 und r06

In den Versuchsreihen r05 und r06 ging es einerseits darum, die SmartNeat-Agenten (r05) mit den SmartSimpleNeat-Agenten (r06) zu vergleichen. Gleichzeitig sollte die Langzeitstabilität des Verhaltens geprüft werden und der Einfluss von verschiedenen Mutationswahrscheinlichkeiten noch einmal getestet werden.

Zunächst wurde jeder der beiden Agententypen 25.000 Schritte lang simuliert. Danach wurden kürzere Durchläufe (5.000 Schritte) gemacht, einmal mit einer verringerten Mutationswahrscheinlichkeit für die add-Node-Mutation, und einmal mit einer verringerten Wahrscheinlichkeit für sowohl add-Node- als auch add-Link-Mutationen.



(a) r05: 50 SmartNeat-Agenten

(b) r06: 50 SmartSimpleNeat-Agenten

Abbildung A.21: Vergleich der Ergebnisse aus den Versuchen r05(a) und r06(b): Die Kurven verlaufen gleichförmig. Der einzige bedeutende Unterschied ist, dass die Werte der SmartSimpleNeat-Agenten etwas über denen der SmartNeat-Agenten liegt.

Die Ergebnisse dieser insgesamt 8 Versuche sind unspektakulär. Die Langzeitversuche (25.000 Schritte) ergaben, dass die Leistung der jeweiligen Agenten über die Zeit relativ konstant ist. Auch ergaben die Variationen der Mutationsraten keine erkennbaren Veränderungen. Der einzige Unterschied, der tatsächlich erkennbar war, ist in Abbildung A.21 zu sehen. Hier wird der Verlauf der SmartNeat-Agenten mit dem Verlauf des SmartSimpleNeat-Agenten verglichen. Man kann erkennen, dass die Leistung des SmartSimpleNeat-Agenten knapp 50 Zellen höher ist, als dies beim SmartNeat-Agenten der Fall ist.

Dies unterstützt die Theorie, dass weniger Eingabeneuronen (9 beim SmartSimpleNeat-Agenten gegenüber 27 beim SmartNeat-Agenten) das Lernverhalten positiv beeinflussen.

Was insbesondere bei den Langzeitversuchen auffiel, war die mit zunemender Evolutionszeit immer weiter zunehmende Größe der resultierenden Genome. Lag diese nach 5.000 Schritten (1.666 Generationen) noch bei knapp über 30 Kilobyte, so wuchs sie im Laufe von 25.000 Schritten (8.333 Generationen) auf über 155 Kilobyte an. Die Leistung aus den größeren Genome entwickelten KNN war jedoch nicht besser, also die der Kleineren.

A.12 Versuchsreihe 29

Nachdem mit dem SmartNeat-Agenten die Leistung über das Niveau der Zufallsstrategie gehoben wurde, soll festgestellt werden, ob der evolutionäre Ansatz eine allgemeingültige Lösung hervorbringt oder ob er „selbsttragend“ ist, also keine intelligente Lösung entwickelt sondern lediglich evolutionär auf die jeweilige Situation reagiert.

Dazu wird der evolutionäre Prozess nach einer bestimmten Anzahl von Evolutionsschritten abgebrochen, die Agenten müssen ab diesem Moment mit der Strategie auskommen, die sie bis dahin entwickeln konnten.

A.12.1 Versuch Nr. 29-SmartNeat-01

In diesem Versuch lernt der SmartNeat-Agent 33 Generationen (99 Schritte) lang evolutionär. Danach wird 150 Schritte lang sein Verhalten beobachtet, das sich aus dem letzten Evolutionsschritt ergeben hat.

Ergebnis

Mit dem Ende des Evolutionsprozesses sinkt die Anzahl der sauberen Zellen rapide ab.

Schlussfolgerungen, Ideen

Evtl. wird das „evolutionäre Reagieren“ durch die kurze Generationszeit (3 Schritte) gefördert. Eine längere Generationszeit sollte hier für Klärung sorgen.

A.12.2 Versuch Nr. 29-SmartNeat-02

Der Versuch wird mit der Konfiguration von Versuch 29-SmartNeat-01 wiederholt, mit dem einzigen Unterschied, dass die Generationszeit auf 15 Schritte erhöht wurde.

Ergebnis

Auch hier sinkt der CleanCells Wert mit Ende der Evolution rapide.

