Universität Hamburg
Fakultät für Mathematik, Informatik und
Naturwissenschaften
Department Informatik

Diplomarbeit

Entwicklung hierarchischer Klassifikatoren beim Wabenbau von Insekten

Vorgelegt von:
Marius Zirngibl
Matrikelnummer: 5299377

Betreuer:
Prof. Dr. Bernd Page
Prof. Dr. Andreas G. Fleischer

Hamburg, 24. September 2009

Inhaltsverzeichnis

1. Einleitung **9**
1.1. Zielsetzung 9

2. Theoretische Grundlagen **13**
2.1. Nestbau von Bienen und Wespen 13
2.1.1. Stigmergie als Kommunikationsprinzip 14
2.1.2. Nestbau in Insektenstaaten als Komplexes Adaptives System . 16
2.2. Evolutionstheorie 18
2.3. Genetische Algorithmen 21
2.3.1. Funktionsweise 22
2.3.2. Schrittweiser Ablauf 23
2.3.3. Codierung des Genotyps 27
2.3.4. Schematheorie 29
2.4. Learning Classifier Systems 34
2.4.1. Credit Assignment: Reinforcement Learning 38
2.4.2. Rule Discovery: Genetische Algorithmen 39
2.5. Komplexität der Problemstellung 40

3. Methode **43**
3.1. Benutzte Werkzeuge 44
3.2. Konzept des Simulationsmodells 44
3.3. Programmierung der Simulationsumgebung 44
3.3.1. Regelsatz und Parameter 46
3.4. Programmierung der Agentenobjekte 46
3.4.1. Implementation der Klassifikatoren 47
3.4.2. Klassifikation und Manipulation der Umgebung 48
3.4.3. Erweiterung durch hierarchische Klassifikatoren 50
3.4.4. Bewegung der Agenten 52
3.5. Berechnung der Gesamtfitness der Wabenstruktur 52
3.6. Implementation des Genetischen Algorithmus 54
3.6.1. Parameter des Genetischen Algorithmus 55
3.6.2. Ablauf des Genetischen Algorithmus 56
3.7. Graphische Darstellung 58
3.7.1. Darstellung des hexagonalen Gitters 58
3.7.2. Darstellung der einzelnen Wabenzellen 58

3.8. Durchführung der Simulationen 59

4. Simulationsergebnisse **65**
4.1. Ergebnisse mit einem Regelsatz und variablen Stärkewerten 65
4.1.1. Ergebnisse ohne Beschränkung der Populationsgröße 66
4.1.2. Ergebnisse mit beschränkter Populationsgröße 66
4.1.3. Ergebnisse mit Crossover-Wahrscheinlichkeit 68
4.1.4. Ergebnisse mit Mehrfachauswahl der Elternindividuen 69
4.2. Simulation mit statischem Regelsatz ohne Genetischen Algorithmus . 70
4.3. Simulation mit zwei hierarchisch geordneten Regelsätzen 73

5. Diskussion **77**

Literaturverzeichnis **83**

Abbildungsverzeichnis **87**

Tabellenverzeichnis **89**

Anhang **89**

A. MATLAB-Quellcode **91**
A.1. simulationsumgebung.m 91
A.2. bauagent.m 96
A.3. classifier.m 106
A.4. fitness.m 109
A.5. EckPunkte.m 110
A.6. strengthga.m 110
A.7. nachbarn.m 115
A.8. plotgitter.m 115
A.9. plotwabe.m 116
A.10.plotwaben.m 117

Danksagung

Zusammenfassung

Der Nestbau von Wespen und Bienen gilt als ein Beispiel natürlich vorkommmender selbstorganisierender Prozesse. Das einzelne Individuum arbeitet nach einfachen Regeln, die in ihrer Gesamtheit eine Struktur entstehen lassen, die trotz fehlender zentraler Koordination den Anforderungen von Brutpflege, Vorratshaltung und Schutz gegen Feinde bei veränderlichen Umweltbedingungen entsprechen. Im Rahmen dieser Arbeit wird untersucht, inwieweit es möglich ist, mit einem agentenbasierten Simulationsmodell unter Verwendung adaptiver lernender Systeme den Nestbau von Wespen abstrakt nachzubilden. Im Mittelpunkt der Arbeit steht die Frage, inwieweit das Modell in der Lage ist, sich selbständig aufgrund von Informationen aus der Umwelt an gegebene Kriterien anzupassen und sein Verhalten entsprechend zu ändern.

Es wird ein vom realen Nestbau abstrahiertes Simulationsmodell vorgestellt, welches einen Schwarm adaptiver Agenten als Lernendes Klassifizierendes System implementiert, der eine Struktur aus hexagonalen Einzelbausteinen zusammensetzen kann. Das Bauverhalten der Agenten wird dabei durch einfache lokal definierte Regeln (*Klassifikatoren*) bestimmt, welche durch Rückmeldung aus der Simulationsumgebung an das erwünschte Verhalten angepasst werden. Um dem Modell den Bau komplexerer Strukturen zu ermöglichen, wurden hierarchisch geordnete Ebenen von Klassifikatoren eingeführt, die das Verhalten der Agenten in unterschiedlichen Phasen der Simulation steuern.

Anhand des Modells wird im Rahmen dieser Arbeit überprüft, ob die erwähnten Hypothesen bestätigt werden können. Dazu wurden mehrere Simulationsexperimente durchgeführt und deren Ergebnisse analysiert und bewertet.

Abbildung 0.1.: Eine Papierwespe (*Polistes dominulus*) beim Nestbau (Foto: Don Van Dyke, veröffentlicht unter `creative commons` auf `http://www.flickr.com/`)

1. Einleitung

1.1. Zielsetzung

Der Nestbau sozialer Insekten ist ein anschauliches Beispiel für in der Natur auftretende emergente Phänomene. Ameisen, Wespen oder Bienen bringen in kollektiver Arbeit Nestbauten zustande, deren Größe und Komplexität den menschlichen Beobachter bei oberflächlicher Betrachtung zu der Annahme verleiten könnten, dass eine derartige Struktur nicht ohne einen Bauplan oder einen übergeordneten „Architekten“ entstehen kann (e.g. Matsuura und Yamane, 1990; Winston, 1991; Camazine et al., 2001). Da ein einzelnes Insekt nicht die Fähigkeiten besitzt, um eine globale Struktur wie die eines komplexen Nestes zu erfassen, wird im Allgemeinen davon ausgegangen, dass soziale Insekten ihre Nester nach Prinzipien der Selbstorganisation bauen. Das einzelne Individuum arbeitet nach einfachen, genetisch evolvierten lokalen Regeln, die in ihrer Gesamtheit eine Struktur entstehen lassen, die trotz fehlender zentraler Koordination den Anforderungen von Brutpflege, Vorratshaltung und Schutz gegen Feinde bei veränderlichen Umweltbedingungen entsprechen.

Das Ziel der vorliegenden Arbeit ist es, zu untersuchen inwieweit die bei Bienen und Wespen beobachteten Mechanismen des Wabenbaus auf vom Menschen geschaffene lernende Systeme übertragbar sind. Zu diesem Zweck wurde ein vom realen Nestbau abstrahiertes Agentenmodell realisiert, in welchem eine beliebige Anzahl adaptiver Agenten eine wabenförmige Struktur aus hexagonalen Einzelbausteinen zusammensetzen kann. Die Agenten bewegen sich auf einem zweidimensionalen Gitternetz und entscheiden anhand definierter Regeln, ob an ihrer aktuellen Position ein Baustein gesetzt wird oder nicht. Die lokale Umgebung des Agenten gibt hierbei den Ausschlag für die Entscheidung, ob an dieser Position gebaut wird oder nicht. Es wird kein explizites Zielmuster vorgegeben, sondern die entstehenden Wabenstrukturen anhand

von festgelegten Kriterien bewertet und die das Verhalten der Agenten steuernden Regeln aufgrund dieser Bewertungen angepasst.

> „ [. . .] a few simple behavioural rules lead to coordinated building and the production of amazingly complex architectures without needing to develop a strategy.“ (Theraulaz et al., 1999, S.327)

Die adaptiven Agenten verfügen über einen Satz von Regeln (auch *Klassifikatoren*), mit dessen Hilfe sie über ihr konkretes Verhalten in einer bestimmten Situation entscheiden können. Die Klassifikatoren sollen hierbei im Wettbewerb zueinander stehen, wobei der Agent mittels eines internen Bewertungssystems entscheidet, welche der passenden Regeln angewandt werden soll. Die Bewertung der Regeln erfolgt zum größten Teil durch eine numerische Gewichtung, welche die Bedeutung der Regel für das gesamte System repräsentiert. Dieser Wert wird *Stärke* oder *Fitness* der Regel genannt, wobei in dieser Arbeit der Begriff *Stärke* bevorzugt wird, um Verwechslungen zu vermeiden. Um verschiedene Abschnitte im Nestbau von Insekten zu berücksichtigen, soll es mehrere *hierarchisch geordnete* Regelsätze geben. Als *hierarchischer Klassifikator* wird ein Klassifikator bezeichnet, der in eine solche Hierarchie eingebettet ist und somit einer bestimmten Hierarchieebene zugeordnet ist. Durch definierte äußere Triggerereignisse kann von einer zur anderen hierarchischen Ebene von Klassifikatoren umgeschaltet werden. Die Agenten sollen also zu unterschiedlichen Zeitpunkten der Simulation nach verschiedenen Regeln funktionieren, was für den Bau einer komplexeren Neststruktur wichtig ist.

Die Anpassung der Agenten an die Umweltbedingungen erfolgt durch die Anwendung eines *Genetischen Algorithmus* (Goldberg, 1989; Nissen, 1997), der mittels heuristischer Optimierung die Gewichtung der Klassifikatoren ändert. Es werden Regelsätze in der Simulation angewendet und die jeweils entstandenen Strukturen durch den Genetischen Algorithmus bewertet. Diese Variante selbstorganisierten Lernens orientiert sich auch am Konzept der ebenfalls auf Holland zurückgehenden *Learning Classifier Systems* oder kurz *LCS* (Sigaud und Wilson, 2007; Holland, 1992).

Der Wabenbau von Insekten wird hier als Optimierungsproblem betrachtet. Die zentrale Hypothese der vorliegenden Arbeit ist, dass es möglich sein sollte, anhand eines agentenbasierten Simulationsmodells unter Einsatz eines adaptiven lernenden Systems den Nestbau von Insekten abstrakt nachzubilden und die Parameter des Mo-

dells während der Simulation so anzupassen, dass sich die Strukturen in Bezug auf vorgegebene Kriterien immer weiter verbessern, bis ein Optimum erreicht ist.

2. Theoretische Grundlagen

2.1. Nestbau von Bienen und Wespen

Ob man nun Blattschneiderameisen beobachtet, die in weitverzweigten Bauten zu ihrer Nahrungsversorgung Pilze züchten (Hölldobler und Wilson, 1990), oder Termitenarten, die das Innere ihrer Bauten durch ein ausgefeiltes Belüftungssystem auf nahezu konstanter Temperatur halten (Camazine et al., 2001), man stellt sich unweigerlich die Frage, wie und wovon das kollektive Verhalten von Insektenkolonien gesteuert wird. Aus menschlicher Sicht fehlt dem System eine übergeordnete Kontrollinstanz, die dem einzelnen Individuum mitteilt, was es zu tun hat. Daher werden Verhaltensweisen sozialer Insekten gerne als anschauliches Beispiel für *selbstorganisierende Prozesse* herangezogen. Das einzelne Insekt hat eine sehr begrenzte Wahrnehmungs- und Entscheidungsfähigkeit, vollbringt aber im Kollektiv erstaunliche Leistungen. Es ist im Tierreich zu beobachten, dass die Fähigkeit, komplexe Strukturen zu errichten, nicht zwangsläufig mit der Intelligenz des Individuums zunimmt. So sind beispielsweise Schimpansen oder Delphine, denen hohe Intelligenz nachgesagt wird, im Vergleich zu vielen Insekten verhältnismäßig unfähig zur Errichtung von Bauwerken (Deneubourg et al., 1992). Die Eigenschaft selbstorganisierender Systeme, komplexe Strukturen hervorzubringen, welche allein durch die Summe der Fähigkeiten der beteiligten Einzelteile des Systems nicht erklärbar sind, wird als *Emergenz* bezeichnet. Mit anderen Worten,

> „ [. . .] the resulting structure emerges from the collective work of individual organisms that execute simple behaviors based on local information and do not possess a global plan of the end result.“ (Floreano und Mattiussi, 2008, S. 516)

So stellt z.B. der belüftete Termitenbau eine emergente Struktur dar, die der selbstorganisierende Termitenstaat hervorgebracht hat.

Im speziellen Fall des Nestbaus bei sozialen Insekten beschreibt die Emergenz die Beobachtung, dass durch kollektive Zusammenarbeit aller Individuen des Insektenstaates Strukturen entstehen, die in ihrer Komplexität nicht allein durch das Zusammenwirken einzelner Individuen erklärbar sind. Anders als bei vom Menschen errichteten Bauwerken gibt es nach dem Stand der Forschung keine zentrale Koordination, welche den Ablauf des Nestbaus steuert, vielmehr kommunizieren viele gleichberechtigte Individuen direkt oder indirekt miteinander, um die entstehende Struktur zu verändern. Man geht davon aus, dass sich die steuernden Mechanismen im Laufe der Evolution entwickelt haben, doch wie genau diese Mechanismen funktionieren, ist nur in Ansätzen erforscht. Die Bauten vieler Ameisen- und Termitenarten sind beeindruckende Beispiele von Emergenz in biologischen Systemen, wobei sich die vorliegende Arbeit auf den Nestbau auf Basis sechseckiger Waben beschränkt, wie er bei den Kolonien der Honigbiene und verschiedener staatenbildender Wespenarten zu beobachten ist. Viele Wespenarten sind in der Lage, sehr komplexe Nester zu bauen. Werden diese an Ästen von Bäumen oder anderen Orten befestigt, muss für diesen Zweck mit einer Aufhängung, dem sogenannten *Pedikel* (siehe Abbildung 2.1), begonnen werden, die als Grundstein für das Nest dient (Theraulaz und Bonabeau, 1995). Auf diesem Grundstein aufbauend wird das Nest von mehreren Individuen des Wespenstaats kollektiv gebaut. Ein Beispiel für ein fertiges mehrstöckiges Wespennest ist auf Abbildung 2.2 zu sehen, eine vergleichende Darstellung der Nester verschiedener Wespenarten in Abbildung 2.3.

2.1.1. Stigmergie als Kommunikationsprinzip

Es ist davon auszugehen, dass beim Nestbau sozialer Insekten indirekte Kommunikation in Form von *Stigmergie* ein zentrales Steuerungsprinzip darstellt (Bonabeau et al., 1999; Karsai, 1999). Der Begriff Stigmergie wurde von Grassé (1959) geprägt und ist von den griechischen Wörtern $\sigma\tau\iota\gamma\mu\alpha$ (*stigma*: Zeichen, Markierung) und $\epsilon\rho\gamma\sigma\nu$ (*ergon*: Arbeit, Aktion) abgeleitet. Als Stigmergie wird eine Form der indirekten Kommunikation verstanden, die sich dadurch auszeichnet, dass Individuen durch Manipulation ihrer Umgebung Informationen austauschen. Die gesetzten Zeichen oder Markierungen (*stigma*), auch *Stimuli* genannt, beeinflussen also das Ver-

Abbildung 2.1.: Beginn des Nestbaus der Wespenart *Polistes dominulus* mit Bau des Pedikels und der ersten Zelle (Theraulaz und Bonabeau, 1995)

halten bzw. die Aktionen (*ergon*) des Individuums. Bonabeau et al. (1999) unterscheiden hier zwei Arten der Stigmergie: zum einen die *quantitative Stigmergie*, bei der die Stärke der relevanten Stimuli über das Verhalten des Individuums entscheidet, zum anderen die *qualitative* oder *diskrete Stigmergie*, bei der qualitativ verschiedene Stimuli unterschiedliches Verhalten hervorrufen. Die Wegmarkierung mittels Pheromonen, welche von Ameisen zur Anzeige einer Futterquelle genutzt werden, wäre ein Beispiel für quantitative Stigmergie, da die Intensität der Pheromonspur dafür verantwortlich ist, wie stark deren Anziehungskraft auf die einzelne Ameise ist. Ein Beispiel für qualitative Stigmergie stellt der Nestbau verschiedener Wespenarten dar, wobei das Individuum aufgrund unterschiedlicher Wabenstrukturen entscheidet, an welcher Stelle eine zusätzliche Wabe angebaut wird (Karsai, 1999).

Auf Abbildung 2.4 ist zu sehen, wie Stigmergie beim Wabenbau von Wespen schematisch funktioniert. Verschiedene Individuen kommen zeitlich nacheinander an die gleiche Stelle eines Nestes und entscheiden aufgrund der lokal vorgefundenen Wabenstruktur darüber, ob und an welcher Stelle eine neue Wabenzelle gebaut wird.

Abbildung 2.2.: Verlassenes Wespennest der Sächsischen Wespe mit mehreren Ebenen von Waben (von `http://www.honighaeuschen.de/`)

Empirische Untersuchungen haben ergeben, dass die statistische Wahrscheinlichkeit für den Bau einer neuen Zelle bei Wespen von der Anzahl der direkt angrenzenden bereits gebauten Zellen abhängt. Je höher die Anzahl der bereits bebauten benachbarten Zellen ist, desto höher ist auch die Wahrscheinlichkeit für den Bau einer neuen Zelle (siehe Abbildung 2.6). Findet ein Individuum also beispielsweise die in Abbildung 2.5 gezeigte Struktur vor, ist die Wahrscheinlichkeit eines Bauvorgangs an der Position S_3 wesentlich höher als an den Positionen S_2 und S_1.

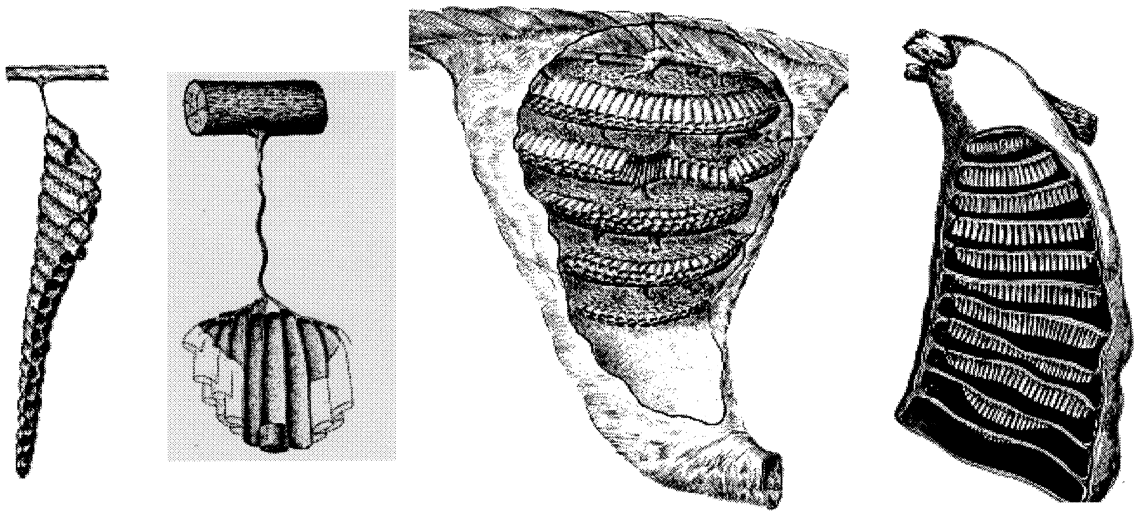
2.1.2. Nestbau in Insektenstaaten als Komplexes Adaptives System

Große Teile der in der vorliegenden Arbeit verarbeiteten Theorie basieren auf dem Werk von John Henry Holland, der mit der Einführung des Begriffs der Komplexen Adaptiven Systeme (engl. CAS, *Complex Adaptive Systems*) einen grundlegenden Beitrag zur Theorie selbstorganisierender Prozesse geleistet hat (Holland, 1975).