Schlussfolgerungen, Ideen

Mit Erhöhung der Generationszeit sinkt die Zahl der Generationen (bei 15 auf 100 Schritten sind es nur noch 6 Generationen). Diese Zeit reicht vermutlich nicht aus um ein akzeptables Verhalten zu entwickeln.

A.12.3 Versuch Nr. 29-SmartSimpleNeat-01

Zum Vergleich mit den Vorgängerversuchen kam noch einmal der SmartSimpleNeat-Agent zum Einsatz. Bei einer Generationszeit von 15 Schritten hatte er 250 Schritte Zeit, eine Strategie zu evolvieren, danach wurde diese für weitere 250 erprobt, ohne sie weiter zu verändern.

Ergebnis

Das Ergebnis fällt wiederum sehr schlecht aus. Auffällig ist jedoch, dass der Clean-Cells Wert von zunächst über 150 fast linear abfällt um sich dann bei ca. 20 wieder zu fangen.

Ebenfalls fällt bei einem Blick auf die Path-Maps auf, dass die Agenten nach Evolutionsende bis zur nächsten Wand weiterlaufen und dort stehen bleiben. Nur selten kommt es vor, dass sie sich dann noch einmal bewegen.

Schlussfolgerungen, Ideen

Vielleicht reichen auch die 16 Generationen für die Bildung einer Strategie noch nicht aus.

Das Stehenbleiben und Wiederloslaufen wird vermutlich durch eine Verschmutzung ausgelöst die zufällig in ihrem Wahrnehmungsbereich entsteht.

A.12.4 Versuch Nr. 29-SmartSimpleNeat-02

Um dem Agenten mehr Zeit zur Bildung einer Strategie zu lassen, wird der Versuch 29-SmartSimpleNeat-01 wiederholt. Diesmal wird für 1500 Schritte (100 Generationen) evolviert und dann 500 weitere Schritte ohne Evolution getestet.

Ergebnis

Wie zuvor fällt der Wert auf ca. 20, sobald die Evolution stoppt.

Die Agenten laufen teilweise jeweils 2-3 Schritte vor und wieder zurück, scheinbar ist dies das beste erlernte Verhalten.

Schlussfolgerungen, Ideen

Es wird vermutet, dass NEAT allein durch die Variationen, die bei jeder neuen Generation erzeugt werden, insgesamt ein zufälliges Verhalten erzeugt, welches für einen Großteil der Leistung verantwortlich ist, die während des Evolutionsprozesses erbracht wird.

A.12.5 Versuch Nr. 29-SmartSimpleNeat-03

Um zu überprüfen, ob allein die zufällige Variation der KNN (anstelle der NEAT-Evolution) eine Leistung erbringen kann, die ähnlich hoch ist, wie die der NEAT-Agenten, werden nun 100 Schritte lang, anstelle eines fitnessbasierten Evolutionsschrittes, zufällige neue Genome erzeugt. Nach Ablauf von 100 Schritten setzt die NEAT-Evolution wie gewohnt ein.

Ergebnis

Der Wert während der Zufallsphase pendelt um 75 und steigt mit Beginn der Evolution auf über 200.

Schlussfolgerungen, Ideen

Zufällig belegte KNN erreichen (unter ständiger Veränderung) 75 sauber gehaltene Zellen, die von NEAT entwickelten Netze (nach Ende der Evolution, ohne weitere Veränderung) ca. 25. Zum Vergleich: Die zufallsbasierten SimpleVector-Agenten halten im Schnitt 150 Zellen sauber.

A.12.6 Fazit

NEAT leistet zwar im Evolutionären Prozess eine zufriedenstellende Leistung, entwickelt jedoch keine lauffähigen Strategien.

Anhang B

CD-ROM

Anhang C

Softwareinstallation

Während der Inbetriebnahme des Repast Frameworks unter Linux kam es zu einigen Fehlern, deren Behebung sehr Zeitaufwendig war. Um dies anderen zu ersparen, sind hier diejenigen zusammengefasst, die im Rahmen dieser Arbeit gelöst wurden bzw. zu der Lösungen im Internet gefunden wurden.

C.1 Bugfixes

Die aktuelle Repast-Version wird mit ein paar bekannten Bugs ausgeliefert, die vor der Benutzung noch behoben werden müssen (zumindest unter Linux). Es handelt sich dabei vor allem um Groß-/Kleinschreibung von Dateinamen und Problem mit verschiedenen Zeichensätzen.