Wichtige Eigenschaften Komplexer Adaptiver Systeme sind nach Holland (1992) u.a., dass sie aus sehr vielen hochgradig vernetzten Einzelbausteinen bestehen, was ihr Verhalten schwer vorhersehbar macht. Solche Systeme sind in der Lage, sich an ihre Umwelt anzupassen. Ein Insektenstaat besitzt diese Eigenschaften und kann daher als Beispiel eines natürlich vorkommenden CAS betrachtet werden. Holland stellt auch bereits eine Verbindung zwischen adaptiven Systemen und Optimierungsprozessen her:

> „ Adaptive processes are fundamentally optimization processes made difficult because the structures being modified are complex and their performance is uncertain.“ (Holland, 1976, S. 264)

Diese Ideen stellen eine Verbindung zwischen natürlichen und künstlichen adaptiven Systemen her. Es stellt sich die Frage, inwieweit die Regeln, die das Verhalten sozialer Insekten steuern, und die charakteristischen Eigenschaften natürlicher adaptiver Systeme wie Selbstorganisation und Emergenz auf künstliche adaptive Agentensysteme übertragbar sind. Man könnte die zentralen Fragen folgendermaßen formulieren:



(a) (b) (c) (d)

Abbildung 2.3.: Nester verschiedener Wespenarten im Vergleich. (a) *Ropalidia variegata*, (b) *Mischocyttarus drewseni*, (c) *Angiopolybia pallens*, (d) *Epipona morio* (Theraulaz und Bonabeau, 1995)

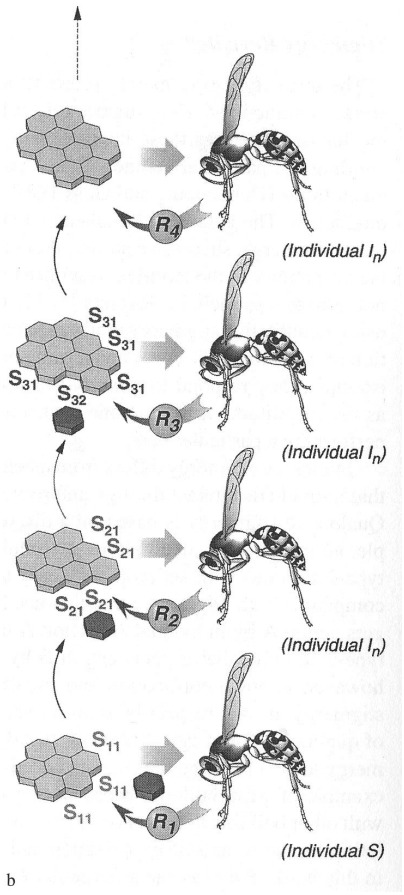


Abbildung 2.4.: Kommunikation durch Stigmergie beim Wabenbau einer Papierwespe (Camazine et al., 2001)

„ [. . .] 'What are the rules governing the insects' behaviour to produce a nest' or 'What rules should govern artificial creatures which must build Notre-Dame'. “ (Deneubourg et al., 1992, S. 123)

2.2. Evolutionstheorie

Das auf Darwin zurückgehende biologische Evolutionsmodell spielte eine entscheidende Rolle bei der Entwicklung der Genetischen Algorithmen, weswegen an dieser Stelle ein kurzer Überblick über die elementaren Begriffe der Evolutionstheorie gegeben werden soll. Die folgenden Darstellungen folgen im Wesentlichen Nissen (1997).

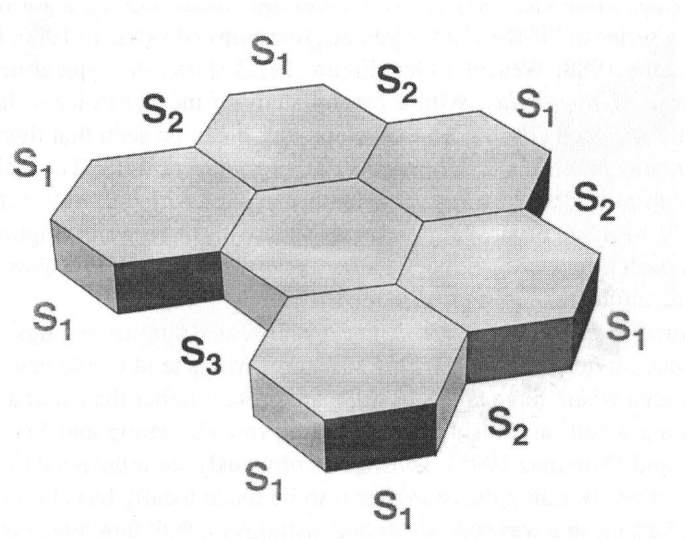


Abbildung 2.5.: Unterschiedliche Anzahl von Nachbarzellen S_1, S_2, S_3 (Camazine et al., 2001)

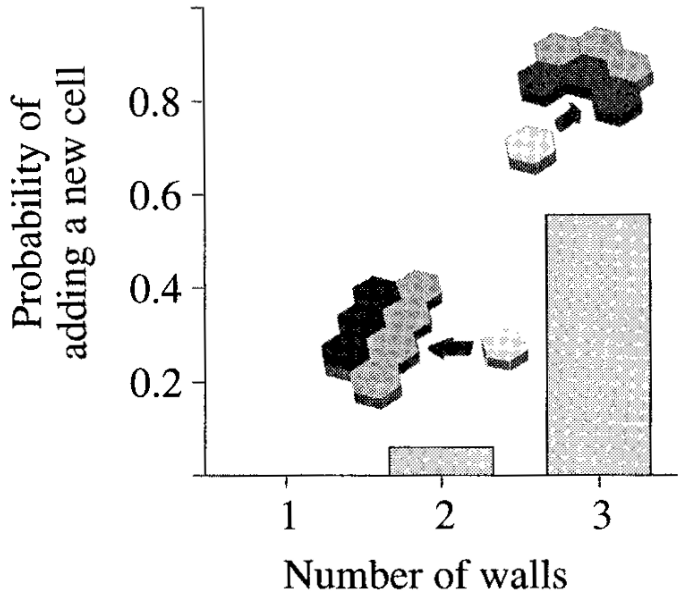


Abbildung 2.6.: Wahrscheinlichkeit für den Bau einer neuen Zelle abhängig von der Anzahl der Nachbarzellen (Camazine et al., 2001)

Die Evolution wird von Darwin (1988) zentral als Zusammenspiel von Mutation als stochastischer Veränderung des Erbguts und Selektion als natürlicher Auslese unter den Individuen beschrieben. Außerdem wird bei der Zeugung von Nachkommen das Erbmaterial beider Elternteile miteinander kombiniert, was *Rekombination* genannt wird. Die Nachkommen unterscheiden sich aufgrund der Mutationen und Rekombi-

nationen genetisch voneinander und weisen unterschiedliche Anpassungsfähigkeit an ihre Umwelt auf. Der Grad der Angepasstheit an die Umweltbedingungen wird als *Fitness* bezeichnet. Im biologischen Zusammenhang lässt sich der Fitness-Wert eines Individuums nicht direkt bestimmen, sondern wird u.a. von seiner Überlebensfähigkeit und der Anzahl seiner Nachkommen abgeleitet. Man geht hierbei davon aus, dass sich nach den Mechanismen der Selektion Individuen mit höherer Fitness häufiger fortpflanzen. Lebewesen die aufgrund ihrer besseren Anpassung an ihre Umgebung eine höhere Überlebenschance haben, können sich mit höherer Wahrscheinlichkeit fortpflanzen, was zu einer natürlichen Auslese (*Selektion*) führt.

Alle Lebewesen haben gemeinsam, dass ihre Erbinformationen als sogenannter genetischer Code in Form von DNA (engl. *deoxyribonucleic acid*) in den Chromosomen in jedem einzelnen Zellkern gespeichert sind. Als Gen wird ein DNA-Abschnitt bezeichnet, welcher unter anderem eine Bauanleitung für die Proteinsynthese darstellt. Die unterschiedlichen Ausprägungen, in denen ein Gen auftreten kann, werden *Allele* genannt. Zur Veranschaulichung betrachte man z.B. ein Gen, das im menschlichen Erbgut die Haarfarbe codiert und in verschiedenen Allelen für blonde, schwarze oder rote Haare ausgeprägt sein kann.

Man bezeichnet die Gesamtheit der Erbinformationen als *Genotyp* und das äußere Erscheinungsbild als *Phänotyp* des Lebewesens. Diese Unterscheidung ist in unserem späteren Kontext insofern relevant, dass es in der Biologie keine zwangsläufige eineindeutige Relation zwischen Genotyp und Phänotyp gibt. Eineiige Zwillinge etwa besitzen identisches Erbgut (gleicher Genotyp), haben aber trotzdem eindeutige Unterscheidungsmerkmale wie z.B. unterschiedliche Fingerabdrücke (unterschiedlicher Phänotyp).

Zusammenfassend kann man festhalten, dass in der biologischen Evolution ein Anpassungsprozess der Arten an ihre jeweiligen Umweltbedingungen stattfindet, wobei sich der genetische Code der Individuen von einer Generation zur nächsten in relativ kleinen Schritten ändert (Weicker, 1999). Die Veränderung biologischen Lebens im größeren Maßstab, wie z.B. die Entstehung neuer Arten, resultiert aus der ständigen Wiederholung kleiner genetischer Änderungen über viele Generationen. Die zentralen Faktoren dieses Evolutionsmodells sind Mutation und Selektion, wobei die Mutation durch Fehler bei der Zellteilung „zufällige“ Veränderungen einzelner Elemente des Erbmaterials verursacht und die Selektion die natürliche Auslese beschreibt, die zur

Folge hat, dass sich besser angepasste Individuen mit höherer Wahrscheinlichkeit fortpflanzen können und sich somit auch deren Erbgut weiter verbreitet. Mutationen treten nach heutigem Wissensstand nicht zielgerichtet auf, wohingegen durch Selektion die zufälligen Änderungen im Genpool nach Verbesserungen und Verschlechterungen sortiert werden.

Obwohl die Evolution als Anpassungsprozess betrachtet werden kann, muss sie klar von klassischen mathematischen Optimierungsproblemen abgegrenzt werden, da es kein globales Optimum gibt, auf das die Evolution zwangsläufig zusteuert. Die Evolution füllt vielmehr möglichst viele biologische Nischen aus, was an der weltweiten Artenvielfalt sehr anschaulich wird.

Nach einer Theorie von Eigen und Schuster (1979) könnten selektive Selbstorganisation und Evolution biologischer Makromoleküle die Grundlage biologischer Information darstellen. Die Selbstreproduktion von Informationsträgern wird hier als autokatalytische Reaktion verstanden. Nach Darwin ist das Zusammenwirken von Mutation und natürlicher Selektion die Basis der biologischen Evolution. Eigens Theorie wird aufgrund der Parallen zu Darwins Evolutionstheorie auch *Molekular-Darwinismus* genannt. Die Idee des *survival of the fittest* wird hierbei von der Ebene der Individuen auf die genetische Ebene übertragen, was bedeutet, dass sich auch nur diejenigen Gene ausbreiten, welche die größte Zahl von Nachkommen erzeugen.

2.3. Genetische Algorithmen

Genetische Algorithmen sind heuristische Optimierungsverfahren, die in ihrer Funktionsweise versuchen, in der biologischen Evolution beobachtete Anpassungsmechanismen nachzuahmen. Sie arbeiten auf Grundlage einer Menge von möglichen Lösungen für ein spezifisches Problem, wobei in Anlehnung an die biologischen Begriffe die einzelne Lösung Individuum und die Gesamtmenge der Lösungen Population genannt wird. Das Ziel eines Genetischen Algorithmus ist es, aus einer großen Anzahl möglicher Lösungen für ein Problem eine oder mehrere möglichst gute Lösungen herauszufinden. Besonders interessant ist die Methode zur Lösung von Problemen mit hoher Komplexität und dementsprechend großem Zustandsraum, da eine analytische Lösung derartiger Probleme oft nicht möglich ist.

Das Grundkonzept der Genetischen Algorithmen basiert im Wesentlichen auf den Arbeiten von John Holland, der sich als einer der ersten Wissenschaftler mit der Beschreibung und Analyse adaptiver Systeme beschäftigt hat. Den Entwurf des Konzepts, welches heute unter dem Begriff *Genetischer Algorithmus* (im Folgenden auch GA genannt) große Verbreitung gefunden hat, nannte Holland ursprünglich *reproductive plan* (Holland, 1992). Obwohl oder vielleicht gerade weil dieses Konzept eine sehr einfache Version eines Genetischen Algorithmus darstellt, soll mit einer detaillierten Vorstellung dieses in der heutigen Literatur oft *kanonischer GA* genannten Verfahrens begonnen werden. Anschließend soll auf einige ausgewählte interessante Erweiterungen und Modernisierungen sowie Hollands *Schematheorem* eingegangen werden.

2.3.1. Funktionsweise

Die hier gewählte Darstellung der elementaren Funktionsweise eines Genetischen Algorithmus orientiert sich im Wesentlichen am sogenannten *kanonischen GA*, wie er bei Nissen (1997) vorgestellt wird. Jedes einzelne Individuum $\vec{a}$ wird intern als Bitstring dargestellt, dessen Länge L je nach Anwendung auf eine feste Anzahl von Bits festgelegt wird. Hier wird von einer binären Darstellung ausgegangen, das heißt die Bits können entweder den Wert 0 oder 1 haben:

$$\vec{a} = (a_1, a_2, \ldots, a_{L-1}, a_L) \in \{0, 1\}^L$$

Der Bitstring $\vec{a}$ wird *Genotyp* des Individuums genannt. Jeder Genotyp wird in n Segmente variabler Länge aufgeteilt. Analog zur Betrachtung des gesamten Strings als interne Repräsentation einer Lösung entspricht jedes Segment der internen Repräsentation einer Variable des zu lösenden Optimierungsproblems.

Man betrachtet bei einem Optimierungsproblem eine zu maximierende Zielfunktion $F(\vec{x})$, die im Kontext Genetischer Algorithmen auch *Fitnessfunktion* genannt wird. $\vec{x} = x_1, x_2, \ldots, x_n$ stellt den Vektor aller Variablen der Funktion dar. Da durch die binäre Codierung die Anzahl verschiedener Werte für eine Variable beschränkt ist, muss für jede Variable x_j mit $(j = 1, 2, \ldots, n)$ festgelegt werden, welchen Wertebereich sie annehmen können soll. Auf die detaillierten Verfahren binärer Codierungen wird in

Abschnitt 2.3.3 noch näher eingegangen. Ziel des GA ist es, einen Eingabevektor $\vec{x}$ zu finden, der die Fitnessfunktion maximiert.

In Anlehnung an das biologische Vorbild wird der Bitstring auch als *Chromosom*, eine bestimmte Bitposition im String als *Gen* und deren konkreter Wert analog dazu als *Allel* bezeichnet. Eine mögliche Lösung wird *Individuum* genannt, wobei hier das konkrete Ergebnis einer einzelnen Lösung als *Phänotyp* und ihre interne binäre Repräsentation als *Genotyp* des Individuums bezeichnet wird.

2.3.2. Schrittweiser Ablauf

Im Folgenden soll der Ablauf eines Genetischen Algorithmus schrittweise beschrieben werden.

Initialisierung

Als erster Schritt muss eine initiale Population P von μ Individuen $\vec{a}_i$ mit $(i = 1, 2, \ldots, \mu)$ erzeugt werden, wobei die Anzahl μ der Individuen je nach Anwendung sehr unterschiedliche Werte besitzen kann. In den meisten Fällen wird die Ausgangspopulation stochastisch erzeugt, d.h. sämtliche Bits aller Genotypen $\vec{a}_i$ werden in allen erzeugten Individuen stochastisch unabhängig voneinander per Zufall auf 0 oder 1 gesetzt.

Bewertung der Lösungen

Um einen Genetischen Algorithmus erfolgreich einsetzen zu können, benötigt man eine geeignete Fitnessfunktion F. $\vec{a}_i$ wird von F auf seinen Fitnesswert abgebildet. Geht man von einer zu maximierenden Zielfunktion F aus, so folgt aus einem hohen Fitnesswert eine gute Eignung des Individuums für das zu lösende Problem. Der Erfolg eines Genetischen Algorithmus hängt in hohem Ausmaß von der Wahl einer geeigneten Fitnessfunktion ab.

Selektion und Replikation

Im Selektionsschritt werden aus der aktuellen Population P diejenigen Individuen ausgewählt, welche als „Eltern“ für die nächste Generation fungieren sollen. Es werden n Individuen stochastisch ausgewählt, wobei n eine gerade Zahl zwischen 2 und μ sein muss. Jedes Individuum $\vec{a}_i \in P$ besitzt eine spezifische Selektionswahrscheinlichkeit p_s:

$$p_s(\vec{a}_i) = \frac{F(\vec{a}_i)}{\sum_{j=1}^{\mu} F(\vec{a}_j)}$$

Dieses Selektionsverfahren wird fitnessproportionale Selektion genannt und zählt zu den *nicht diskriminierenden* Selektionsverfahren (engl. *not extinctive*), so genannt da jedes Individuum eine positive Selektionswahrscheinlichkeit besitzt und damit eine theoretische Chance auf Reproduktion besitzt. Einzelne Individuen dürfen beliebig oft gewählt werden, was zur Folge hat, dass sich in der Praxis Individuen mit hohem Fitnesswert mehrmals und solche mit niedrigem Fitnesswert selten bis gar nicht „fortpflanzen“ können.

In der Literatur wird hierzu gerne der anschauliche Vergleich mit einem Glücksrad benutzt, welches in μ Abschnitte eingeteilt ist. Jeder Abschnitt entspricht einem Individuum aus P und hat eine der Selektionswahrscheinlichkeit des Individuums entsprechende Breite auf dem Glücksrad. Das Rad wird n-mal gedreht und das Ergebnis jeder Drehung festgehalten. Aufgrund des Vergleichs wird dieses Auswahlverfahren auch *Roulette-Wheel-Selektion* genannt.

Von den zur Reproduktion ausgewählten Individuen aus der Population P werden Kopien erstellt und in einen sogenannten *mating pool* $\mathcal{M}_t$ gelegt, in welchen n Individuen passen. Nach n Selektionsvorgängen erhält man ein $\mathcal{M}_t$, das mit n Kopien von Individuen aus P gefüllt ist. Dabei darf jedes Individuum aus P beliebig oft vom Selektionsalgorithmus ausgewählt und in $\mathcal{M}_t$ kopiert werden, weswegen man von einer stochastischen Auswahl *mit Zurücklegen* spricht. Der beschriebene Kopierprozess wird als *Replikation* bezeichnet.

Reproduktion

Die folgenden Schritte 1 bis 4 müssen jeweils $\frac{n}{2}$-mal durchlaufen werden, da in jedem Durchlauf zwei Individuen aus $\mathcal{M}_t$ reproduziert werden.

Schritt 1: Partnerwahl Es werden zufällig zwei Elternindividuen $\vec{a}_{E_1}$ und $\vec{a}_{E_2}$ aus $\mathcal{M}_t$ ausgewählt, aus denen in den nächsten drei Teilschritten zwei Nachkommen entstehen werden. Dabei wird in dieser einfachen Variante des GA jedes Individuum in $\mathcal{M}_t$ mit gleicher Wahrscheinlichkeit $\frac{1}{n}$ gezogen. Die Fitness der einzelnen Individuen wird bei der Auswahl nicht mehr berücksichtigt, weil schon die Wahrscheinlichkeit für die Übernahme in $\mathcal{M}_t$ von den jeweiligen Fitnesswerten abhängt und Individuen mit außergewöhnlich hoher Fitness auch mehrmals in $\mathcal{M}_t$ vorhanden sein können.

Die stochastische Auswahl aus $\mathcal{M}_t$ erfolgt „ohne Zurücklegen“, das bedeutet jedes Individuum kann nur einmal gewählt werden und steht im nächsten Durchlauf nicht mehr zur Verfügung.

Schritt 2: Crossover Das Crossover im Genetischen Algorithmus entspricht der Rekombination des Erbguts zweier sich miteinander fortpflanzender Lebewesen in der Natur. Es wird als zentraler Operator des GA betrachtet, da durch das Crossover die größten Veränderungen im digitalen Genpool passieren.

Im hier betrachteten kanonischen GA wird als einfachste Variante *Ein-Punkt-Crossover* verwendet. Es wird im Vorfeld eine Wahrscheinlichkeit p_{xover} festgelegt, die bestimmt, mit welcher Wahrscheinlichkeit bei der Reproduktion ein Crossover stattfinden soll. Es wird ein stochastischer Auswahlprozess benutzt, der mit Wahrscheinlichkeit p_{xover} für ein Crossover und mit Wahrscheinlichkeit $1-p_{xover}$ gegen ein Crossover entscheidet. Im Allgemeinen wird p_{xover} in einer Größenordnung von $p_{xover} \geq 0,6$ gewählt. Bei zu groß gewähltem p_{xover} besteht die Gefahr, dass sinnvolle Ergebnisse und auftretende Schemata zerstört werden (siehe Abschnitt 2.3.4).

Entscheidet der Zufall gegen ein Crossover, so werden die beiden Elternstrings ($\vec{a}_{E_1}$ und $\vec{a}_{E_2}$) unverändert in zwei Nachkommenstrings ($\vec{a}_{K_1}$ und $\vec{a}_{K_2}$) kopiert:

$$\vec{a}_{K_1} = \vec{a}_{E_1}$$
$$\vec{a}_{K_2} = \vec{a}_{E_2}$$

Soll ein Crossover stattfinden, so wird stochastisch ein Crossover-Punkt c zwischen 1 und $L-1$ festgelegt. Beim Kopieren der Elternstrings in die Nachkommenstrings werden nun jeweils die Bits hinter dem Crossover-Punkt mit denen des Reproduktionspartners ausgetauscht, was zur Folge hat, dass zwei Individuen entstehen, die aus dem Erbgut der Eltern zusammengesetzt sind:

$$\vec{a}_{K_1} = (a_{E_1,1}, a_{E_1,2}, \ldots, a_{E_1,c}, a_{E_2,c+1}, \ldots, a_{E_2,L})$$
$$\vec{a}_{K_2} = (a_{E_2,1}, a_{E_2,2}, \ldots, a_{E_2,c}, a_{E_1,c+1}, \ldots, a_{E_1,L})$$

Schritt 3: Mutation Im Allgemeinen wird die Mutation als weniger bedeutend für die Funktion eines Genetischen Algorithmus eingestuft als das Crossover. Sie ist allerdings wichtig um zu gewährleisten, dass die Population auch dann noch Veränderungen aufzeigt, wenn sich alle Individuen weitgehend ähnlich sind und durch Rekombination kaum mehr Veränderungen zu erreichen sind.

Für die Mutation wird ebenso eine feste Wahrscheinlichkeit festgelegt wie für das Crossover, nur dass die Mutationswahrscheinlichkeit p_{mut} meist sehr viel kleiner gewählt wird. Übliche Werte sind etwa $p_{mut} = 0,01$ oder $p_{mut} = 0,001$. Jedes Bit in jedem Individuum der Population wird nun mit dieser Wahrscheinlichkeit p_{mut} invertiert. Ein Wert von $p_{mut} = 0,01$ hätte beispielsweise zur Folge, dass im Mittel eines von hundert Bits invertiert würde.