NetworkProjectionDryer.java
: In der Datei

  > repast.symphony.core/src/repast/freezedry/freezedryers/proj/Network-ProjectionDryer.java

  muß in Zeile 90 das „@Override“ entfernt bzw. auskommentiert werden.

HtmlDoc.xpt
: Betroffen ist die Datei

  > repast.symphony.score.modeler/model/score/tmpl/HtmlDoc.xpt

  In dieser Datei wird mit den französischen Anführungszeigen gearbeitet. Durch verschiedene Verwirrungen werden diese Zeichen nicht richtig vom iso8859-1 zu utf-8 konvertiert (vermutlich nur auf Systemen die utf-8 benutzen). Als einfachtes Workaround die betroffene Datei in einem Editor (z.B. kate) öffnen, die richtige Zeichenkodierung wählen (iso8859-1), den gesamten Text kopieren und im Eclipse Fenster wieder einfügen (und dabei den alten, fehlerhaften Text ersetzen).

RepastAspect.xpt
Betroffen ist die Datei

repast.symphony.score.modeler/model/score/tmpl/RepastAspect.xpt

Das vorgehen ist identisch mit dem unter *HtmlDoc.xpt* beschriebenen.

Scenario.xpt
Betroffen ist die Datei

repast.symphony.score.modeler/model/score/tmpl/Scenario.xpt

Das vorgehen ist identisch mit dem unter *HtmlDoc.xpt* beschriebenen.

StarfireExt.jar
Betroffen ist die Datei

repast.simphony.visualization/lib/StarfireExt.jar

Die Datei muß von „StarfireExt.jar“ zu „StarFireExt.jar“ umbenannt werden.

js.JPG
Betroffen ist die Datei

repast.simphony.visualization/icons/js.jpg

Die Datei muß von „js.JPG“ zu „js.jpg“ umbenannt werden.

In beiden Fällen ist die verweisende Datei

repast.simphony.visualization/plugin_jpf.xml

Literaturverzeichnis

[AFB05] Ingrid Aguilar, Alicia Fonseca und Jacobus C. Biesmeijer. Recruitment and communication of food source location in three species of stingless bees (hymenoptera, apidae, meliponini). In *Apidologie 36*, pages 313–324. INRA/DIB-AGIB/ EDP Sciences, 2005.

[ALBD03] Jan C. Albiez, Tobias Luksch, Karsten Berns und Rüdiger Dillmann. Reactive reflex-based control for a four-legged walking machine. *Robotics and Autonomous Systems*, 44(3-4):181–189, 2003.

[BCR+08] Fabio Bellifemine, Giovanni Caire, Giovanni Rimassa, Agostino Poggi, Tiziana Trucco, Elisabetta Cortese, Filippo Quarta, G. Vitaglione, Nicolas Lhuillier, und J. Picault. Jade: Java agent developement framework. Projektseite von Jade, Februar 2008. URL: http://jade.tilab.com/ [Stand: April 2008].

[BDAM04] David Bruemmer, Donald Dudenhoeffer, Matthew Anderson und Mark McKay. Components of swarm intelligence. In *10th International Conference on Robotics and Remote Systems for Hazardous Environments.* Idaho National Laboratory (INL), März 2004.

[BID98] Karsten Berns, Wintried Ilg und Rüdiger Dillmann. Adaptive control of the four-legged walking machine bisam. In *Proceedings of the 1998 IEEE International Conference on Control Applications*, volume 1, pages 428–432, Haid-und-Str., Karlsruhe, Germany;, September 1998.

[BIO08] BIOKON e.V. Biokon – bionik-kompetenz-netz. Informationsseite der BIOKON Vereins, gefördert durch das BMBF – u.a. über 100 Verweise auf Fachbücher zum Thema Bionik, April 2008. URL: http://www.biokon.net [Stand: April 2008].

[Bra93] Valentin Braitenberg. *Vehikel : Experimente mit kybernetischen Wesen.* Rowohlt, 1993.

[Bra94] Valentin Braitenberg, editor. *Entwicklung und Organisation in der Natur ; das Bozner Treffen 1993.* Rowohlt, 1994.

[Bru99] John T. Bruer. *The Myth of the First Three Years: A New Understanding of Early Brain Development and Lifelong Learning.* Simon and Schuster, 100 Front St., Riverside, NJ 08075, 1999.