Schritt 4: Übergang in die neue Population Die beiden Nachkommen $\vec{a}_{K_1}$ und $\vec{a}_{K_2}$ werden in die neue Population übernommen. Für jeden Nachkommen wird ein Individuum aus der ursprünglichen Population entfernt, wobei die Auswahl der zu entfernenden Individuen analog zur Auswahl der Elternindividuen erfolgt, mit dem Unterschied, dass Individuen mit niedrigerem Fitnesswert mit höherer Wahrscheinlichkeit entfernt werden.

Die Schritte 1 bis 4 werden wiederholt, bis alle Individuen aus $\mathcal{M}_t$ verarbeitet wurden. Die Populationsgröße μ bleibt dabei konstant, weil für jedes neu entstandene Individuum ein anderes Individuum aus der Population entfernt wird.

Abbruchkriterium

Die Schritte Bewertung, Selektion, Replikation und Reproduktion werden nun wiederholt, bis ein festgelegtes Abbruchkriterium greift. Abbruchkriterien können verschiedenartig sein, z.B. kann man eine Maximallaufzeit oder eine maximale Generationenzahl t_{max} definieren, nach deren Ablauf der Algorithmus terminiert. Alternativ kann man die Termination des GA auch von Qualitätsmerkmalen des Ergebnisses abhängig machen. Wenn z.B. über mehrere Generationen keine Verbesserung in den Ergebnissen mehr beobachtet werden kann oder sich die Individuen einer Population weitgehend gleichen, ist meist keine Verbesserung mehr zu erwarten und ein Abbruch vermutlich sinnvoll. Diese Festlegung eines Abbruchkriteriums ist notwendig, da Genetische Algorithmen mathematisch gesehen nicht konvergieren. Nach Termination des Genetischen Algorithmus wird im Allgemeinen die beste der gefundenen Lösungen (sprich das Individuum mit dem höchsten Fitnesswert) ausgegeben.

2.3.3. Codierung des Genotyps

Bei den meisten Varianten Genetischer Algorithmen wird eine binäre Form der Lösungsrepräsentation oder Codierung verwendet. Es muss festgelegt werden, welcher Art der codierte Wert überhaupt ist, da ein Bitstring theoretisch alles mögliche repräsentieren kann, z.B. eine Fließkommazahl, eine ganze Zahl, oder auch einen Buchstaben oder einen Farbwert. Auch wenn in unserem Zusammenhang hauptsächlich numerische Werte interessant sind, muss doch festgelegt werden, in welchen Grenzen sich die codierten Werte befinden, und ob sie als Ganzzahlen oder Fließkommazahlen zu interpretieren sind. Selbst bei Festlegung auf einen bestimmten Datentyp ist zu spezifizieren, welche Codierung genau verwendet wird.

In den meisten Fällen arbeiten Genetische Algorithmen mit numerischen Werten, die ganze Zahlen repräsentieren. Der hierfür normalerweise benutzte Standard-Binärcode ist allerdings nicht unbedingt die beste Lösung. Der Grund dafür ist, dass im Stan-

dard-Binärcode jedes Bit eine unterschiedliche Wertigkeit besitzt: wird das Bit mit der niedrigsten Wertigkeit verändert, ändert sich der Wert der codierten Zahl nur um 1. Wird dagegen das höchstwertige Bit geändert, hat das je nach Länge des Bitstrings einen ungleich größeren Einfluss auf den Zielwert. Arbeitet man beispielsweise mit Binärstrings der Länge $L = 8$, kann man mit einem String $2^8 = 256$ verschiedene Werte codieren. Bei der Standardcodierung hat das letzte Bit den Wert $2^0 = 1$ und das erste Bit den Wert $2^7 = 128$. Das bedeutet, das Kippen des letzten Bits ändert den Zielwert um 1, beim ersten Bit ändert er sich um 128.

Diese Diskrepanzen möchte man bei Genetischen Algorithmen möglichst vermeiden, daher wird häufig eine Codierung verwendet, die als *Gray-Code* bekannt ist. Die Besonderheit des Gray-Code ist, dass sich aufeinanderfolgende Dezimalzahlen in Gray-Codierung nur in jeweils einem Bit unterscheiden, was beim Standard-Binärcode nicht der Fall ist. Eine Auswahl von Dezimalzahlen mit entsprechender binärer Standardcodierung und Gray-Code-Codierung ist in Tabelle 2.1 zu finden.

dezimal	Standard binär	Gray-Code binär
0	00000000	00000000
1	00000001	00000001
2	00000010	00000011
3	00000011	00000010
4	00000100	00000110
5	00000101	00000111
6	00000110	00000101
7	00000111	00000100
8	00001000	00001100
9	00001001	00001101
10	00001010	00001111
31	00011111	00010000
32	00100000	00110000
63	00111111	00100000
64	01000000	01100000
127	01111111	01000000
128	10000000	11000000
255	11111111	11111111

Tabelle 2.1.: Zahlenwerte in Dezimal-, Standardbinär- und Gray-Codierung

2.3.4. Schematheorie

Der Begriff des Schemas wurde von John Holland (1975) zur formalen Beschreibung der von ihm untersuchten Vorgänge in adaptiven Systemen eingeführt. Hollands Definition des Schemas ist ein Muster, welches eine Menge von Strings definiert, indem es Bits an bestimmten Positionen im String festlegt. Zur Veranschaulichung kann man sich ein Schema als Schablone vorstellen, welche über die einzelnen Strings gelegt und auf Übereinstimmung getestet wird. Die ursprüngliche Definition bezog sich auf einen einfachen, sogenannten *kanonischen* Genetischen Algorithmus, mit binären Individuen bzw. Strings und Ein-Punkt-Crossover, vergleichbar dem in Abschnitt 2.3.1 dargestellten Grundprinzip. Später wurde die Schematheorie von anderen Autoren unter anderem auf nicht-binäre Lösungsrepräsentationen verallgemeinert. Hier wird das ursprüngliche Konzept dargestellt, wie es u.a. in Holland (1986, 1992) zu finden ist.

Die zentrale Aussage der Schematheorie ist, dass sich Schemata mit bestimmten Eigenschaften in evolutionären Prozessen (wie z.B. Genetischen Algorithmen) besser durchsetzen können als andere. Im Zusammenhang der vorliegenden Arbeit ist die Schematheorie relevant, weil sich die Idee der Schemata auf die Klassifikatoren der bauenden Wespenagenten übertragen lässt und die aus ihr folgenden Überlegungen in die Konzipierung des LCS miteinfließen.

Legt man eine binäre Codierung zugrunde, das heißt für jeden String $\vec{a}$ mit Länge L gelte

$$\vec{a} \in (0,1)^L$$

so gilt für ein Schema H analog

$$H \in (0,1,*)^L$$

Ein Schema kann somit für jedes der L Bits entweder festsetzen, dass es 0 oder 1 sein muss, oder aber mittels des *don't-care-Symbols* $*$ anzeigen, dass der Wert des entsprechenden Bits für das Schema nicht relevant ist. Ein String, der einem Schema H entspricht, wird Instanz von H genannt. Hierbei kann jeder beliebige binäre String mit Länge L Instanz von genau 2^L Schemata sein, da für jedes Bit

im String gilt, dass es im korrespondierenden Schema entweder gleich oder egal sein muss.

Man betrachte beispielsweise Strings der Länge $L = 5$ und das Schema $H_1 = *1*10$. Dieses Schema steht für alle Individuen mit Länge $L = 5$, deren zweites Bit eine 1, viertes Bit eine 1 und fünftes Bit eine 0 ist. Dies entspricht der folgenden Menge von Individuen: $\{01010, 01110, 11010, 11110\}$. Diese vier Individuen sind also *Instanzen* des Schemas H_1. Wichtig ist hierbei zu bemerken, dass jedes Individuum Instanz verschiedener Schemata sein kann. So sind alle Instanzen des Schemas H_1 auch gleichzeitig Instanzen des Schemas $H_2 = *1*1*$. Die Menge aller Instanzen von H_1 ist eine echte Teilmenge aller Instanzen von H_2, da H_2 ein Bit weniger definiert als H_1.

In diesem Zusammenhang wird die *Ordnung* $o(H)$ eines Schemas H als Anzahl der in H festgelegten Bits definiert. Beispielsweise ist $o(H_1) = 3$ und $o(H_2) = 2$. Als weitere Eigenschaft wird die *definierende Länge* $\delta(H)$ eines Schemas H als Abstand zwischen erstem und letztem vorgegebenen Bit festgelegt, also $\delta(H_1) = 5 - 2 = 3$ und $\delta(H_2) = 4 - 2 = 2$.

So wie in Abschnitt 2.3.2 die Fitness eines einzelnen Individuums definiert wurde, kann ebenso die Fitness eines Schemas bewertet werden. Es bezeichnet hierbei $I(H)$ die Menge aller möglichen Instanzen von H, $P(t)$ die Population zum Zeitpunkt t und $N(H, t)$ die tatsächliche Anzahl von Instanzen von H in der Population zum Zeitpunkt t.

Die durchschnittliche Fitness aller Instanzen von H zum Zeitpunkt t wird folgendermaßen berechnet:

$$F(H,t) = \frac{\sum F(\vec{a}(t))}{N(H,t)} \quad \text{für alle} \quad \vec{a}(t) \in I(H) \cap P(t) \tag{2.1}$$

Für eine Anzahl von μ Individuen in der Population ergibt sich die durchschnittliche Fitness $\overline{F}(t)$ aller Individuen in $P(t)$:

$$\overline{F}(t) = \frac{1}{\mu} \cdot \sum_{i=1}^{\mu} \vec{a}_i(t) \tag{2.2}$$

Im Verhältnis von $F(H,t)$ zu $\overline{F}(t)$ drückt sich nun implizit die Qualität oder Fitness des Schemas H aus. Unter Vernachlässigung externer Einflussfaktoren wie Mutation und Crossover ergibt sich für die Anzahl der Instanzen von H in der nächsten Generation folgende Differenzengleichung:

$$N(H,t+1) = N(H,t) \cdot \frac{F(H,t)}{\overline{F}(t)} \tag{2.3}$$

Dies bedeutet, dass das Verhältnis von $F(H,t)$ zu $\overline{F}(t)$ darüber bestimmt, ob sich ein Schema H in einer Population durchsetzt oder nicht. Falls die durchschnittliche Fitness der Instanzen eines bestimmten Schemas H über dem Durchschnitt aller Individuen liegt ($F(H,t) > \overline{F}(t)$), so wird die Anzahl der Instanzen dieses Schemas von einer Generation zur nächsten zunehmen und über die Zeit exponentiell wachsen. Gilt dagegen $F(H,t) < \overline{F}(t)$, so kann man eine Verminderung der Instanzen von H in der nächsten Generation sowie eine exponentielle Abnahme im zeitlichen Verlauf erwarten.

Durch Mutation und Crossover wird das Überleben einzelner Individuen und demzufolge bestimmter Schemata zusätzlich beeinflusst. Die Wahrscheinlichkeit, dass ein Schema H eine Mutation bzw. ein Ein-Punkt-Crossover überlebt, kann über die Ordnung $o(H)$ und die definierende Länge $\delta(H)$ berechnet werden.

Wie bereits erwähnt, wird bei der Mutation jedes Bit stochastisch unabhängig mit der Mutationswahrscheinlichkeit p_{mut} invertiert. Die Wahrscheinlichkeit, dass ein einzelnes Bit eines Schemas H einen Mutationsschritt überlebt, ist daher $1 - p_{mut}$, und für das gesamte Schema H mit $o(H)$ Bits ergibt sich folgende Überlebenswahrscheinlichkeit bei der Mutation:

$$p_{\text{mu\_survive}}(H) = (1 - p_{mut})^{o(H)}$$

Für $p_{mut} \ll 1$, also eine sehr geringe Mutationswahrscheinlichkeit, welche in der Praxis meistens gegeben ist, gilt näherungsweise

$$p_{\text{mu\_survive}}(H) = (1 - p_{mut})^{o(H)} \approx 1 - p_{mut} \cdot o(H) \tag{2.4}$$

Was zusätzlich zur Mutation die Weiterverbreitung eines Schemas bedroht, ist das Crossover. Betrachtet man als einfachste Variante das 1-Punkt-Crossover, so wird ein Schema H durch das Crossover genau dann zerstört, wenn ein Crossover-Punkt zwischen erstem und letztem definierendem Bit des Schemas gewählt wird. Auf der anderen Seite muss der Fall berücksichtigt werden, dass beide Eltern Instanzen von H sind, da das Schema dann auch bei einem Austausch bestehen bleibt. Für eine allgemeine Crossover-Wahrscheinlichkeit p_{xover} und eine Population von μ Individuen mit Stringlänge L ergibt sich für H folgende Überlebenswahrscheinlichkeit beim 1-Punkt-Crossover:

$$p_{\text{xo\_survive}}(H) = 1 - p_{xover} \cdot \frac{\delta(H)}{L-1} \cdot (1 - \frac{N(H)}{\mu}) \tag{2.5}$$

In der Praxis ist die Wahrscheinlichkeit für den Fall, dass beide Elternindividuen Instanzen des Schemas H sind, vergleichsweise zu vernachlässigen. Vor allem bei großen Populationen ist die Zahl der Individuen, die ein Schema H repräsentieren, im Vergleich zur Gesamtpopulation sehr gering. Es gilt also $N(H) \ll \mu$ und daraus resultierend $1 - \frac{N(H)}{\mu} \approx 1$. Als vereinfachte Näherung für Gleichung 2.5 wird daher auch folgende Gleichung verwendet:

$$p_{\text{xo\_survive}}(H) = 1 - p_{xover} \cdot \frac{\delta(H)}{L-1} \cdot (1 - \frac{N(H)}{\mu}) \approx 1 - p_{xover} \cdot \frac{\delta(H)}{L-1} \tag{2.6}$$

Aus den Gleichungen 2.3, 2.4 und 2.5 lässt sich die Ungleichung zusammensetzen, die als *Hollands Schematheorem* (Holland, 1992) bekannt wurde:

$$N(H, t+1) \geq N(H,t) \cdot \frac{F(H,t)}{\overline{F}(t)} \cdot [1 - p_{xover} \cdot \frac{\delta(H)}{L-1} \cdot (1 - \frac{N(H)}{\mu})] \cdot (1 - p_{mut})^{o(H)} \tag{2.7}$$

Um die Gleichung 2.7 zu vereinfachen, werden die Näherungsgleichungen 2.4 und 2.6 verwendet, woraus die folgende Version des Schematheorems resultiert:

$$N(H, t+1) \geq N(H,t) \cdot \frac{F(H,t)}{\overline{F}(t)} \cdot [1 - p_{xover} \cdot \frac{\delta(H)}{L-1} - p_{mut} \cdot o(H)] \tag{2.8}$$

Die zentralen Aussagen des Schematheorems sind:

1. Schemata mit überdurchschnittlicher Fitness ($F(H,t) > \overline{F}(t)$) überleben mit höherer Wahrscheinlichkeit.

2. Je kleiner die Ordnung $o(H)$ des Schemas, desto höher die Überlebenswahrscheinlichkeit

3. Je kürzer die definierende Länge $\delta(H)$, desto höher die Überlebenswahrscheinlichkeit

Eine höhere Überlebenswahrscheinlichkeit bedeutet gleichzeitig, dass sich die Instanzen des betreffenden Schemas gegen andere Individuen besser durchsetzen können. Instanzen von Schemata mit überdurchschnittlicher Fitness, niedriger Ordnung und kurzer definierender Länge nehmen im Laufe mehrerer Generationen exponentiell zu.

Goldberg (1989) nannte diejenigen Schemata mit oben angeführten Eigenschaften *building blocks* und formulierte die anschauliche aber inzwischen umstrittene *Building-Block-Hypothese*, welche besagt, dass ein Genetischer Algorithmus seine endgültige Lösung aus einzelnen *building blocks* zusammensetzt:

> „Because highly fit schemata of low defining length and low order play such an important role in the action of genetic algorithms, we have already given them a special name: building blocks. Just as a child creates magnificent fortresses through the arrangement of simple blocks of wood, so does a genetic algorithm seek near optimal performance through the juxtaposition of short, low-order, high-performance schemata, or building blocks.“ (Goldberg, 1989, S.41)

Der von Goldberg hier benutzte Vergleich eines Genetischen Algorithmus mit einem Kind, das komplexe Gebilde aus vielen kleinen Holzklötzen erschafft, veranschaulicht sehr klar auch den zentralen Kritikpunkt an der Building-Block-Hypothese: sie vereinfache die Funktionsweise des GA zu stark. Kritiker sehen den Fehler unter anderem darin, dass vorausgesetzt wird, das vom GA zu lösende Problem ließe sich in voneinander unabhängige Teilprobleme aufspalten, was in der Realität eher die Ausnahme darstellt.

2.4. Learning Classifier Systems

Da die Agenten des implementierten Simulationsmodells nach dem Vorbild der *Learning Classifier Systems* entworfen wurden, wird die Funktionsweise dieser Systeme im Folgenden erläutert. Lernende Klassifizierende Systeme (*LCS*, engl. *Learning Classifier Systems*) gehen wie die Genetischen Algorithmen zurück auf Arbeiten von Holland (1976). LCS wurden als regelbasierte Systeme für Maschinelles Lernen konzipiert, die ihr Verhalten durch Verarbeitung von Rückmeldungen aus ihrer Umgebung optimieren sollen. Interessant sind LCS für die Lösung von Problemen, deren Bewertung nicht sofort nach einer einzelnen Aktion, sondern erst nach mehreren Aktionsschritten erfolgen kann.

Ein einfaches Beispiel für derartige Fragestellungen ist das *Animat-Problem* (Bernauer, 2007), bei dem es darum geht, dass eine Figur sich durch ein Labyrinth bewegt und auf schnellstem Weg zu einem Ziel kommen soll (Abbildung 2.7). Die Figur bekommt eine Belohnung, wenn sie das markierte Ziel erreicht. Da sich die Figur in jedem Zeitschritt aber nur um ein Feld weiterbewegen kann, stellt sich die zentrale Frage, wie die Figur lernen kann, dass vor Erreichen des Ziels und der Belohnung scheinbar sinnlose (weil unbelohnte) Schritte ausgeführt werden müssen. Ein LCS

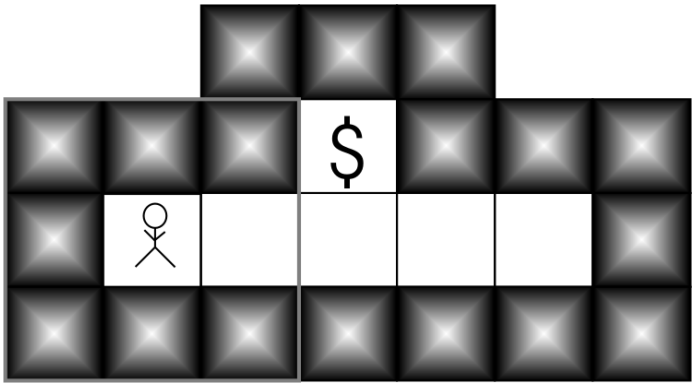


Abbildung 2.7.: Illustration des Animat-Problems (Bernauer, 2007)

besteht im Wesentlichen aus einem beliebig großen Satz von *Klassifikatoren*, welche das Verhalten des LCS steuern. Jeder Klassifikator besteht für sich aus einem *Detektor*-String B, einem *Effektor*-String A und einem numerischen *Stärkewert* S. Für den Begriff *Klassifikator* wird in der Literatur synonym auch der Begriff *Regel* verwendet. Nachrichten aus der Umgebung werden von *Detektoren* angenommen

und mit den Detektor-Strings der einzelnen Klassifikatoren verglichen (siehe Abbildung 2.8). Die Codierung und Länge der Nachrichten muss bekannt sein und mit den Detektor-Strings übereinstimmen. Im einfachsten Fall wird eine binäre Codierung mit fester Stringlänge verwendet. In LCS ist jede Regel mit einem numerischen Wert

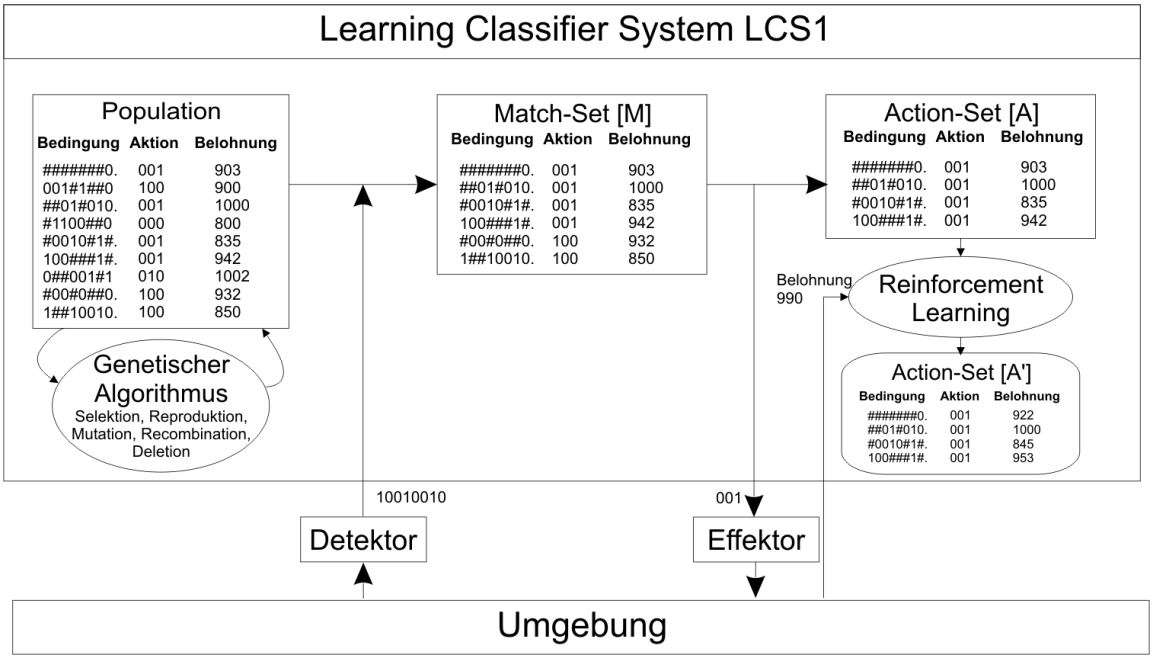


Abbildung 2.8.: Schematische Darstellung eines *Learning Classifier System* (Bernauer, 2007)

verknüpft, der ihre *Stärke* oder auch *Fitness* definiert. Um Verwechslungen mit der globalen Fitness der Wabenstrukturen zu vermeiden soll im Folgenden der Begriff *Stärke* verwendet werden. Die Regeln werden im Kontext der LCS *Klassifikatoren* genannt. Die *Stärke* einer Regel beeinflusst wesentlich die Wahrscheinlichkeit für ihre Anwendung. In Hollands Modell wird jeder erfüllten Regel ein bestimmter Wert zugeordnet, der *bid* (dt. *Gebot*) genannt wird. Holland vergleicht die Auswahl einer Regel mit einer Auktion, in der jede passende Regel ein Gebot abgibt und die Höhe des Gebots die Wahrscheinlichkeit bestimmt, mit der die Regel tatsächlich Anwendung findet. Außer von der Stärke $S(R)$ der Regel R ist $bid(R)$ noch abhängig von einem Wert, den Holland *Spezifizität* $sp(R)$ nennt. Damit ist die Menge der in der Regel spezifizierten Information gemeint.

Das Gebot einer Regel R mit Stärke $S(R)$ berechnet sich wie folgt:

$$bid(R) = S(R) \cdot \log_2[sp(R)] \tag{2.9}$$

Die Spezifizität wird logarithmisch in die Formel mit einberechnet, um ihren Einfluss gegenüber der Stärke zu reduzieren.

Sind in einem Zeitschritt mehrere Regeln gleichzeitig gültig, so wird anhand ihrer entsprechenden *bid*-Werte entschieden, welche Regel zur Anwendung kommt. Wird ein Input I als *passend* zu einer Regel R klassifiziert, wird R einer Liste von passenden Regeln (engl. *match set*) $\mathcal{M}_t$ hinzugefügt. Nach einem stochastischen Verfahren wird aus allen Regeln in $\mathcal{M}_t$ diejenige ausgewählt, die ausgeführt werden soll, wobei der Stärkewert S der Klassifikatoren in diese Auswahl miteinfließt. Der Effektor-String A der ausgewählten Regel bestimmt die von den Effektoren ausgeführte Aktion des LCS. Analog zu den Detektoren, die die „Sinnesorgane“ des LCS repräsentieren, bezeichnen Effektoren diejenigen Komponenten des LCS, die aktiv mit der Umwelt kommunizieren bzw. diese manipulieren.

Von der Umgebung bekommt das LCS Rückmeldungen darüber, wie das Verhalten des LCS zu bewerten ist. Eine positive Rückmeldung wird als *Belohnung* und eine negative Rückmeldung als *Bestrafung* interpretiert. Allerdings kommen diese Rückmeldungen im Allgemeinen nicht nach jeder einzelnen Aktion, so dass das LCS dafür sorgen muss, die positive bzw. negative Rückmeldung an die verantwortlichen Regeln zurückzugeben. Ziel dieses Verfahrens ist es, dass sich das System mit der Zeit an das erwünschte Verhalten anpasst, da der Stärkewert von Regeln mit positiver Rückmeldung erhöht und von solchen mit negativer Rückmeldung abgeschwächt wird. Positiv bewertete Regeln werden in Zukunft mit höherer Wahrscheinlichkeit ausgewählt werden als negativ bewertete.

Im Folgenden werden die Bestandteile eines einfachen Learning Classifier Systems formal eingeführt. Die Darstellung geht von einer binären Codierung aus und orientiert sich an Weicker (2002). Eine schematische Darstellung der Funktionsweise ist in Abbildung 2.8 zu sehen.

Ein Klassifikator R besteht aus einer Bedingung (Detektor-String) B der Länge l, einer Ausgabe (Effektor-String) A der Länge m und einem numerischen Stärkewert S. Jeder binäre Input-String I hat wie der Detektor-String die Länge l.

Für jede Bedingung (Detektor-String) B gilt:

$$B \in \mathcal{B} = \{0, 1, *\}^l \tag{2.10}$$

Für jede Ausgabe (Effektor-String) A gilt:

$$A \in \mathcal{A} = \{0,1\}^m \tag{2.11}$$

Für jeden Stärkewert S gilt:

$$S \in \mathbb{R} \tag{2.12}$$

Die Stärkewerte S werden bei Initialisierung des Systems auf bestimmte Werte gesetzt, was in der implementierten Version mit Zufallszahlen geschieht.

Jeder Klassifikator R besteht also aus einer Bedingung B, einer Ausgabe A und einem Fitnesswert S:

$$R = (B, A, S) \in \mathcal{B} \times \mathcal{A} \times \mathbb{R} \tag{2.13}$$

Die Gesamtheit aller Klassifikatoren eines LCS wird Regelsatz $\mathcal{R}$ genannt:

$$\mathcal{R} \subset \mathcal{B} \times \mathcal{A} \times \mathbb{R} \tag{2.14}$$

Ein Klassifikator R heißt genau dann *aktiviert* oder *passend* zum Zeitpunkt t, wenn die Eingabe I zum Zeitpunkt t mit der Bedingung des Klassifikators übereinstimmt. In diesem Zusammenhang bedeutet Übereinstimmung, dass jedes Zeichen des Input-Strings I dem entsprechenden Zeichen im Detektor-String B entspricht, außer wenn bei B an der Stelle ein don't-care-Symbol $*$steht, was bedeutet dass die Zeichen nicht übereinstimmen müssen. Formal geschrieben heißt das mit $I \in \mathcal{I} = \{0,1\}^l$, dass ein Klassifikator R zum Zeitpunkt t mit Input I aktiviert ist, wenn folgendes Prädikat wahr ist:

$$\bigwedge_{i=1}^{l} (B_i \neq * \Rightarrow B_i = I_i) \tag{2.15}$$

B_i bezeichnet das Zeichen an Stelle i im Detektor-Sring B.

Alle zum Zeitpunkt t aktivierten Klassifikatoren im Regelsatz $\mathcal{R}$ werden zum *match set* $\mathcal{M}_t$ hinzugefügt. Werden verschiedene Ausgaben von den Klassifikatoren in $\mathcal{M}_t$ vorgeschlagen, so wird durch ein stochastisches Verfahren für jede Ausgabe A eine Auswahlwahrscheinlichkeit berechnet. Diese berücksichtigt sowohl den Stärkewert S

als auch die Spezifizität sp aller Klassifikatoren R in $\mathcal{M}_t$, die A vorschlagen. sp ist die Anzahl der Zeichen in B, die für die entsprechende Stelle in I ein spezifizisches Zeichen verlangen. Grundsätzlich sind das alle Zeichen, die kein *don't-care-Symbol* $*$ sind. Es gilt z.B. $sp(**0011**) = 4$ und $sp(101**010) = 6$. Je höher Spezifizität und Fitness von R sind, desto höher ist die Wahrscheinlichkeit dafür, dass R ausgewählt wird.

Im Modell von Holland (1986) gibt jede Regel R in $\mathcal{M}_t$ ein Gebot (engl. *bid*) ab, welches aus $sp(R)$ und dem momentanen Stärkewert $S(R,t)$ berechnet wird. Außerdem wird eine systemabhängige Konstante β einberechnet, womit für das Gebot für eine Regel R zum Zeitpunkt t gilt:

$$bid(R,t) = \beta \cdot sp(R) \cdot S(R,t) \tag{2.16}$$

Die Aktion A der so ausgewählten Regel wird ausgeführt, und das System wartet auf den nächsten Input I zum Zeitpunkt $t+1$.

Learning Classifer Systeme kombinieren zwei grundlegende Mechanismen für Maschinelles Lernen (Bull und Kovacs, 2005). Beide Mechanismen adressieren jeweils ein grundlegendes Teilproblem, das in Zusammenhang mit LCS auftritt. Das erste Teilproblem wird *Credit Assignment* oder „apportionment of credit problem“ (Geyer-Schulz, 1995) genannt und dreht sich um die Frage, wie das System entscheiden soll, welche der zu einem Zeitpunkt t aktivierten Regeln zur Erreichung eines Ziels zu einem zukünftigen Zeitpunkt $t+n$ beigetragen haben und wie das positive bzw. negative Feedback auf diese Regeln verteilt werden soll (siehe Abschnitt 2.4.1). Das zweite Teilproblem ist das Finden neuer Klassifikatoren (engl. *Rule Discovery*), wofür in LCS ein Genetischer Algorithmus verwendet wird. Die Grundprinzipen Genetischer Algorithmen wurden bereits in Abschnitt 2.3.1 erläutert, wobei die Anwendung im Kontext der LCS in Abschnitt 2.4.2 beschrieben wird.

2.4.1. Credit Assignment: Reinforcement Learning

In einem Learning Classifier System werden die Fitnesswerte der einzelnen Klassifikatoren durch Rückmeldung aus der Umgebung schrittweise angepasst, was *Credit Assignment* genannt wird. Das zugrundeliegende Verfahren heißt *Reinforcement*

Learning (RL) (Sutton und Barto, 1998) und basiert darauf, dass Klassifikatoren mit positivem Effekt auf die Umgebung verstärkt und solche mit negativem Effekt abgeschwächt werden sollen. Eine mögliche Definition von Reinforcement Learning lautet „ [...] learning from interaction with an environment, from the consequences of action, rather than from explicit teaching.“ (Sutton, 2004) Die Herausforderung für das System besteht darin, die Rückmeldung der Umgebung an alle beteiligten Klassifikatoren zu verteilen, da normalerweise nicht nach einem einzelnen Zeitschritt beurteilt werden kann, ob der Gesamteffekt auf die Umwelt positiv oder negativ ist. Sehr anschaulich wird dies bei Betrachtung eines Brettspiels wie Schach, wo oft erst nach sehr vielen Schritten beurteilt werden kann, ob ein Zug zu Beginn des Spiels gut oder schlecht war.

Die im ursprünglichen Modell von Holland (1975) eingesetzte Methode für das *Credit Assignment* wurde *Bucket-Brigade-Algorithmus* (BBA) genannt. Das System merkt sich in jedem Zeitschritt t alle passenden nach $\mathcal{M}_t$ übertragenen Regeln. Im nächsten Zeitschritt $t+1$ wird das Gebot jeder ausgewählten Regel in einem sogenannten *bucket* abgespeichert. Der Inhalt dieses *bucket* wird dann zu gleichen Teilen unter den in Zeitschritt t aktivierten Regeln aufgeteilt (Bull und Kovacs, 2005). Das heißt, der Stärkewert $S(R)$ wird mit dem entsprechenden Anteil des Inhalts des *bucket* verrechnet. Bekommt das System eine Rückkopplung aus der Umgebung, so wird diese Rückkopplung ihrerseits mit dem Stärkewert der letzten aktivierten Regel verrechnet.

Holland modellierte den Bucket-Brigade-Algorithmus bewusst in Analogie zu einem Wirtschaftskreislauf. Das System kann als Handelskette gesehen werden, in dem jede Regel als Geschäftspartner mit ihrer Stärke als Kapital agiert. In den meisten neueren LCS-Varianten wie XCS (Wilson, 1995) wird ein RL-Algorithmus namens *Q Learning* eingesetzt, der erstmals von Watkins (1989) beschrieben wurde.

2.4.2. Rule Discovery: Genetische Algorithmen

Das Finden neuer Klassifikatoren geschieht in Learning Classifier Systemen mithilfe Genetischer Algorithmen (GA, siehe Abschnitt 2.3). Es gibt verschiedene Möglichkeiten, den Genetischen Algorithmus im LCS einzusetzen, wobei grundsätzlich zwei Varianten der LCS unterschieden werden: *Pittsburgh*- und *Michigan*-LCS (Sigaud und Wilson, 2007). Pittsburgh-LCS, so genannt, weil der Ansatz von Smith (1980) an der

University of Pittsburgh entwickelt wurde, setzen, wie oben beschrieben, einen GA auf einer Population von Regelsätzen ein. Im Gegensatz dazu betrachten Michigan-LCS, entwickelt von John Holland an der University of Michigan, den einzelnen Klassifikator als Individuum. Es wird also nur ein einziger Regelsatz betrachtet, der für den GA eine Population von Klassifikatoren darstellt, und der durch Reinforcement Learning und den Genetischen Algorithmus optimiert werden soll. Im Folgenden wird die Pittsburgh-Variante für den Einsatz Genetischer Algorithmen in LCS beschrieben, da diese im Rahmen der vorliegenden Arbeit für die Implementation verwendet wurde.

In Pittsburgh-LCS operiert ein Genetischer Algorithmus auf einer Population von Regelsätzen. Es werden mehrere Simulationsläufe mit verschiedenen Regelsätzen durchgeführt und die Ergebnisse der Simulationen jeweils von einer Fitnessfunktion F bewertet. Das Ergebnis dieser Bewertung wird *Globaler Fitnesswert* oder englisch *Overall Fitness* genannt. Zu jedem verwendeten Regelsatz wird der erzielte globale Fitnesswert gespeichert.

Der GA wählt nun nach einem stochastischen Selektionsverfahren Regelsätze aus und erzeugt durch Rekombination und Mutation (siehe Abschnitt 2.3) neue Regelsätze, die dann im nächsten Durchlauf getestet werden und im Idealfall bessere Ergebnisse erzielen. Als Selektionsverfahren wird hier die *Roulette-Wheel-Selektion* (siehe Abschnitt 2.3.2) verwendet, bei der die Wahrscheinlichkeit für die Auswahl eines Individuums, also eines Regelsatzes $\mathcal{R}$, von seinem globalen Fitnesswert $F(\mathcal{R})$ abhängt, also von der Rückmeldung der Umgebung.

2.5. Komplexität der Problemstellung

Die in der vorliegenden Arbeit betrachtete Fragestellung ist formal gesehen ein Optimierungsproblem. Die zentrale Frage lautet, wie man eine Struktur im zweidimensionalen Raum so erzeugen kann, dass eine festgelegte Fitnessfunktion maximiert wird. Die Gesamtheit aller möglichen Zustände des Wabengitters stellt den Zustandsraum des Problems dar. Als Zustand eines Punktes wurde die Art der dort gebauten Wabenzelle definiert. Bei einem Gitternetz mit W Punkten und n möglichen Zuständen eines einzelnen Punktes ergibt sich ein Zustandsraum der Größe n^W, was schon im

einfachsten Fall von nur einem Wabentyp zu einem exponentiell zu W wachsenden Zustandsraum führt. Jeder Punkt hat in diesem Fall zwei mögliche Zustände - bebaut oder nicht bebaut - daher gilt $n = 2$ und für die Größe des Zustandsraums 2^W.

Da jede Fitnessfunktion, die die Qualität einer entstandenen Wabenstruktur beurteilen soll, von allen Zellen des Gitters abhängt, wächst daher auch die Berechnungszeit für die optimale Lösung der Fitnessfunktion exponentiell zu W, was die Fragestellung zu einem *NP-vollständigen* Problem macht. NP-vollständig bedeutet, dass ein solches Problem nicht in polynomieller Zeit lösbar ist, weil der Zustandsraum des Problems schneller wächst als polynomiell (in diesem Beispiel etwa exponentiell). NP-vollständige Probleme gelten in der Theoretischen Informatik als *nicht effizient lösbar*.

Eine Möglichkeit, dennoch akzeptable Lösungen für derartige Probleme zu finden, stellen heuristische Optimierungsverfahren dar. Diese werden dadurch charakterisiert, dass die gefundenen Lösungen nicht unbedingt ein globales Maximum der Zielfunktion darstellen, aber für viele Probleme trotzdem näherungsweise optimale oder gute Lösungen erzielen. Zur Analyse des Problems wurde ein Simulationsmodell entwickelt, das verschiedene heuristische Verfahren wie *Genetische Algorithmen* und *Learning Classifier Systems* einsetzt, um eine möglichst optimale Lösung zu finden. Der Einsatz hierarchischer Ebenen von Klassifikatoren verspricht eine Möglichkeit, komplexe Strukturen zu erzeugen, deren Fitness von mehreren Parametern abhängt. Je nach Zustand der Umwelt kann von einer zur anderen hierarchischen Ebene umgeschaltet werden, wodurch komplexere Wabenstrukturen möglich werden.

3. Methode

Um eine Antwort auf die in Abschnitt 2.5 erläuterte Frage zu erhalten, wurde im Rahmen dieser Arbeit eine Simulationsumgebung entwickelt. Ziel der damit durchgeführten Simulationsläufe war es zu klären, ob die Vorgänge beim Wabenbau von Insekten auf ein Modell übertragbar und inwieweit die Ergebnisse eines solchen Systems mit realen Insektennestern vergleichbar sind. Aufgrund der Größe des Zustandsraumes der Problemstellung ist eine direkte Berechnung der optimalen Lösung nicht praktikabel (siehe Abschnitt 2.5), weswegen eine Simulationsumgebung auf Basis Lernender Klassifizierender Systeme entwickelt wurde. Zur Veranschaulichung der Ergebnisse wurde außerdem eine graphische Darstellung der durch die Simulation entstandenen Wabenstrukturen realisiert.

Da Begriffe wie *Population* und *Individuum* sowohl im Kontext staatenbildender Insekten als auch in der Nomenklatur der Genetischen Algorithmen eine zentrale Bedeutung besitzen, ist es wichtig, klar voneinander abzugrenzen, was die Bedeutung welchen Begriffes im jeweiligen Zusammenhang ist. Ein Insektenstaat - natürlich wie simuliert - besteht aus einer Menge von Insekten. Das einzelne Insekt wird als *Individuum*, die Gesamtheit aller Individuen als *Population* bezeichnet. Im Kontext des Simulationsmodells wird das simulierte Insekt vorwiegend als *Agent* bezeichnet, um Verwechslungen zu vermeiden. Ein Genetischer Algorithmus arbeitet ebenfalls mit einer Population von Individuen, wobei die interne Repräsentation eines Individuums *Genotyp* und die von ihm vorgeschlagene Problemlösung *Phänotyp* genannt werden.

In diesem Kapitel werden die Komponenten des implementierten Modells im Einzelnen beschrieben. Sofern Bezug auf den in Anhang A beigefügten MATLAB-Quellcode genommen wird, bezeichnen in `Schreibmaschinenschrift` geschriebene Wörter die entsprechenden Variablen- oder Funktionsnamen. `<dateiname>.m` bezeichnet die entsprechende Datei.

3.1. Benutzte Werkzeuge

Die Simulationsumgebung wurde in MATLAB® 7.7.0.471 (R2008b) implementiert und die Simulationen auf einem PC mit 2 Gigabyte RAM und Ubuntu mit Linux-Kernel 2.6.28 als Betriebssystem durchgeführt. MATLAB ist eine proprietäre Skriptsprache mit Möglichkeit zu objektorientierter Programmierung und umfangreicher graphischer Darstellung.

3.2. Konzept des Simulationsmodells

Zur Simulation des Wabenbaus wurde ein *zeitdiskretes* Simulationsmodell implementiert. Zeitdiskrete Simulationsmodelle zeichnen sich dadurch aus, dass Änderungen des Systemzustands nur zu bestimmten, diskreten Zeitpunkten betrachtet werden und der Systemzustand zwischen den diskreten Zeitpunkten für die Simulation als konstant angenommen wird (Page et al., 2000). Konkret bedeutet das in diesem Modell, dass jeder Agent pro Zeitschritt einen neuen Baustein bauen und sich danach um eine festgelegte Schrittweite auf der Wabe in eine zufällig gewählte Richtung bewegen kann. Im nächsten Zeitschritt ist der Baustein gebaut und der Agent befindet sich auf seiner neuen Position, die verstrichene Zeit wird nicht direkt berücksichtigt. Hierin unterscheiden sich diskrete Simulationsmodelle von *kontinuierlichen* Modellen, bei denen sich der Systemzustand stetig über die Zeit ändert.

3.3. Programmierung der Simulationsumgebung

Das Skript `simulationsumgebung.m` stellt das Rahmenwerk für den gesamten Simulationsablauf zur Verfügung. Hier wird der Simulationsraum und die Bauagenten als Modelle der nestbauenden Insekten erzeugt. Auch die global geltenden Parameter werden hier eingestellt. Über den zweidimensionalen kontinuierlichen Raum wird eine Struktur aus regelmäßigen Sechsecken gelegt, welche den Raum in gleich große Zellen aufteilt. Die Größe des Raumes ist frei wählbar und wird vom Sechseckgitter maximal ausgenutzt.