[BT00] Eric Bonabeau und Guy Theraulaz. Swarm smarts (behavior of social insects as model for complex systems). *Scientific American*, 282(3):72–74,76,77,79, März 2000.

[BTS05] Matthias Bonn, Frederic Toussaint und Hartmut Schmeck. Joschka: Job-scheduling in heterogenen systemen. In Erik Maehle, editor, *PARS Mitteilungen 2005*, pages 99–06. 20. PARS Workshop, Gesellschaft für Informatik, Juni 2005.

[CO73] Francis Crick und Leslie Orge. Directed panspermia. *Icarus. International journal of solar system studies*, 19(3):341–346, 1973.

[CR06] Neil A. Campbell und Jane B. Reece. *Biologie.* Pearson Studium, 6. Aufl., [geringfügig überarb. Nachdr.] , 2006.

[CS92] Patricia S. Churchland und Terence J. Sejnowski. *The computational brain.* MIT Press, Cambridge, MA, USA, 1992.

[DBS07] Alexandre Devert, Nicolas Bredeche und Marc Schoenauer. Unsupervised learning of echo state networks: A case study in artificial embryogeny, 2007. TAO team - INRIA Futurs - LRI/CNRS, Bat 490 - Universite Paris-Sud - France.

[DCG99] Marco Dorigo, Gianni Di Caro und Luca M. Gambardella. Ant algorithms for discrete optimization. *Artif. Life*, 5(2):137–172, 1999.

[Dr.08] Dr. von Göler Verlagsgesellschaft mbH. Roboterversand - der Händler für Haushaltsroboter. Website des Webshops, 2008. URL: http://www.roboterversand.de/ [Stand: April 2008].

[Eic06] Gerrit Eicker. Mähroboter - gartentechnik.de. Website - Newsletter, August 2006. URL: http://www.gartentechnik.de/Forum/Rasenmaeher/Maehroboter/ [Stand: April 2008].

[Fed06] Diego Federici. Evolving a neurocontroller through a process of embryogeny, 2006. Norwegian University of Science and Technology Department of computer and information science N-7491 Trondheim, Norway federici@idi.ntnu.no.

[Gor02] Deborah M. Gordon. The regulation of foraging activity in red harvester ant colonies. *The American Naturalist*, 159(5):509 – 518, Mai 2002.

[Gro08] The Swarm Development Group. The swarm development group wiki. Projektseite des Swarm Projekts, Februar 2008. URL: http://www.swarm.org/ [Stand: April 2008].

[Gru07] Robin Gruna. Analysis of redundant genotype-phenotypemappings - investigating the effect of neutrality on evolvability and robustness. Master's thesis, Institut für Angewandte Informatik und Formale Beschreibungsverfahren (AIFB) der Universität Karlsruhe (TH), Oktober 2007.

[HCN+06] Thomas R. Howe, Nick T. Collier, Michael J. North, Miles T. Parker und Jerry R. Vos. Containing agents: Contexts, projections, and agents. In D. Sallach, C.M. Macal und M.J. North, editors, *Proceedings of the Agent 2006 Conference on Social Agents: Results and Prospects*. The University of Chicago, September 2006.

[HO07] Masato Hirose und Kenichi Ogawa. Honda humanoid robots development. *Philosophical Transactions fo the Royal Society*, 365(1850):11–19, Januar 2007.

[Hor99] Gerda Horneck. Leben, ein kosmisches Phänomen? *DLR-Nachrichten – Magazin des Deutschen Zentrums für Luft- und Raumfahrt / DLR*, 94:16–25, 1999.

[JG00] Kam-Chuen Jim und Lee C. Giles. Talking helps: Evolving communicating agents for the predator-prey pursuit problem. *Artificial Life*, 6(3):237–254, 2000.

[JTH05] Derek James, Philip Tucker und Malcolm Hutson. Anji : Another neat java implementation. Projektseite von ANJI bei Sourceforge.net, Mai 2005. URL: http://anji.sourceforge.net/ [Stand: April 2008].

[KL01] Sven Koenig und Yaxin Liu. Terrain coverage with ant robots: a simulation study. In *AGENTS '01: Proceedings of the fifth international conference on Autonomous agents*, pages 600–607, New York, NY, USA, 2001. ACM Press.

[Koe04] Deddy Koesrindartoto. Setting up for repast, and running a repast stand alone example, 2004. Department of Economics – Iowa State University.

[Lee06] Malrey Lee. A study of evolution strategy based cooperative behavior in collective agents. *Artificial Intelligence Review*, 25(3):195–209, Mai 2006.

[LRH06] Steve Lytinen, Steve Railsback und Tom Howe. How to set up repast in eclipse, Juni 2006.

[Mar04] Mikel Maron. Evolution of natural and artificial swarms, project for animal and machine intelligence. Technical report, Evolutionary and Adaptive Systems, University of Sussex, Januar 2004.

[May04] Brian Mayoh. Multi-agent modelling: Evolution and skull thickness in hominids, 2004. University of Aarhus,Denmark.

[Mef04] Klaus Meffert. Auf Darwins Spuren - Genetische Algorithmen mit Java. *Javamagazin*, 8, 2004.

[Mef05] Klaus Meffert. Einführung in die Genetische Programmierung. *Der Entwickler*, 02, 2005.

[Mil53] Stanley L. Miller. A Production of Amino Acids under Possible Primitive Earth Conditions. *Science*, 117:528–529, Mai 1953.

[MMM+07] Klaus Meffert, Javier Meseguer, Enrique D. Marti, Audrius Meskauskas, Jerry Vos und Neil Rotstan. Jgap: Java genetic algorithms package:. Projektseite von JGAP bei Sourceforge.net, Dezember 2007. URL: http://jgap.sourceforge.net/ [Stand: April 2008].

[MNA05] Michelle McPartland, Stefano Nolfi und Hussein A. Abbass. Emergence of communication in competitive multi-agent systems: a pareto multi-objective approach. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 51–58, New York, NY, USA, 2005. ACM Press.

[MP88] Warren S. McCulloch und Walter Pitts. *Neurocomputing: foundations of research*, chapter A logical calculus of the ideas immanent in nervous activity, pages 15–27. MIT Press, Cambridge, MA, USA, 1988.

[MSMW04] Christian Müller-Schloer, Christoph Malsburg und Rolf Würt. Organic computing. *Informatik-Spektrum*, 27(4):332–336, August 2004.

[MU59] Stanley L. Miller und Harold C. Urey. Organic Compound Synthesis on the Primitive Earth. *Science*, 130:245–251, Juli 1959.

[NHCV05a] Michael J. North, Thomas R. Howe, Nick T. Collier und Jerry R. Vos. The repast simphony development environment. In C.M. Macal, M.J. North und D. Sallach, editors, *Proceedings of the Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms*, 2005.

[NHCV05b] Michael J. North, Thomas R. Howe, Nick T. Collier und Jerry R. Vos. The repast simphony runtime system. In *Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms*. Argonne National Laboratory, Argonne, IL USA, Oktober 2005.

[NSV+06] Michael J. North, Peter Sydelko, Jerry R. Vos, Thomas R. Howe und Nick T. Collier. Legacy model integration with repast simphony. In *Proceedings of the Agent 2006 Conference on Social Agents: Results and Prospects*, pages 95–106, September 2006.

[NTCO06] Michael J. North, Eric Tatara, Nick.T. Collier und Jonathan Ozik. Visual agent-based model development with repast simphony, 2006.

[ONSP07] Jonathan Ozik, Michael J. North, David L. Sallach und Joshua W. Panici. Road map: Transforming and extending repast with groovy, 2007.

[Pao97] Ezequiel A. Di Paolo. An investigation into the evolution of communication. *Adapt. Behav.*, 6(2):285–324, 1997.

[Pao00] Ezequiel A. Di Paolo. behavioral coordination structural congruence and entrainment in a simulation of acoustically coupled agents. *Adapt. Behav.*, 8(1):27–48, 2000.

[Pat97] Dan Patterson. *Künstliche neuronale Netze – Das Lehrbuch*. Prentice Hall, Hans-Pinsel-Straße 9b, 85540 Haar bei München / Germany, 1997.

[PNC+06] Miles T. Parker, Michael J. North, Nick T. Collier, Thomas R. Howe und Jerry R. Vos. Agent-based meta-models. In *Proceedings of the Agent 2006 Conference on Social Agents: Results and Prospects*, September 2006.

[RN04] Stuart J. Russell und Peter Norvig. *Künstliche Intelligenz*. Pearson Studium, 2. edition, 2004.