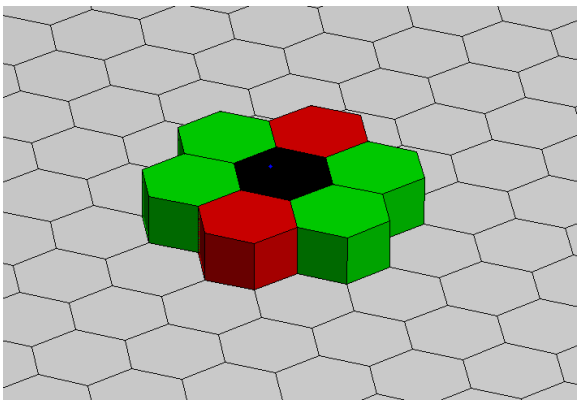


Abbildung 3.1.: Eine begonnene Simulation nach sechs Bauschritten. Der schwarze Baustein markiert die Ausgangswabe, die roten und grünen Steine markieren die gebauten Zellen von Typ 1 (rot) und Typ 2 (grün). Der kleine blaue Punkt auf der schwarzen Zelle markiert die Position des Agenten.

Auf jeder sechseckigen Zelle des Gitters können verschiedene Arten von Wabenbausteinen gebaut werden, die in der grafischen Darstellung mit verschiedenen Farben markiert werden (siehe Abschnitt 3.7). Vor Beginn der Simulation werden sieben Zellen in der Mitte des Gitters als Ausgangspunkt mit Bausteinen vom Typ 1 bebaut, was als Analogie zum Aufhängepunkt eines realen Wespennestes (siehe Abschnitt 2.1) verstanden werden kann. Zur Unterscheidung von den anderen Bausteinen werden die initial gebauten Zellen schwarz markiert (siehe Abbildung 3.2). Bei Ausführung

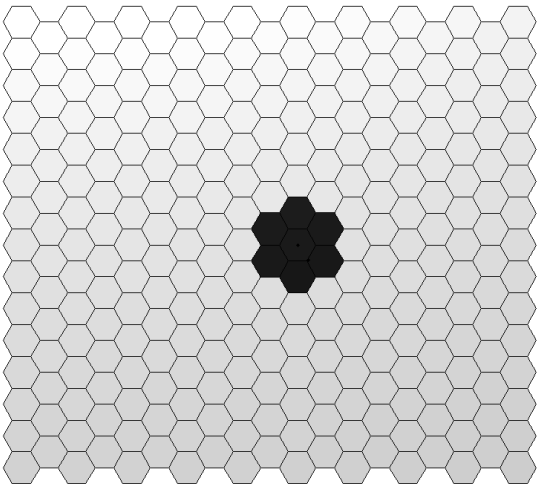


Abbildung 3.2.: Initialzustand der Simulation mit bebauten Anfangszellen

des Skripts wird eine von den angegebenen Parametern abhängige Anzahl von Simulationsläufen gestartet.

3.3.1. Regelsatz und Parameter

Der genaue Ablauf einer Simulation ist abhängig von mehreren variablen Parametern sowie dem Regelsatz, nach dem die Agenten ihre Umgebung klassifizieren. Im Folgenden sind alle Parameter der Simulationsumgebung aufgeführt:

- Größe des Simulationsraumes in X- (`DX`) und Y-Richtung (`DY`)
- Anzahl der gleichzeitig aktiven Agenten (`wespenanzahl`)
- Maximale Anzahl der Schritte pro Agent und Simulationsdurchgang (`MaxSteps`)
- Größe der Population des Genetischen Algorithmus - entspricht der Anzahl der Simulationen, die mit der Population pro Durchlauf des GA durchgeführt werden (`numberofruns`)
- Anzahl der Anwendungen des Genetischen Algorithmus (`numberofgaruns`)
- In jedem Simulationslauf arbeitet der einzelne Agent mit dem gleichen statischen Regelsatz von 31 Regeln (siehe Tabelle 3.2), der im Array `RuleSet1` abgelegt ist. Eine detaillierte Erklärung der Regelcodierung findet sich in Abschnitt 3.4.2.

3.4. Programmierung der Agentenobjekte

Die Insekten, die im Rahmen der Simulation an der Wabenstruktur bauen, wurden als autonome *Agenten* modelliert, die sich unabhängig voneinander frei im zweidimensionalen Raum bewegen können. Hierzu wurden die Möglichkeiten von MATLAB zur objektorientierten Programmierung genutzt und in der Datei `bauagent.m` (Listing in Anhang A.2) eine eigene Objektklasse zur Beschreibung der Agenten programmiert.

Die Position der Agentenobjekte im Raum wird als kontinuierlicher Wert angegeben und gespeichert, das heißt jeder Agent besitzt zu jedem Zeitpunkt einen absoluten zweidimensionalen Positionsvektor bestehend aus X- und Y-Koordinate. Die Position wurde derart modelliert, um möglichst nah am biologischen Vorbild zu bleiben. Da für die Regelverarbeitung des Agenten entscheidend ist, auf welcher Zelle des Gitternetzes er sich befindet, wird die Nummer der entsprechenden Zelle in jedem Zeitschritt berechnet.

Die Funktionsweise der Agenten entspricht in den Grundzügen der eines Learning Classifier Systems (siehe Abschnitt 2.4). Jeder Agent besitzt einen Satz von Klassifikatoren, mit deren Hilfe die lokale Umgebung *klassifiziert* und über eine entsprechende Aktion entschieden wird. Die Agenten können nicht miteinander kommunizieren, haben keine Wahrnehmung der globalen Wabenstruktur und auch keinen Bauplan, wie die fertige Wabe auszusehen hat. Die einzige Wahrnehmungsfähigkeit ist die Erkennung ihrer direkten Umgebung, wie sie in Abschnitt 3.4.2 detailliert beschrieben ist.

Zu jedem diskreten Zeitpunkt t wird für jeden Agenten die Funktion `movebuild` aufgerufen, die die Aktivität des Agenten steuert. Diese Aktivität setzt sich aus zwei Einzelschritten zusammen. Zuerst erfolgt die Klassifikation der Umgebung mit eventueller Manipulation des Wabengitters, danach die Bewegung des Agenten auf eine neue Position im Raum.

3.4.1. Implementation der Klassifikatoren

In der Datei `classifier.m` (Listing in Anhang A.3) wird die interne Struktur der Klassifikatoren festgelegt. Dazu wird wie für die Agenten eine eigene Objektklasse `classifier` definiert. Die Klasse enthält außer einer Konstruktor-Funktion zur Erzeugung der `classifier`-Objekte alle Eigenschaften zur Beschreibung des Klassifikators, die in objektabhängigen Variablen gespeichert werden.

In der folgenden Liste sind alle Eigenschaften der `classifier`-Klasse aufgeführt:

- `detector` enthält den Detektor-String als Zeichenkette (siehe Abschnitt 3.4.2).

- `effector` enthält die bei Aktivierung des Klassifikators auszuführende Aktion als numerischen Wert. `0` steht hier für *nicht bauen* und die Zahlen `1,2,3` für den entsprechenden Bausteintyp.

- `strength` ist eine ganze Zahl, die die *Stärke* des Klassifikators angibt. Dieser Wert wird bei Erzeugung des `bauagent`-Objektes für alle Klassifikatoren stochastisch unabhängig voneinander per Zufall auf einen Wert zwischen 0 und 255 gesetzt.

- `fitnessstring` enthält den *Stärke*-Wert aus `strength` als 8-bit-Gray-Code (siehe Abschnitt 2.3.3) in Form einer binären achtstelligen Zeichenkette. Ein `strength`-Wert von 11 wäre in `fitnessstring` z.B. als '00001111' gespeichert. Diese Form der Codierung ist wichtig für die Funktion des Genetischen Algorithmus (siehe Abschnitt 3.6).

- `specificity` bezeichnet die Spezifizität $sp(R)$ des Klassifikators. Das entspricht der Zahl der Stellen im Detektor-String, die ungleich dem *don't-care-Symbol* `*`) sind.

- `bid` ist der aus `strength` und `specificity` für alle passenden Klassifikatoren in einem Zeitschritt berechnete Wert, der zur Entscheidung über die Aktivierung eines Klassifikators verwendet wird. Die genaue Beschreibung des Verfahrens ist in Abschnitt 3.4.2 zu finden.

3.4.2. Klassifikation und Manipulation der Umgebung

Die Klassifikation der Umgebung erfolgt durch eine Verknüpfung von Umgebungsmustern mit entsprechenden Aktionen. Im String-Array `RuleSet1` werden die Muster und die damit verknüpfte Aktion abgespeichert. Die ersten sechs Zeichen in einer solchen Zeichenkette repräsentieren die direkte Umgebung der potentiellen Bauposition und werden wie bei LCS-Systemen *Detektor-String* genannt. Das siebte Zeichen steht für die Aktion, die der Agent bei Aktivierung des Klassifikators ausführt, und wird als *Effektor-String* bezeichnet.

Da der Agent sich nur auf bereits bebauten Zellen befinden darf, muss zunächst eine leere Zelle in unmittelbarer Umgebung ausgewählt werden, auf der ein neuer

Baustein gebaut werden könnte. Dazu werden die direkt angrenzenden Zellen - bei der senkrecht oberhalb angrenzenden Zelle angefangen - im Uhrzeigersinn getestet, ob sie noch unbebaut sind. Findet der Agent eine unbebaute Zelle, ist dies eine potentielle Bauposition.

Im nun folgenden eigentlichen Klassifizierungsvorgang wird der Zustand der die potentielle Bauposition direkt umgebenden Zellen - ebenfalls bei der oberen Zelle angefangen im Uhrzeigersinn - nacheinander abgefragt. Die Zustände der umgebenden Zellen bilden aneinandergereiht die *Input-String* I genannte Eingabe in das System. Eine `0` in I steht dabei für eine unbebaute Zelle, `1`, `2` oder `3` steht für eine mit Bausteintyp `1`, `2` bzw. `3` bebaute Zelle. I wird dann nacheinander mit allen im Regelsatz vorhandenen Klassifikatoren verglichen. Für jeden Klassifikator werden die einzelnen Positionen in I der Reihe nach mit den einzelnen Positionen des Detektor-String B verglichen. Die getestete Regel gilt als *passend*, wenn es keine Widersprüche zwischen I und B gibt. Die Symbole `0`, `1`, `2` und `3` haben in B die gleiche Bedeutung wie in I. Zusätzlich sind für B noch die Symbole `B` und `*` definiert. `B` steht für eine Zelle, die mit einem *beliebigen Baustein* besetzt ist, und `*` wird *don't-care-Symbol* genannt, das heißt dass die entsprechende Position für die Regel nicht relevant ist.

Für $I =$ `110011` würde z.B. sowohl $B_1 =$ `110011` wie auch $B_2 =$ `11**11` oder $B_3 =$ `BB**11` als passend erkannt werden. Eine graphische Übersicht über die Regelsyntax ist in Tabelle 3.1 zu finden.

Zeichen	Bedeutung	Graphische Darstellung
`0`	leeres Feld	
`1`	Baustein von Typ 1	
`2`	Baustein von Typ 2	
`3`	Baustein von Typ 3	
`B`	beliebiger Baustein	
`*`	*don't-care* - Zustand egal	

Tabelle 3.1.: Vom Regelwerk verwendete Zeichen und ihre Bedeutung

Wie in Abschnitt 2.4 bereits erläutert werden alle zu einem Zeitpunkt t als passend erkannte Klassifikatoren in einer Menge $\mathcal{M}_t$ zuammengefasst. Aus $\mathcal{M}_t$ wird nun ein

Klassifikator ausgewählt. Dazu wird für jeden Klassifikator R ein entsprechendes Gebot $bid(R)$ berechnet. Es gilt

$$bid(R) = S(R) \cdot \log_2(sp(R))$$

$bid(R)$ berechnet sich also aus Stärke und Spezifizität des Klassifikators.

Mittels *Roulette-Wheel-Selektion* (siehe Abschnitt 2.3.2) wird nun anhand der *bid*-Werte einer der Klassifikatoren aus $\mathcal{M}_t$ ausgewählt. Die Auswahlwahrscheinlichkeit ist hierbei direkt proportional zum *bid*-Wert des Klassifikators.

Wurde ein Klassifikator R aus $\mathcal{M}_t$ ausgewählt, so wird die in `effector` als Zahlenwert gespeicherte Aktion ausgeführt. Mögliche Aktionen sind in diesem Modell das Bauen verschiedener Typen von Zellen (`effector` = 1, 2 oder 3) oder das explizite Nicht-Bauen einer Zelle (`effector` = 0). Die verschiedenen Bausteintypen haben keine konkrete Bedeutung, sondern sollen auf das Modell abbilden, dass Wespen und Bienen die Wabenzellen ihrer Nester für unterschiedliche Aufgaben nutzen. Es könnten z.B. Zellen von Typ 1 für Zellen stehen, in denen die Brut gepflegt wird, Zellen von Typ 2 für die Lagerung von Nahrung und Zellen von Typ 3 für eine beliebige andere Nutzung.

Der komplette verwendete Regelsatz `RuleSet1` ist in Tabelle 3.2 zu sehen. Die graphische Darstellung ist folgendermaßen zu interpretieren, dass die mittlere sechseckige Zelle die zu bebauende Position darstellt und die umgebenden sechs Zellen den für den Klassifikator relevanten Input I repräsentieren. Die farbige Markierung entspricht der in Tabelle 3.1 angegebenen.

3.4.3. Erweiterung durch hierarchische Klassifikatoren

Zur Implementierung einer zweiten hierarchischen Ebene von Klassifikatoren wurde das ursprüngliche Modell um einen zweiten Regelsatz erweitert, der analog zum ersten Regelsatz im Array `RuleSet2` abgelegt ist. Es wurde eine neue globale Variable `rulesetanzahl` in `simulationsumgebung.m` angelegt, die die Zahl der hierarchisch hintereinandergeschalteten Regelsätze angibt. Gültige Werte für `rulesetanzahl` sind 1 und 2. Setzt man die Variable auf 1, verhält sich das Modell so, als existiere der zweite Regelsatz nicht. Setzt man sie auf 2, wird nach Ausführung der Simulationen

Index	*B*	*A*	Grafik
1	BBBBBB	1	
2	BBBBB0	1	
3	0BBBBB	1	
4	B0BBBB	1	
5	BB0BBB	1	
6	BBB0BB	1	
7	BBBB0B	1	
8	BBBB00	1	
9	0BBBB0	1	
10	00BBBB	1	
11	B00BBB	1	
12	BB00BB	1	
13	BBB00B	1	
14	BBB000	1	
15	0BBB00	1	
16	00BBB0	1	

Index	*B*	*A*	Grafik
17	000BBB	1	
18	B000BB	1	
19	BB000B	1	
20	BB0000	1	
21	0BB000	1	
22	00BB00	1	
23	000BB0	1	
24	0000BB	1	
25	B0000B	1	
26	B00000	0	
27	0B0000	0	
28	00B000	0	
29	000B00	0	
30	0000B0	0	
31	00000B	0	

Tabelle 3.2.: Die 31 Klassifikatoren aller Simulationen in Abschnitt 4.1 mit Indexnummer, Detektor-String B, Effektor-String A und graphischer Darstellung

mit Regelsatz 1 derjenige Regelsatz mit der höchsten erreichten Fitness ausgewählt und in `bestclassifierset1` zwischengespeichert. Die neu gestartete Simulation arbeitet zunächst nach wie vor mit Regelsatz `RuleSet1`, jedoch nur so lange bis die maximale Schrittzahl `MaxSteps` erreicht ist. An diesem Punkt wird auf den zweiten Regelsatz `RuleSet2` umgeschaltet. Alle vorhandenen Klassifikatoren der Agenten werden gelöscht und durch die neuen Regeln aus `RuleSet2` ersetzt. Die Initialisierung der Stärkewerte erfolgt wie beim ersten Regelsatz zufällig. Danach läuft die Simulation auf der bereits entstandenen Struktur mit dem neuen Regelsatz weiter, bis die maximale Schrittzahl `MaxSteps2` erreicht ist.

Der komplette Regelsatz aus `RuleSet2` ist in den Tabelle 3.3 und 3.4 aufgelistet.

Ergebnisse der Simulationen mit zwei hierarchisch geordneten Sätzen von Klassifikatoren sind in Abschnitt 4.3 zu finden.

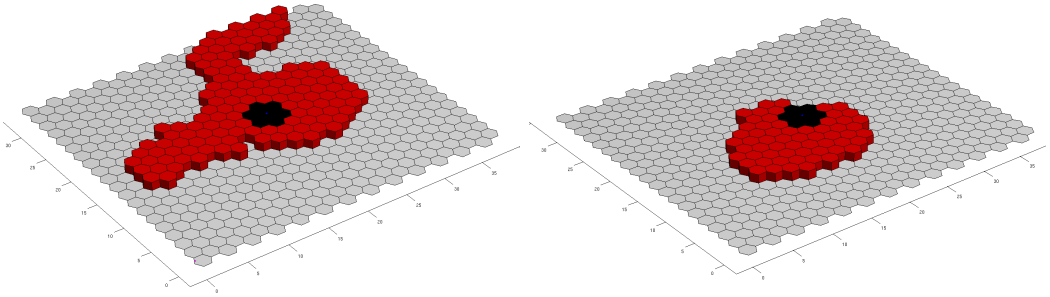
3.4.4. Bewegung der Agenten

Die Ausgangsposition aller Agenten befindet sich auf der mittleren der zu Beginn bebauten Zellen. Nachdem die Klassifizierung und Manipulation der Umgebung durch den Agenten innerhalb eines Zeitschrittes abgeschlossen ist (siehe Abschnitt 3.4.2), wird die Drehrichtung des Agenten nach dem Zufallsprinzip neu festgelegt. Hierzu wird eine Zufallszahl zwischen 0 und 1 mit 360 multipliziert. Das Ergebnis bestimmt den Drehwinkel des Agenten. Anschließend bewegt sich der Agent in der bestimmten Drehrichtung um eine in der Variable `Speed` festgelegte Entfernung im zweidimensionalen Raum. Die Agenten bewegen sich nur auf Zellen, die bereits bebaut sind und verlassen die gebaute Wabenstruktur nicht.

3.5. Berechnung der Gesamtfitness der Wabenstruktur

Die Funktion `fitness` (Listing siehe Anhang A.4) dient zur Berechnung der Gesamtfitness einer in der Simulation entstandenen Wabenstruktur. Als Grundlage der Berechnung dient das Kriterium der *Kompaktheit*. Kompaktheit bedeutet in diesem

Kontext, dass die Wabenstruktur möglichst wenige Verästelungen oder Löcher aufweist, wobei eine kreisförmige Wabenstruktur den Idealfall von Kompaktheit darstellt. Auf Abbildung 3.3 a) ist eine stark verästelte Struktur mit relativ niedrigem Fitnesswert zu sehen, auf Abbildung 3.3 b) eine kompaktere Struktur mit wenig Verzweigungen und entsprechend höherem Fitnesswert.



(a) $F = 0.23$ (b) $F = 0.57$

Abbildung 3.3.: Vergleich zwischen zwei Wabenstrukturen mit (a) niedrigem und (b) hohem Fitnesswert F

Die Kompaktheit bzw. der globale Fitnesswert F der entstandenen Wabenstruktur wird nach jedem einzelnen Simulationslauf bestimmt. Die dazu verwendete Fitnessfunktion berechnet einen Wert, der den Grad der Kompaktheit der entstandenen Struktur angeben soll. Um die Kompaktheit einer Struktur als numerischen Wert darzustellen, wird die Gesamtfläche der Wabe durch ihren Umfang geteilt. Die Ausgangsidee für diese Berechnung ist, dass ein Kreis von allen geometrischen Figuren für eine gegebene Fläche den kleinsten Umfang besitzt und eine kreisförmige Wabe einen (nicht erreichbaren) Idealzustand darstellen würde. Zur Berechnung der Fitness wird das Verhältnis der Oberfläche A_W der Wabe zu ihrem Umfang U_W berechnet und angenommen, dass der Grad der Kompaktheit proportional zu diesem Wert verläuft. Da dieses Verhältnis bei einer kreisförmigen Figur maximal wäre, wird der Umfang U_K eines hypothetischen Kreises mit gleicher Fläche wie der entstandenen Wabenstruktur berechnet, und das Verhältnis des Kreisumfangs zur Fläche als idealer Fitnesswert F_{ideal} betrachtet. Aus den bekannten Kreisgleichungen $U_K = 2 \cdot r \cdot \pi$ und $A_K = r^2 \cdot \pi$ ergibt sich das Verhältnis

$$F_{ideal} = \frac{A_K}{U_K} = \frac{r}{2} = \frac{U_K}{4 \cdot \pi} \tag{3.1}$$

Der Quotient aus tatsächlichem Fitnesswert F und idealem Fitnesswert F_{ideal} ist der endgültige Fitnesswert F_{GA}, den der Genetische Algorithmus verwendet. Im Unterschied zum üblichen Vorgehen bei Lernenden Klassifizierenden Systemen, wird in diesem Modell der globale Fitnesswert nicht wie in Abschnitt 2.4.1 beschrieben mittels *Reinforcement Learning* zu allen beteiligten Regeln zurückpropagiert. Stattdessen erfolgt die Bewertung ausschließlich durch einen Genetischen Algorithmus, wie im folgenden Abschnitt erläutert wird.

3.6. Implementation des Genetischen Algorithmus

In der Funktion `strengthga` (Listing in Anhang A.6) ist ein Genetischer Algorithmus implementiert, der die in Abschnitt 2.3 beschriebenen Methoden verwendet, um neue Regelsätze für die Simulation des Wabenbaus zu erzeugen. In dieser Implementation bleiben dabei Detektor- und Effektor-String aller Klassifikatoren gleich, der Genetische Algorithmus verändert jedoch die *Stärke*-Werte der Klassifikatoren, mit dem Ziel, eine möglichst gute Verteilung der *Stärke*-Werte zu erreichen.

Es gibt zwei prinzipiell zu unterscheidende Möglichkeiten, einen Genetischen Algorithmus in Zusammenhang mit einem LCS anzuwenden. Diese unterscheiden sich darin, was als *Individuum* und *Population* für den GA definiert wird:

1. Das Individuum entspricht einem *Regelsatz*, also der *Gesamtheit aller Klassifikatoren* eines Agenten. Man arbeitet mit Agenten mit unterschiedlichen Regelsätzen, die zusammen genommen die Population des GA darstellen. Man testet das System mit den verschiedenen Regelsätzen und lässt den GA den bestmöglichen Regelsatz suchen. Dies entspricht dem Pittsburgh-Ansatz für LCS (siehe Abschnitt 2.4.2).

2. Das Individuum entspricht einem *einzelnen Klassifikator*. Der gesamte benutzte Regelsatz ist damit die Population des Genetischen Algorithmus. Das entspricht dem Michigan-Ansatz für LCS (siehe Abschnitt 2.4.2). Man arbeitet mit einem anfangs festgelegten Regelsatz, wobei die einzelnen Klassifikatoren durch den GA verändert werden.

Im Rahmen der vorliegenden Arbeit wurde ein LCS nach dem Pittsburgh-Ansatz implementiert und beschrieben. Eine ausführliche Betrachtung von Learning Classifier Systemen nach dem Michigan-Ansatz ist bei Butz (2006) zu finden und soll hier nicht näher behandelt werden. Eine Implementation für das vorliegende Problem könnte trotzdem interessante Ergebnisse bringen. Für den Genetischen Algorithmus stellt die Gesamtheit der verwendeten Regelsätze und die aus ihnen resultierenden Wabenstrukturen seine `Population` dar, in der jeder einzelne Regelsatz als `Individuum` zu sehen ist. *Genotyp* des Individuums ist die Menge der Stärke-Werte aller Klassifikatoren des entsprechenden Regelsatzes. Es wird die in `fitnessstring` gespeicherte Gray-Code-Darstellung jeder einzelnen Regel aneinandergehängt und in `classifierfitnessstring` abgespeichert. Hätte man zum Beispiel einen aus drei Klassifikatoren R_1, R_2 und R_3 bestehenden Regelsatz $\mathcal{R}$ und die drei Klassifikatoren hätten die folgenden Fitnesswerte in Gray-Code-Darstellung: `fitnessstring(`R_1`)` = '11010011', `fitnessstring(`R_2`)` = '00101101' und `fitnessstring(`R_3`)` = '01011010', dann wäre `classifierfitnessstring (`$\mathcal{R}$`)` = '110100110010110101011010' und repräsentiert den Genotyp des Individuums für den Genetischen Algorithmus.

Im beschriebenen Regelsatz von 31 Klassifikatoren (siehe Abbildung 3.2) mit einem 8-bit-Stärkewert besteht der Genotyp eines Individuums somit aus einer Kette von $8 \times 31 = 248$ Nullen und Einsen. Als *Phänotyp* des Individuums kann die durch Anwendung des Regelsatzes entstandene Wabenstruktur gesehen werden.

3.6.1. Parameter des Genetischen Algorithmus

In `strengthga.m` können drei Parameter für den Genetischen Algorithmus direkt angegeben werden:

- `probcrossover` gibt die Wahrscheinlichkeit p_{xover} an, mit der bei der Reproduktion der Individuen eine Rekombination der genetischen Codes der Eltern stattfindet.

- `probmutation` definiert die Mutationswahrscheinlichkeit p_{mut}, mit der ein ausgewähltes Bit in einem neu erzeugten Individuum verändert wird (siehe Abschnitt 2.3.2).

- `elternzahl` ist die Anzahl der Individuen, die pro Durchlauf des Genetischen Algorithmus für eine „Fortpflanzung“ ausgewählt werden. Der Wert muss ein Vielfaches von 2 sein, da jeweils zwei Individuen für einen Rekombinationsvorgang benötigt werden.

3.6.2. Ablauf des Genetischen Algorithmus

Selektion der Elternindividuen

Zur *Selektion* der Elternindividuen wird ein *Roulette-Wheel*-Selektionsalgorithmus (siehe Abschnitt 2.3.2) verwendet, der als Funktion `roulettemz` implementiert ist. Anhand der in `overallfitness` gespeicherten Fitnesswerte der gebauten Strukturen wird in jedem Durchlauf des Selektionsalgorithmus ein Agent ausgewählt, wobei die Auswahlwahrscheinlichkeit direkt proportional zum Fitnesswert ist. Es handelt sich um stochastische Auswahl *mit Zurücklegen*, das heißt jedes Individuum kann beliebig oft zur Replikation ausgewählt werden. Dadurch wird erreicht, dass sich Individuen mit überdurchschnittlich hoher Fitness auch überdurchschnittlich vermehren können.

Rekombination

Nach Auswahl der Elternindividuen erfolgt ihre Replikation durch die Funktion `onepointxover`. Die genetischen Strings der Elternindividuen (gespeichert in `classifierfitnessstring`) werden zuerst in zwei neue Strings kopiert. Danach wird eine stochastische Auswahl darüber getroffen, ob eine Rekombination der beiden Strings erfolgt. Mit Wahrscheinlichkeit p_{xover} werden die Strings rekombiniert, mit Wahrscheinlichkeit $1 - p_{xover}$ wird die Rekombination übersprungen. Entscheidet der Zufall für eine Rekombination, wird einfaches *Ein-Punkt-Crossover* angewendet (siehe Abschnitt 2.3.2), das heißt es wird eine zufällige Stelle auf dem Bitstring ausgewählt und die Teile der beiden neu erzeugten Strings werden hinter diesem Crossover-Punkt miteinander vertauscht. Abbildung 3.4 veranschaulicht diesen Vorgang.

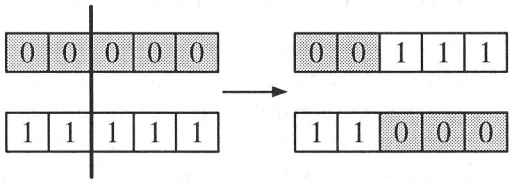


Abbildung 3.4.: Schematische Darstellung von *Ein-Punkt-Crossover* (Nissen, 1997)

Mutation

Auf die so entstandenen genetischen Strings wird im nächsten Schritt die Funktion `mutation` angewendet. Sie implementiert den in Abschnitt 2.3.2 beschriebenen Mutationsvorgang. Wie bei der Rekombination erfolgt vor der eigentlichen Durchführung der Mutation eine stochastische Entscheidung darüber, ob tatsächlich eine Mutation durchgeführt wird oder nicht. Mit Wahrscheinlichkeit p_{mut} wird ein Bit im String mutiert, mit Wahrscheinlichkeit $1 - p_{mut}$ nicht. Wird für eine Mutation entschieden, wird zufällig eine Stelle auf dem String gewählt und das entsprechende Bit gekippt, d.h. aus einer '0' wird eine '1' und aus einer '1' eine '0'.

Einfügen in die Population

Da die Größe der Population konstant gehalten werden soll, unter anderem um einen Selektionsdruck aufzubauen und Individuen mit niedrigerer Fitness aus der Population zu entfernen, müssen ebenso viele Individuen aus der bisherigen Population entfernt werden, wie bei der Elternselektion zur Vermehrung ausgewählt wurden. Die Auswahl der zu entfernenden Individuen erfolgt ebenfalls durch Roulette-Wheel-Selektion, die in Funktion `roulette` implementiert wurde. Es gibt zwei entscheidende Unterschiede zu dem bei der Selektion der Elternindividuen verwendeten Algorithmus. Die Auswahlwahrscheinlichkeit hängt nicht direkt proportional vom erreichten Fitnesswert ab wie bei der Elternselektion, sondern umgekehrt proportional. Je geringer die vom Individuum erreichte Fitness ist, desto höher ist die Wahrscheinlichkeit, dass es aus der Population gelöscht wird. Außerdem erfolgt in dieser Variante eine Auswahl aus der Population ohne Zurücklegen, d.h. jedes Individuum darf in einem Durchgang des GA nur einmal gewählt werden. Dies ist äußerst wichtig, da für jedes neu dazukommende Individuum ein anderes aus der Population gelöscht werden

muss. Würde man ein Individuum in diesem Schritt zweimal auswählen, könnte man es trotzdem nur einmal aus der Population entfernen, was eine relative Vergößerung der Population zur Folge hätte. Die beschriebenen Schritte des Genetischen Algorithmus werden so oft wiederholt, bis die festgelegte Anzahl von neuen Individuen erzeugt wurde.

3.7. Graphische Darstellung

Um eine Vorstellung davon vermitteln zu können, wie die in der Simulation entstandenen Strukturen aussehen, wurde eine graphische Darstellung des Simulationsraumes implementiert. MATLAB stellt umfangreiche Möglichkeiten zur Graphikprogrammierung zur Verfügung. Als Anforderungen an die Darstellung wurden eine übersichtliche Repräsentation des sechseckigen Gitternetzes, die farbige Markierung von bebauten Zellen mit je nach Bausteintyp unterschiedlichen Farben und die Möglichkeit der Markierung der momentanen Agentenposition festgelegt.

3.7.1. Darstellung des hexagonalen Gitters

Die Darstellung des Gitternetzes wurde in der Funktion `plotgitter` (Listing in Anhang A.8) realisiert. Um den Mittelpunkt jeder sechseckigen Zelle wird ein regelmäßiges sechseckiges `patch`-Objekt erzeugt, welches genau die Umrisse der Zelle nachzeichnet. `patch`-Objekte sind graphische Objekte in MATLAB und können beliebige Formen und Farben haben. Die hier benutzten Objekte haben eine weiße Oberfläche und schwarze Ränder, wodurch in ihrer Aneinanderreihung das Gitternetz entsteht (siehe Abbildung 3.5).

3.7.2. Darstellung der einzelnen Wabenzellen

Um die bebauten Wabenzellen zu markieren, wurde die Funktion `plotwabe` geschrieben. Die Funktion benötigt als übergebene Parameter den logischen Index der zu markierenden Zelle und ihre Farbe. Es wird ein zusätzliches sechseckiges `patch`-Objekt an der Position der Zelle erzeugt, wobei die Flächen des Objekts in der dem Bau-

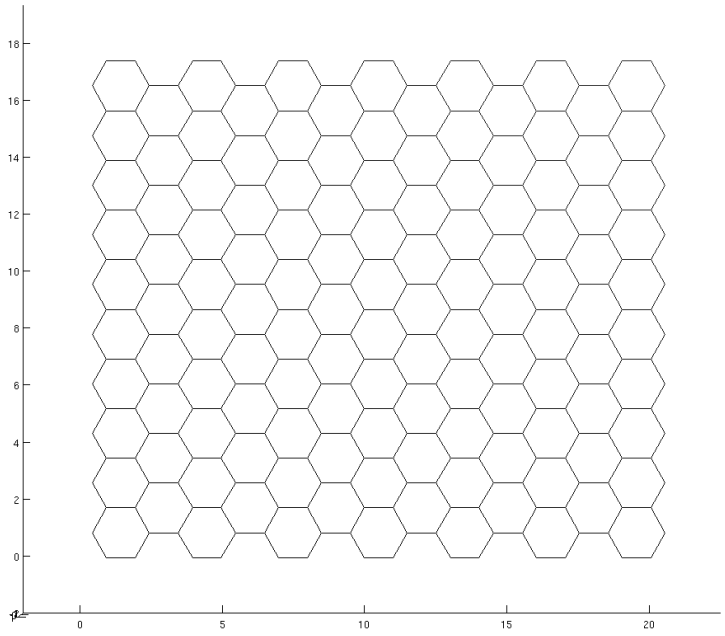


Abbildung 3.5.: Unbebaute sechseckige Gitterstruktur

steintyp entsprechenden Farbe eingefärbt werden. Um das Modell für eine spätere dreidimensionale Erweiterung offenzuhalten, wurden die Bausteine nicht als flache Sechsecke sondern als dreidimensionale Polygonobjekte modelliert. Ein Beispiel für eine von vier Agenten gleichzeitig gebaute Wabenstruktur findet sich in Abbildung 3.6.

3.8. Durchführung der Simulationen

Der Ablauf der mit Hilfe des in diesem Kapitel beschriebenen Modells durchgeführten Simulationen erfolgte nach dem folgendem Schema:

1. Festlegen der Simulationsparameter: Größe des Feldes, Anzahl der Agenten, Anzahl der Schritte pro Simulationslauf, Größe der Population des GA, Anzahl der Durchläufe des Genetischen Algorithmus

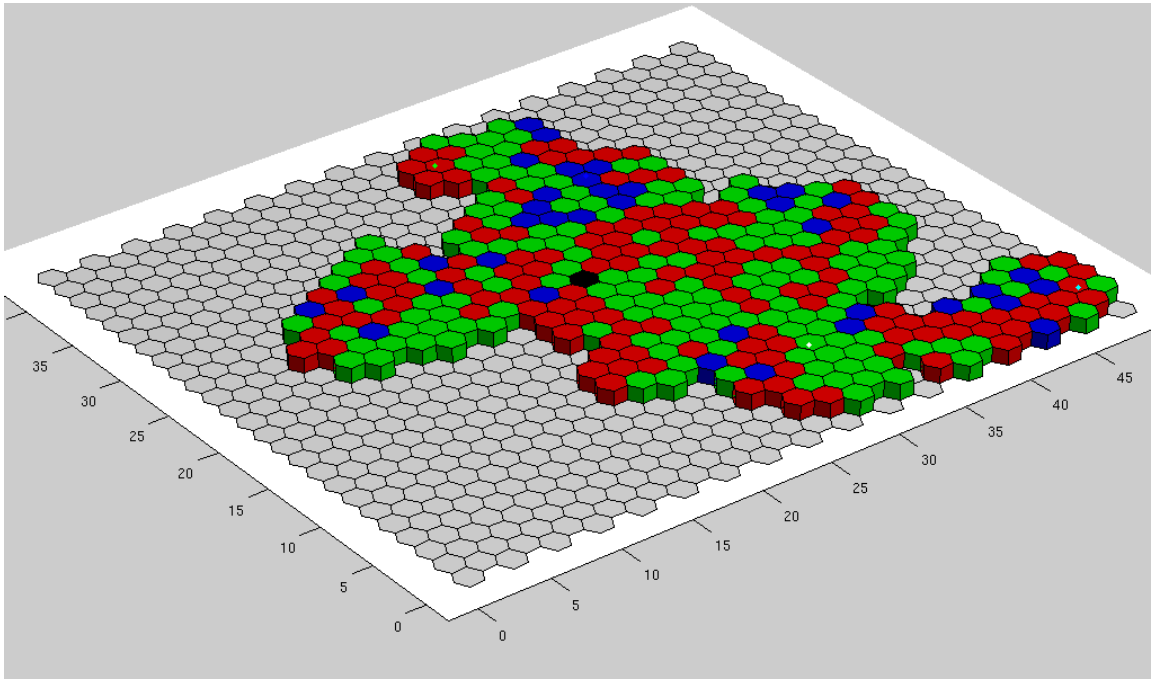


Abbildung 3.6.: Graphisches Beispiel einer von vier Agenten mit drei Bausteintypen gebauten Wabe

2. Festlegen des Ausgangsregelsatzes: Es wird ein Satz von Klassifikatoren definiert und für jeden Klassifikator seine Bedingung (Detektor-String B), die entsprechende Aktion (Effektor-String A) und sein anfänglicher Stärkewert S definiert. Die Stärkewerte der einzelnen Regeln werden für jede Simulation von neuem vergeben, wobei S für jeden Klassifikator als zufälliger ganzzahliger Wert zwischen 0 und 255 zugewiesen wird. Die Beschränkung auf ganze Zahlen S mit $0 \leq S \leq 255$ ist notwendig, da die Stärkewerte in 8-bit-Gray-Code abgespeichert werden (siehe Abschnitt 2.3.3).

3. Durchführung der Simulationsläufe

4. Auswertung der Ergebnisse

Die genauen Parameter der durchgeführten Simulationen sowie die erzielten Ergebnisse werden in Abschnitt 4 im Einzelnen betrachtet.

Die zum Sechseckgitter relative Positionsbestimmung stellte sich im Verlauf der Simulationen als sehr zeitraubend heraus. Ein Simulationslauf mit einer Population von 100 Individuen und 100 Durchläufen des Genetischen Algorithmus benötigte ca.

70 Stunden Rechenzeit. Ein Großteil der Rechenzeit entfiel dabei auf die Positionsbestimmung. Eine Neuimplementierung mit weniger Zeitaufwand wäre von Vorteil, wobei man alternativ dazu das Modell auch dahingehend vereinfachen könnte, dass die Position des Agenten nur durch den Index der Zelle definiert wird, auf dem sich der Agent befindet. Der Agent könnte sich damit nur von Zellenmittelpunkt zu Zellenmittelpunkt bewegen, was die zeitaufwendige Positionsbestimmung überflüssig machen würde. Es bleibt allerdings zu prüfen, ob dies nicht eine Vereinfachung wäre, die sich negativ auf das Verhalten des Modells auswirken würde.

Index	*B*	*A*	Grafik
1	110000	2	
2	011000	2	
3	001100	2	
4	000110	2	
5	000011	2	
6	100001	2	
7	211000	2	
8	021100	2	
9	002110	2	
10	000211	2	
11	100021	2	
12	110002	2	
13	112000	2	
14	011200	2	
15	001120	2	
16	000112	2	
17	200011	2	
18	120001	2	
19	211200	2	
20	021120	2	

Index	*B*	*A*	Grafik
21	002112	2	
22	200211	2	
23	120021	2	
24	112002	2	
25	212000	2	
26	021200	2	
27	002120	2	
28	000212	2	
29	200021	2	
30	120002	2	
31	120000	2	
32	012000	2	
33	001200	2	
34	000120	2	
35	000012	2	
36	200001	2	
37	210000	2	
38	021000	2	
39	002100	2	
40	000210	2	

Tabelle 3.3.: Die 78 Klassifikatoren des Regelsatzes der zweiten Hierarchieebene (Teil 1) mit Indexnummer, Detektor-String B, Effektor-String A und graphischer Darstellung

Index	*B*	*A*	Grafik
41	000021	2	
42	100002	2	
43	211100	2	
44	021110	2	
45	002111	2	
46	100211	2	
47	110021	2	
48	111002	2	
49	111200	2	
50	011120	2	
51	001112	2	
52	200111	2	
53	120011	2	
54	112001	2	
55	211110	2	
56	021111	2	
57	102111	2	
58	110211	2	
59	111021	2	

Index	*B*	*A*	Grafik
60	111102	2	
61	111120	2	
62	011112	2	
63	201111	2	
64	120111	2	
65	112011	2	
66	111201	2	
67	211111	2	
68	121111	2	
69	112111	2	
70	111211	2	
71	111121	2	
72	111112	2	
73	211120	2	
74	021112	2	
75	202111	2	
76	120211	2	
77	112021	2	
78	111202	2	

Tabelle 3.4.: Die 78 Klassifikatoren des Regelsatzes der zweiten Hierarchieebene (Teil 2) mit Indexnummer, Detektor-String B, Effektor-String A und graphischer Darstellung

4. Simulationsergebnisse

4.1. Simulationsergebnisse mit einem statischen Regelsatz und genetisch variierten Stärkewerten

In dieser Simulationsreihe wurde der Regelsatz aus Tabelle 3.2 verwendet. Die Stärkewerte der einzelnen Regeln werden in jedem Lauf neu per Zufall zugewiesen. In jedem Lauf gibt es nur einen einzelnen Agenten, der immer die gleiche Zahl von Zeitschritten läuft.

Die in Abschnitt 3.3.1 beschriebenen Simulationsparameter, die für alle Simulationen gleich waren:

- `DX` = 41
- `DY` = 41
- `wespenanzahl` = 1
  Um die Ergebnisse besser bewerten zu können, wurde mit nur einem gleichzeitig aktiven Agenten simuliert.
- `MaxSteps` = 200
- `probmutation` = 0.2

4.1.1. Ergebnisse ohne Beschränkung der Populationsgröße

Es wurde eine Simulation mit folgenden zusätzlichen Parametern durchgeführt:

- Größe der Population `numberofruns` = 20
- Anzahl der Durchläufe des GA `numberofgaruns` = 20
- Anzahl der pro Durchlauf des GA erzeugten neuen Individuen `elternzahl` = 2

Es wurde eine Anfangspopulation von 20 Agenten erzeugt, die wie zuvor beschrieben mit dem gleichen Regelsatz und unterschiedlichen Stärkewerten der einzelnen Regeln einen Bauvorgang simulierten. In dieser Simulation wurden durch den Genetischen Algorithmus keine Individuen aus der Population gelöscht. Wegen `elternzahl` = 2 kommen mit jedem Durchlauf des GA zwei neue Individuen zur Population dazu, weswegen die Population im Laufe der Simulation wächst. Am Ende der Simulation nach 20 Durchläufen umfasst die Population 60 Inidviduen. Nach Beendigung aller Simulationsläufe wurde jeweils die maximal erreichte globale Fitness F_{max} der entstandenen Strukturen festgehalten. Die Entwicklung des maximalen Fitnesswertes F_{max} ist in Abbildung 4.1 zu sehen. Trotz stochastischer Schwankungen ist in der Grafik ein deutlicher Aufwärtstrend zu erkennen, der vermuten lässt, dass der Genetische Algorithmus den gewünschten Effekt hat und seine weitere Anwendung noch zu einer weiteren Verbesserung geführt hätte.