[SBM05a] Kenneth O. Stanley, Bobby D. Bryant und Risto Miikkulainen. Evolving neural network agents in the nero video game. In *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games (CIG-05)*, 2005.

[SBM05b] Kenneth O. Stanley, Bobby D. Bryant und Risto Miikkulainen. Realtime neuroevolution in the nero video game. *IEEE Trans. Evolutionary Computation*, 9(6):653–668, 2005.

[Sch00] Peter Schmutter. Artificial embryology (Arbeiten von Peter Eggenberer). Referat, Juni 2000.

[Sch05a] Hartmut Schmeck. Organic computing. *Künstliche Intelligenz*, (3/05):68–69, Juli 2005.

[Sch05b] Hartmut Schmeck. Organic computing - a new vision for distributed embedded systems. *isorc*, 00:201–203, 2005.

[Sch05c] Eric Schmitt. U.s. drones crowd iraq's skies to fight insurgents. *The New York Times*, April 2005.

[SK03] Jonas Svennebring und Sven Koenig. Trail-laying robots for robust terrain coverage. In *ICRA*, pages 75–82, 2003.

[SM03a] Kenneth O. Stanley und Risto Miikkulainen. Evolving adaptive neural networks with and without adaptive synapses. In Bart Rylander, editor, *Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 275–282, Chicago, USA, 12–16 Juli 2003.

[SM03b] Kenneth O. Stanley und Risto Miikkulainen. A taxonomy for artificial embryogeny. *Artif. Life*, 9(2):93–130, 2003.

[SM06] Ken Stanley und Risto Miikkulainen. Neuro evolving rootic operatives 2.0. Website, 2006. URL: http://www.nerogame.org/ [Stand: April 2008].

[Sta06] Kenneth O. Stanley. Comparing artificial phenotypes with natural biological patterns. In *GECCO-06, July 8-12, 2006, Seattle, Washington, USA.*, 2006.

[TGT+06] Elio Tuci, Roderich Gross, Vito Trianni, Francesco Mondada, Michael Bonani und Marco Dorigo. Cooperation through self-assembly in multirobot systems. *ACM Trans. Auton. Adapt. Syst.*, 1(2):115–150, 2006.

[TND06] Vito Trianni, Stefano Nolfi und Marco Dorigo. Cooperative hole avoidance in a swarm-bot. In *Robotics and Autonomous Systems 54*, pages 97–103, 2006.

[TNH+06] Eric Tatara, Michael J. North, Thomas R. Howe, Nick T. Collier und Jerry Vos. An introduction to repast simphony modeling using a simple predator-prey example. In *Proceedings of the Agent 2006 Conference on Social Agents: Results and Prospects*, 2006.

[Tse73] Mikhail Lvovich Tsetlin. Automaton theory and the modelling of biological systems. *Mathematics in science and engineering*, 102, 1973.

[VDE] VDE. VDE / ITG / GI - Positionspapier — Organic Computing – Computer- und Systemarchitektur im Jahr 2010. Positionspapier der Gesellschaft für Informatik (GI) und der Informationstechnischen Gesellschaft im VDE (ITG). URL: http://www.vde.com/de/fg/ITG/Arbeitsgebiete/Fachbereich 6/Documents/MCMS/ITGOC2.pdf [Stand: April 2008].

[Vie02] Ugo Vierucci. Neat java (jneat) v1.2 6/24/02 vierucci. Neural Networks Research Group web site, june 2002. URL: http://nn.cs.utexas.edu/soft-view.php?SoftID=5 [Stand: April 2008].

[Wai90] Alex Waibel. Modular construction of time-delay neural networks for speech recognition. *Neural Comput.*, 1(1):39–46, 1990.

[WLB96] Israel A. Wagner, Michael Lindenbaum und Alfred M. Bruckstein. Cooperative covering by ant-robots using evaporating traces, 1996.

[WMC+01] Henri Weimerskirch, Julien Martin, Yannick Clerquin, Peggy Alexandre und Sarka Jiraskova. Energy saving in flight formation. *Nature*, 413(6857):697–698, Oktober 2001.

[ZHB07] Yifeng Zeng, Jorge Cordero H und Dennis Plougman Buus. Swarmarchitect: a swarm framework for collaborative construction. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 186–186, New York, NY, USA, 2007. ACM Press.