4.1.2. Ergebnisse mit beschränkter Populationsgröße

Für diese Simulation galten die folgenden zusätzlichen Parameter:

- `numberofruns` = 100
- `numberofgaruns` = 100
- `elternzahl` = 4

In dieser Simulation wurde für jedes neu erzeugte Individuum je ein Individuum aus der Population entfernt, wie in Abschnitt 3.6.2 beschrieben. Dies entspricht dem

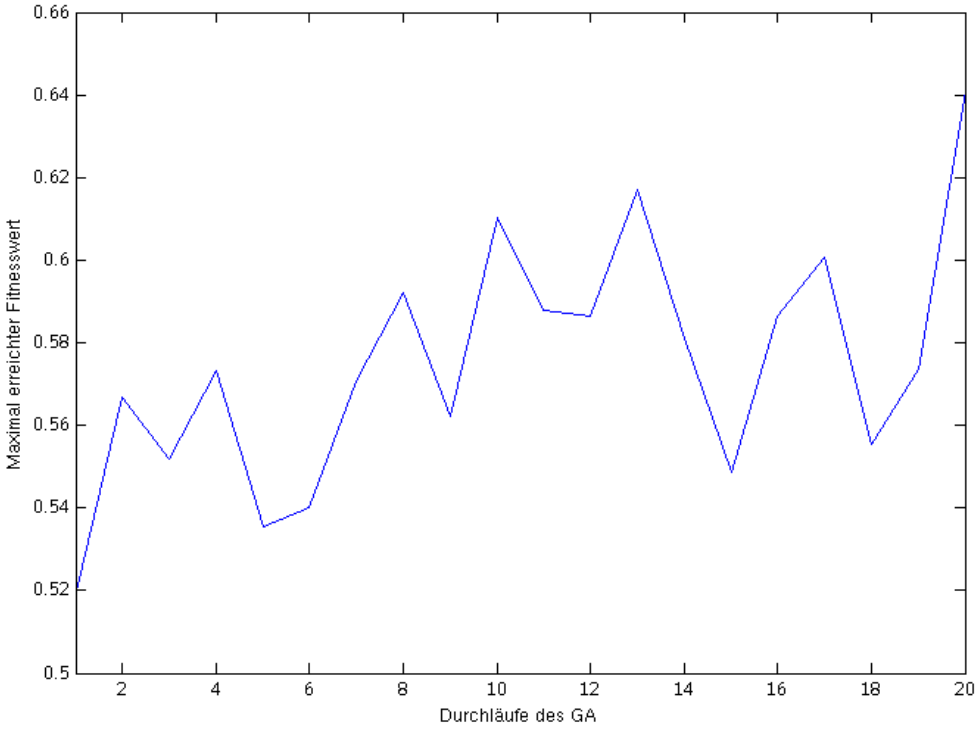


Abbildung 4.1.: Entwicklung der maximal erreichten Overall-Fitness einer Anfangspopulation von 20 Individuen nach 20 Anwendungen des GA

biologischen Vorbild des Selektionsdruckes (siehe Abschnitt 2.2) und verspricht eigentlich bessere Ergebnisse.

Es wurde der maximal erreichte Fitnesswert F_{max} unter allen 100 Individuen ermittelt und abgespeichert. Zusätzlich wurde diesmal auch ein Wert F_{avg} berechnet, der den durchschnittlichen Fitnesswert der Population nach jedem Durchlauf des Genetischen Algorithmus repräsentiert. Die Entwicklung von F_{max} und F_{avg} nach wiederholter Anwendung des GA ist in Abbildung 4.2 dargestellt. Für beide Werte ist in Grafik 4.2 keine eindeutige Tendenz nach oben oder unten zu erkennen. Beide Werte schwanken zwar mehr oder weniger stark, tendieren aber nicht eindeutig zu einer Verbesserung. Wie dieses Ergebnis zu bewerten ist, soll noch ausführlicher in Abschnitt 5 diskutiert werden.

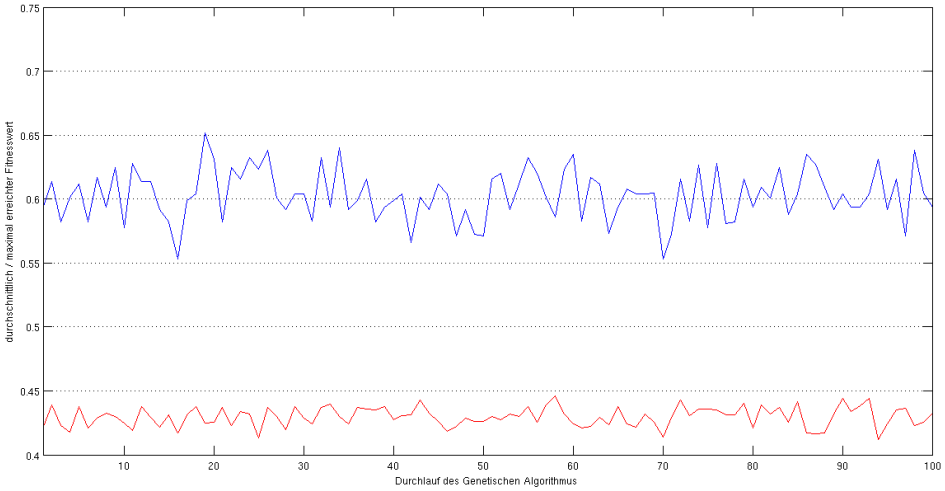


Abbildung 4.2.: Entwicklung der durchschnittlich (F_{avg}, rot) und maximal (F_{max}, blau) erreichten globalen Fitness einer Population von 100 Individuen über 100 Anwendungen des GA

4.1.3. Ergebnisse mit stochastischer Entscheidung über Crossover

Da die Ergebnisse der vorhergehenden Simulation (s. Abschnitt 4.1.2) nicht zufriedenstellend waren, wurde das Programm nochmals um einen entscheidenden Punkt verändert; es wurde ein Parameter p_{xover} als Variable `probxover` in `strengthga.m` eingeführt, der die Crossover-Wahrscheinlichkeit für den Genetischen Algorithmus festlegt. Mit Wahrscheinlichkeit p_{xover} wird also ein Crossover durchgeführt, mit Wahrscheinlichkeit $1 - p_{xover}$ werden die genetischen Strings der Eltern-Individuen unverändert auf die Kinder kopiert. Diese Änderung kann unter Umständen verhindern, dass Individuen mit hoher Fitness durch Crossover zerstört werden. Außerdem wurde in diesem Simulationslauf der Selektionsdruck dadurch erhöht, dass pro Durchlauf des Genetischen Algorithmus 20 Eltern-Individuen mit Roulette-Wheel-Selektion ausgewählt und somit auch 20 neue Individuen erzeugt wurden.

Es wurde also mit folgenden Parametern simuliert:

- `numberofruns` = 100
- `numberofgaruns` = 20

- `elternzahl` = 20
- Crossover-Wahrscheinlichkeit (p_{xover}) `probxover` = 0.6

Die erreichten maximalen bzw. durchschnittlichen Fitnesswerte F_{max} und F_{avg} nach jedem Lauf des GA sind in der Grafik 4.3 zu sehen. Die Ergebnisse unterscheiden

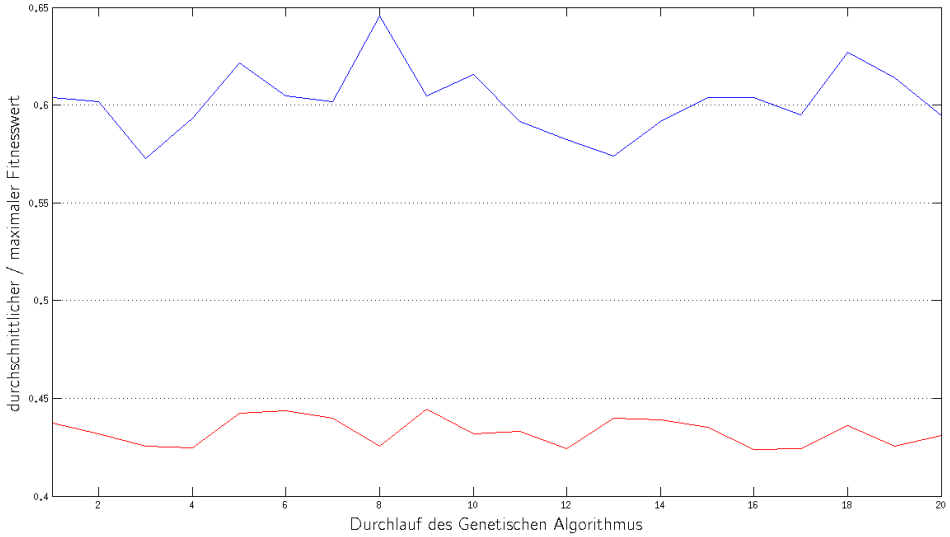


Abbildung 4.3.: Entwicklung von F_{max} (blau) und F_{avg} (rot) bei einer Population von 100 Individuen über 20 Anwendungen des GA

sich kaum von denen in Abschnitt 4.1.2. Auch hier ist keine Verbesserung oder Verschlechterung des erreichten Fitnesswertes mit wiederholter Anwendung des GA zu erkennen. Vielmehr bleibt der Fitnesswert von stochastischen Abweichungen abgesehen mehr oder weniger konstant.

4.1.4. Ergebnisse mit Mehrfachauswahl der Elternindividuen

In den vorherigen Simulationsläufen konnte ein Individuum aus der Population nur einmal als Elternindividuum gewählt werden. Da dies der Vorstellung widerspricht, dass sich Individuen mit hoher Fitness möglichst oft vermehren können, wurde die Implementation für diese Simulation abgeändert, so dass ein Individuum in einem Durchlauf des GA beliebig oft ausgewählt werden kann. Der Roulette-Wheel-Selektionsalgorithmus wurde also *mit Zurücklegen* programmiert (siehe Abschnitt 3.6.2).

Die Simulationsparameter waren wie folgt:

- `numberofruns` = 200
- `numberofgaruns` = 100
- `elternzahl` = 100
- `probxover` = 0.6

Die minimal (F_{min}), durchschnittlich (F_{avg}) und maximal (F_{max}) erreichten Fitnesswerte nach jedem Durchlauf des Genetischen Algorithmus sind in Abbildung 4.4 dargestellt.

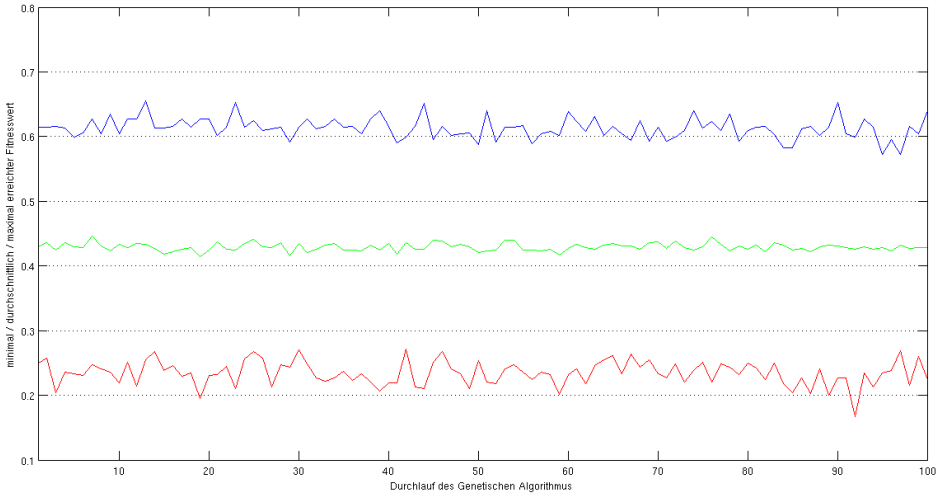


Abbildung 4.4.: Entwicklung von F_{max} (blau), F_{avg} (grün) und F_{min} (rot) bei einer Population von 200 Individuen über 100 Anwendungen des Genetischen Algorithmus

4.2. Simulation mit statischem Regelsatz ohne Genetischen Algorithmus

Um zu ermitteln, inwieweit das Schwanken der Fitnesswerte in Abschnitt 4.1 zufällig ist und ob überhaupt ein direkter Zusammenhang zwischen der Verteilung der

Stärkewerte der einzelnen Klassifikatoren und der Fitness der gebauten Struktur besteht, wurde eine Simulation durchgeführt, in der wie in Abschnitt 4.1 der Regelsatz aus Tabelle 3.2 verwendet wurde. Der Programmcode wurde für diese Simulation angepasst. Für die jedem Klassifikator des Individuums zu Beginn der Simulation zugewiesenen Stärkewerte wurden keine zufälligen Werte zwischen 0 und 255 erzeugt, sondern jeder Regel ein bestimmter Wert zugewiesen. Der Genetische Algorithmus wurde außerdem abgeschaltet. In diesem Experiment wurden also hintereinander 100 Wabenstrukturen von 100 identischen Individuen mit identischem Regelsatz gebaut und die erzielten Ergebnisse verglichen.

Die relevanten Simulationsparameter im Detail:

- `DX = 41`
- `DY = 41`
- `wespenanzahl = 1`
- `MaxSteps = 200`
- `numberofruns = 100`
- Regelsatz siehe Tabelle 3.2
- Stärkewerte der Klassifikatoren siehe Tabelle 4.1. Die Werte wurden so gewählt, dass die Wahrscheinlichkeit für den Bau einer Zelle mit der Anzahl der bereits bebauten Nachbarzellen ansteigt. Dies entspricht den empirischen Beobachtungen beim Nestbau realer Wespen (siehe Abschnitt 2.1.1) und sollte der Erwartung nach Strukturen mit hoher Fitness erzeugen.

In Abbildung 4.5 sind die von den 100 identischen Individuen erzielten Fitnesswerte dargestellt. Es ist sehr starkes Schwanken festzustellen. Die Fitnesswerte schwanken zwischen $F_{min} = 0.24$ und $F_{max} = 0.63$ bei einer durchschnittlichen Fitness $F_{avg} = 0.43$.

Zur weiteren Veranschaulichung zeigt Abbildung 4.6 die beiden zu F_{min} und F_{max} gehörenden Strukturen. Die beiden Waben wurden von identischen Individuen mit gleichem Regelsatz und ebenso gleicher Stärkeverteilung der Klassifikatoren erzeugt.

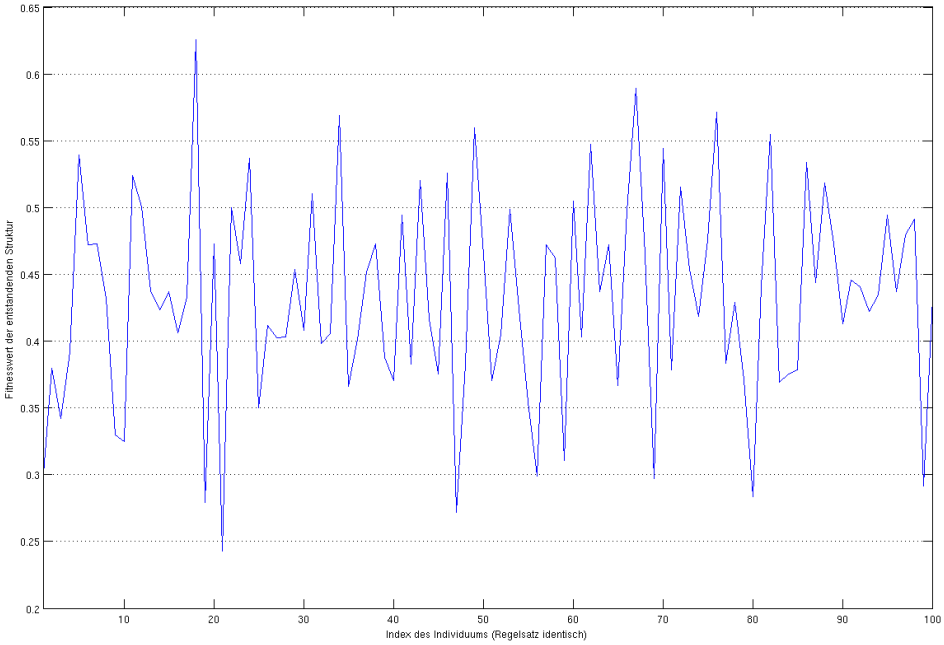
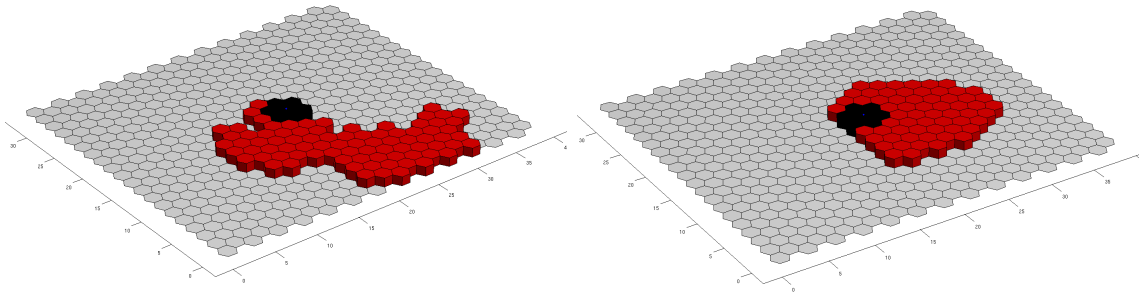


Abbildung 4.5.: Vergleich der Fitnesswerte der von 100 identischen Individuen gebauten Wabenstrukturen

Die starken Unterschied in optischem Erscheinungsbild und Fitnesswert können daher nur durch die zufällige Bewegung der Agenten und die stochastische Auswahl über die zu aktivierende Regel zustande kommen.



(a) $F_{21} = 0.24$ (b) $F_{18} = 0.63$

Abbildung 4.6.: Von identischen Individuen (a) Nr. 21 und (b) Nr. 18 gebaute Strukturen mit zugehörigem Fitnesswert F

4.3. Simulation mit zwei hierarchisch geordneten Regelsätzen

Durch die Erweiterung des Simulationsmodells um einen zweiten Regelsatz, der nach einer festgesetzten Anzahl von Simulationsschritten an Stelle des ersten Regelsatzes aktiviert wird (siehe Abschnitt 3.4.3), konnten komplexere Strukturen gebaut werden. Mit dem verwendeten zweiten Regelsatz, der bereits in Tabelle 3.3 und 3.4 dargestellt wurde, wurde um die bereits mit Regelsatz 1 gebaute Struktur (Bausteintyp 1, rot) ein geschlossener Ring gebaut (Bausteintyp 2, grün), wie in Abbildung 4.7 zu sehen ist. Es wurden folgende Parameter verwendet:

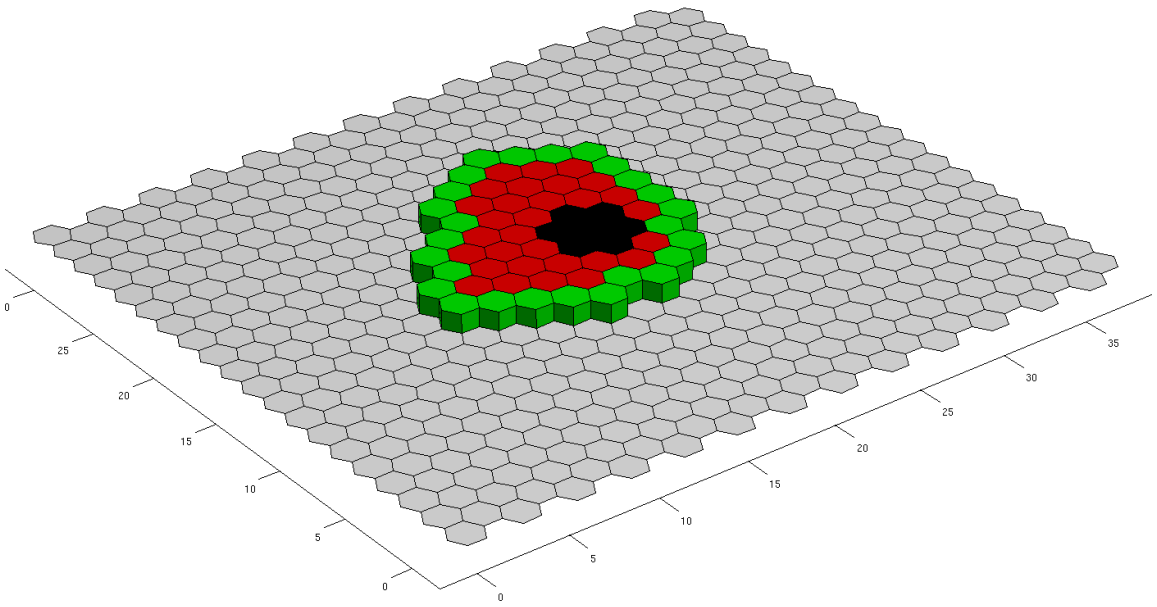


Abbildung 4.7.: Endzustand einer Beispielsimulation mit zwei hierarchisch hintereinandergeschalteten Regelsätzen

- `DX = 41`
- `DY = 41`
- `wespenanzahl = 1`

- `MaxSteps = 100`
- `MaxSteps2 = 1000`
- Zahl der hierarchischen Ebenen von Klassifikatoren: `rulesetanzahl = 2`
- Regelsatz der ersten hierarchischen Ebene: `RuleSet1` (Tabelle 3.2)
- Regelsatz der zweiten hierarchischen Ebene: `RuleSet2` (Tabellen 3.3 und 3.4)

`MaxSteps` und `MaxSteps2` sind so zu interpretieren, dass die Simulation nach `MaxSteps` Schritten auf den zweiten Regelsatz umschaltet und dann nach insgesamt `MaxSteps2` Schritten die Simulation beendet. In diesem Beispiel wurde also 100 Schritte lang `RuleSet1` verwendet und danach 900 Schritte lang `RuleSet2`. `MaxSteps2` wurde vergleichsweise hoch gewählt, um zu gewährleisten, dass der Agent trotz stochastischer Bewegung die Gelegenheit bekommt, an jede Randposition der Struktur zu kommen, und dadurch einen geschlossenen Ring grüner Bausteine zu bauen.

Regelindex	Stärkewert S
1	255
2	250
3	250
4	250
5	250
6	250
7	250
8	230
9	230
10	230
11	230
12	230
13	230
14	150
15	150
16	150
17	150
18	150
19	150
20	50
21	50
22	50
23	50
24	50
25	50
26	250
27	250
28	250
29	250
30	250
31	250

Tabelle 4.1.: Stärkewerte des Regelsatzes in Simulation 4.2

5. Diskussion

Die zentrale Hypothese dieser Arbeit war, dass es möglich sein müsste, anhand eines agentenbasierten Simulationsmodells auf Basis Genetischer Algorithmen und Lernender Klassifizierender Systeme den Nestbau von Insekten abstrakt nachzubilden und die Parameter des Modells während der Simulation so anzupassen, dass sich die erzeugten Strukturen nach vorgegebenen Fitness-Kriterien immer weiter verbessern, bis ein Optimum erreicht ist. Diese Hypothese konnte anhand der durchgeführten Simulationen nicht bestätigt werden.

Was gezeigt werden konnte ist, dass das Modell bei Wahl eines geeigneten Regelsatzes in der Lage ist, eine zusammenhängende Wabenstruktur zu erzeugen, die keine Löcher aufweist. Was das gewählte Fitness-Kriterium der Kompaktheit betrifft, konnte jedoch in sämtlichen Simulationsläufen keine Verbesserung durch Anwendung eines Genetischen Algorithmus beobachtet werden. Die durchschnittliche Fitness der Population bleibt auch bei häufiger Anwendung des Genetischen Algorithmus von stochastischen Abweichungen abgesehen mehr oder weniger konstant. Der Genetische Algorithmus konvergiert also nicht zu einem Optimum hin, wobei sich die Frage stellt, warum er das nicht tut.

Viele Mechanismen und Parameter des Modells hätten anders implementiert werden können. Es wurde zum Beispiel kein *Reinforcement Learning* (siehe Abschnitt 2.4.1) für die Rückpropagierung der globalen Fitness auf die beteiligten Klassifikatoren verwendet. Stattdessen wurde der Genetische Algorithmus auf die Stärkewerte der Klassifikatoren angewendet, um die Stärkewerte zu optimieren. In den meisten Lernenden Klassifizierenden Systemen wird im Gegensatz dazu der Genetische Algorithmus auf die Klassifikatoren selbst angewendet, das heißt es entstehen in solchen Systemen tatsächlich neue Regelsätze (vgl. Bernauer (2007)), während im hier implementierten Modell der Regelsatz selbst gleich bleibt und sich nur die Wahrscheinlichkeit für die Anwendung der einzelnen Regeln ändert.

Auch in der Implementation der Fitnessfunktion sind Änderungen denkbar. Um eine Wabenstruktur zu erzeugen, die den an reale Nester gestellten Ansprüchen nahekommt, müssten außer dem Kriterium der Kompaktheit noch andere Parameter in die Berechnung der Fitness miteinfließen. Bei Betrachtung der erzeugten Strukturen fällt auf, dass diese meist nicht konzentrisch um den Mittelpunkt gebaut sind. Fasst man den Mittelpunkt der Wabe als Aufhängung an einem Ast (*Pedikel*, siehe Abschnitt 2.1) auf, so müsste die Wabe eine gewisse Symmetrie in Bezug auf diesen Mittelpunkt aufweisen, um stabil zu sein. Ein wichtiger zusätzlicher Faktor für die Fitness wäre also die Symmetrie der Wabe. Mit dem hier vorgestellten Modell wird es allerdings auch schwierig sein, eine symmetrische Wabe zu bauen. Der Grund dafür ist die rein stochastische Bewegung der Agenten auf der Wabe. Ein neues Bewegungsmodell für die Agenten wäre nötig, in dem diese sich zielgerichteter bewegen würden. Die Agenten könnten sich zum Beispiel spiralförmig vom Mittelpunkt ausgehend mit größer werdendem Radius über die Wabe bewegen, ähnlich wie dies beim Bau von Spinnennetzen zu beobachten ist (Krink und Vollrath, 1997). Auch die Schrittweite der Agenten, die im Modell konstant ist, könnte verändert werden. In der aktuellen Implementierung bewegen sich die Agenten pro Zeitschritt um eine festgelegte konstante Schrittweite in eine zufällig ausgewählte Richtung. Da dieses vereinfachte Bewegungsmodell die Realität nicht sehr genau abbildet, könnte es eine Verbesserung für das Simulationsmodell darstellen, die Bewegung der Agenten komplexer zu gestalten. Eine naheliegende Lösung wäre, auch die Bewegungsrichtung von Regeln abhängig zu machen, so dass sich die Agenten gezielt in die Richtung bewegen könnten, in der noch gebaut werden muss. Man könnte dazu die Mege der möglichen Aktionen, die im Moment auf Bauaktionen beschränkt sind, um Bewegungsaktionen erweitern. Jeder Agent würde dann in jedem Zeitschritt entweder eine Bewegung ausführen oder eine neue Zelle bauen.

Eine weitere wichtige Frage ist, ob der Wahrnehmungsbereich der Agenten groß genug gewählt ist. Die Regeln, nach denen die Agenten bauen, berücksichtigen nur diejenigen Zellen, die direkt an die potentielle Bauposition angrenzen. Dieser Ansatz ist vermutlich zu eng, um dem realen Vorbild gerecht zu werden. Auch wenn ein einzelnes Insekt nicht in der Lage ist, eine komplexe Struktur wie ein großes Wespennest im Ganzen zu erfassen, kann es den Zustand von Zellen berücksichtigen, die weiter entfernt von ihm sind. Es ist daher davon auszugehen, dass der Zustand der Wabe an einem weiter entfernten Ort durchaus Einfluss auf das Verhalten des einzelnen

Individuums nimmt. Man könnte das Modell in der Form erweitern, dass man für weiter entfernte Zustände eigene Regelsätze aufstellt und diese hierarchisch zueinander geordnet parallel aktiv sind, wobei Regelsätze für den Nahbereich Vorrang vor dem Fernbereich haben. Auch das wäre eine Form der hierarchischen Ordnung von Klassifikatoren, wobei die Regelsätze nicht wie im in Abschnitt 4.3 beschriebenen Beispiel nacheinander, sondern gleichzeitig aktiviert wären.

Diese Idee der gleichzeitig aktiven hierarchisch geordneten Klassifikatoren kann man beliebig erweitern. Man könnte für verschiedene Fitnesskriterien wie Größe, Symmetrie oder Stabilität der Wabe eigene Sätze von Klassifikatoren einführen, die parallel aktiv sind und zwischen denen abhängig vom äußeren Systemzustand umgeschaltet werden kann. Bei diesen Ideen zur Erweiterung des Modells muss allerdings berücksichtigt werden, dass jeder zusätzliche Parameter des Modells auch das Optimierungsproblem komplexer macht. Ein lernendes Sytem zu entwerfen, das zu einem Optimum konvergiert, wäre in einem Modell mit mehreren Regelsätzen noch schwieriger als im aktuell implementierten Modell.

Der Ansatz der im Rahmen dieser Arbeit realisierten Implementation orientiert sich im Wesentlichen an den Arbeiten von Theraulaz und Bonabeau über ihr Simulationsmodell zum Bau komplexer Strukturen durch Schwärme von Agenten (Theraulaz und Bonabeau, 1995; Bonabeau et al., 1999, 2000). Es gibt jedoch einige entscheidende Unterschiede in der Implementation. Bei Bonabeau et al. (1999) wurde ein *deterministischer* Algorithmus zur Regelauswahl der Agenten realisiert. Bei deterministischen Algorithmen werden Zufallsentscheidungen ausgeschlossen, daher wird bei Bonabeau et al. (1999) eine passende Regel immer mit Wahrscheinlichkeit 1 angewendet. Das impliziert auch, dass in dem dort implementierten Modell keine Regeln gleichzeitig zu einem bestimmten Input passen dürfen, weil der deterministische Algorithmus nicht dafür konzipiert ist, zwischen zwei passenden Regeln zu entscheiden. In der vorliegenden Arbeit wurde im Gegensatz dazu ein *stochastischer* Algorithmus zur Regelauswahl implementiert. Bei stochastischen Algorithmen werden zufällige Auswahlkriterien berücksichtigt, so dass wie in Abschnitt 3.4.2 beschrieben auch mehrere Regeln auf einen Input passen können, da der stochastische Algorithmus in der Lage ist, eine der passenden Regeln auszuwählen. Bonabeau et al. (1999) hatten ihr Modell dreidimensional erweitert, was dem dort beschriebenen Modell die Erzeugung komplexer Strukturen ermöglicht, die dem Erscheinungsbild realer Wespennester nä-

her kommen als im zweidimensionalen Raum möglich. Einige Beispiele für in diesem Modell entstandene dreidimensionale Strukturen finden sich in Abbildung 5.1.

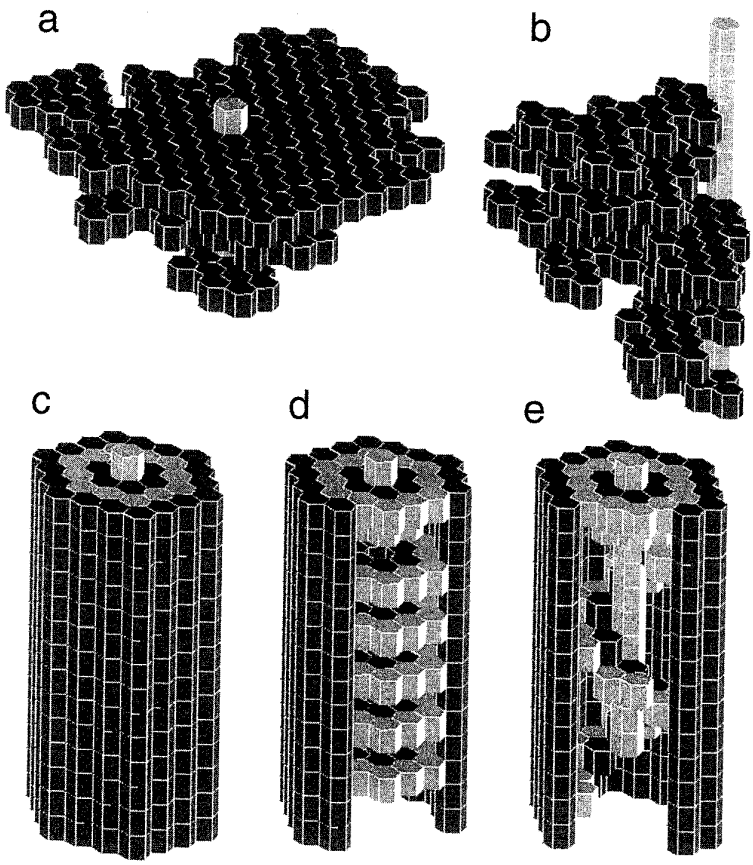


Abbildung 5.1.: Dreidimensionale Wabenstrukturen, erzeugt durch den Baulgorithmus von Bonabeau et al. (2000)

Eine Erweiterung des in dieser Arbeit vorgestellten Modells wäre mit relativ wenig Aufwand möglich, indem man eine Anzahl von Ebenen übereinanderlegt. Man müsste allerdings die Bauregeln der Agenten dahingehend erweitern, dass auch die über und unter dem Agenten liegenden Ebenen berücksichtigt werden, und außerdem festlegen, wann ein Agent in eine andere Ebene wechselt.

Weiterhin werden in der implementierten Regelverarbeitung nur Einzelaktionen betrachtet. Wird ein Klassifikator aktiviert, so baut der entsprechende Agent genau eine neue Zelle. Um komplexere Strukturen zu erzeugen könnten mehrere Aktionen zu einer Sequenz zusammengefasst und als Ganzes aktiviert und bewertet werden. Da

Aktionen vorstellbar sind, die nur in Zusammenhang mit anderen Aktionen sinnvoll sind, erscheint eine solche Erweiterung der Regeldefinitionen vielversprechend.

Eine weitere interessante Frage ist, inwieweit sich die von einem einzelnen Agenten gebauten Wabenstrukturen von denen eines Schwarms von Agenten unterscheidet. In der vorliegenden Arbeit wurden nur Simulationen mit einem einzelnen gleichzeitig aktiven Agenten vorgestellt. Simulationen mit Schwärmen von mehreren Agenten würden unter Umständen zu anderen Ergebnissen führen. Vor allem stellt sich bei Simulationen mit mehreren gleichzeitig bauenden Agenten die Frage, ob alle Agenten den gleichen Regelsatz verwenden, oder ob sich die Regelsätze unterscheiden. Eine Betrachtung der Natur würde auf den ersten Blick nahe legen, für jeden Agenten einen individuellen Regelsatz zu definieren, da es in Kolonien sozialer Wespen wie bei allen staatenbildenden Insekten verschiedene Kasten von Individuen gibt, die für verschiedene Aufgaben zuständig sind (z.B. Arbeiterin, Drohne, Königin bei Bienen), und Angehörige verschiedener Kasten innerhalb eines Insektenstaates offensichtlich unterschiedlichen Verhaltensregeln unterliegen (Wilson, 1971). Auf der anderen Seite können sich die Regeln, nach denen Insekten einer einzelnen Spezies agieren, auch nicht völlig grundsätzlich voneinander unterscheiden, sonst würde man die entstehenden Strukturen nicht so eindeutig bestimmten Arten zuordnen können. So haben die Nester der meisten Wespenarten charakteristische architektonische Eigenschaften, durch die sie sich eindeutig einer bestimmten Spezies zuordnen lassen (Matsuura und Yamane, 1990). Auch hier wäre die Methode, Hierarchien von Klassifikatoren einzuführen, ein Lösungsansatz. Jeder Agent bekommt die gleichen Regelsätze, wobei sich die hierarchische Ordnung der Regelsätze untereinander von Individuum zu Individuum unterscheidet.

Was in dieser Arbeit allerdings gezeigt werden konnte, ist die Eignung eines Systems hierarchischer Klassifikatoren zum Bau komplexer Strukturen, die mit einem einzelnen Regelsatz nicht erzeugt werden können. Wie in Abschnitt 4.3 zu sehen ist, konnte durch eine Kombination von zwei Regelsätzen, die nacheinander benutzt werden, eine Ringstruktur erzeugt werden, die an die Strukturen erinnert, die in den Waben der Honigbiene zu finden sind. In Abbildung 5.2 ist eine Bienenwabe zu sehen, die eine ringförmige Struktur aufweist. Im inneren Ring wird die Brut der Bienen aufgezogen und in den äußeren Ringen Honig und Pollen gelagert. Im Vergleich mit Abbildung 4.7 zeigt sich die Ähnlichkeit der Ringstrukturen.

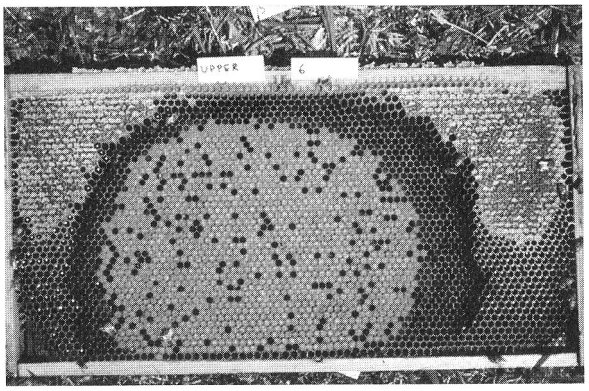


Abbildung 5.2.: Wabe eines Bienenschwarms mit drei konzentrischen Zellregionen (Bonabeau et al., 1999)

Zusammenfassend lässt sich festhalten, dass das vorgestellte Modell zum Wabenbau von Insekten in der aktuellen Form noch nicht komplex genug ist, um die in der Natur zu beobachtenden Vorgänge abzubilden. Auch die Hypothese, dass durch Einsatz von Genetischen Algorithmen und Lernenden Klassifizierenden Systemen eine Anpassung der Bauregeln an festgelegte Fitnesskriterien erfolgen müsste, konnte nicht bestätigt werden. Vermutlich liegt dies an der Art und Weise, wie das LCS und der Genetische Algorithmus im Detail implementiert wurden. Hier wären noch viele Experimente mit unterschiedlichen Implementationen nötig, um zu einem funktionierenden Modell zu kommen. Es konnten allerdings Ansätze geschaffen werden, deren weitere Verfolgung große Fortschritte verspricht. Vor allem der Einsatz hierarchisch geordneter Klassifikatoren stellt eine entscheidende Neuerung für diese Anwendung dar, deren Resultate sehr vielversprechend sind. Eine Weiterentwicklung dieses Konzepts wäre möglicherweise in der Lage, viele der momentanen Einschränkungen des Modells zu beseitigen und Wabenstrukturen zu bauen, die den Strukturen realer Wespennester ähnlicher sind als die, die bisher mit vergleichbaren Modellen erzeugt wurden.

Literaturverzeichnis

[Bernauer 2007] Bernauer, Andreas: *Das Learning Classifier System XCS*. 2007. – URL `http://www-ti.informatik.uni-tuebingen.de/~bernauer/Bernauer07_XCS.pdf`

[Bonabeau et al. 1999] Bonabeau, Eric ; Dorigo, Marco ; Theraulaz, Guy: *Swarm Intelligence. From Natural to Artificial Systems*. Oxford University Press, 1999

[Bonabeau et al. 2000] Bonabeau, Eric ; Guérin, Sylvain ; Snyers, Dominique ; Kuntz, Pascale ; Theraulaz, Guy: Three-dimensional architectures grown by simple 'stigmergic' agents. In: *BioSystems* 56 (2000), S. 13–32

[Bull und Kovacs 2005] Bull, Larry ; Kovacs, Tim: Foundations of Learning Classifier Systems: An Introduction. In: Bull, Larry (Hrsg.) ; Kovacs, Tim (Hrsg.): *Foundations of Learning Classifier Systems*. Berlin : Springer, 2005, S. 1–17

[Butz 2006] Butz, Martin V.: *Studies in Fuzziness and Soft Computing*. Bd. 191: *Rule-Based Evolutionary Online Learning Systems : A Principled Approach to LCS Analysis and Design*. Berlin : Springer, 2006

[Camazine et al. 2001] Camazine, Scott ; Deneubourg, Jean-Louis ; Franks, Nigel R. ; Sneyd, James ; Theraulaz, Guy ; Bonabeau, Eric: *Self-Organization in Biological Systems*. Princeton University Press, 2001

[Darwin 1988] Darwin, Charles: *On the Origin of Species by Means of Natural Selection*. London : William Pickering, 1988

[Deneubourg et al. 1992] Deneubourg, Jean-Louis ; Theraulaz, Guy ; Beckers, Ralph: Swarm-Made Architectures. In: Varela, Francisco J. (Hrsg.) ; Bourgine, Paul (Hrsg.): *Toward a Practice of Autonomous Systems : Proceedings of the First European Conference on Artificial Life*. Cambridge, MA, USA : MIT Press, 1992, S. 123–133

[Eigen und Schuster 1979] Eigen, Manfred ; Schuster, Peter: *The Hypercycle : a principle of natural self-organization*. Berlin : Springer, 1979

[Floreano und Mattiussi 2008] Floreano, Dario ; Mattiussi, Claudio: *Bio-Inspired Artificial Intelligence*. Cambridge, MA, USA : MIT Press, 2008

[Geyer-Schulz 1995] Geyer-Schulz, Andreas: Holland classifier systems. In: *APL '95: Proceedings of the international conference on Applied programming languages*. New York, NY, USA : ACM, 1995, S. 43–55

[Goldberg 1989] Goldberg, David E.: *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison Wesley Longman, 1989

[Grassé 1959] Grassé, Pierre-Paul: La reconstruction du nid et le coordinations inter-individuelles chez *Bellicositermes natalensis* et *Cubitermes sp.* La theorie de la stigmergie: Essai d'interpretation du comportement des termites constructeurs. In: *Insectes Sociaux* 6 (1959), S. 41–81

[Holland 1975] Holland, John H.: *Adaptation in natural and artificial systems : an introductory analysis with applications to biology, control, and artificial intelligence*. Ann Arbor : University of Michigan Press, 1975

[Holland 1976] Holland, John H.: Adaptation. In: Rosen, Robert (Hrsg.) ; Snell, Fred M. (Hrsg.): *Progress in Theoretical Biology 4*. New York : Academic Press, 1976, S. 263–293

[Holland 1986] Holland, John H.: A Mathematical Framework for Studying Learning in Classifier Systems. In: *Physica D* 2 (1986), Nr. 1-3, S. 307–317

[Holland 1992] Holland, John H.: *Adaptation in Natural and Artificial Systems: an introductory analysis with applications to biology, control, and artificial intelligence*. 2. Auflage. Cambridge, MA, USA : MIT Press, 1992

[Hölldobler und Wilson 1990] Hölldobler, Bert ; Wilson, Edward O.: *The Ants*. Berlin : Springer, 1990

[Karsai 1999] Karsai, István: Decentralized control of construction behavior in paper wasps: an overview of the stigmergy approach. In: *Artif. Life* 5 (1999), Nr. 2, S. 117–136

[Krink und Vollrath 1997] Krink, Thiemo ; Vollrath, Fritz: Analysing Spider Web-building Behaviour with Rule-based Simulations and Genetic Algorithms. In: *Journal of Theoretical Biology* 185 (1997), Nr. 3, S. 321–331. – URL `http://www.sciencedirect.com/science/article/B6WMD-45KKS7X-6W/2/16b87cde2eb3c825de73d5d7493ba37e`

[Matsuura und Yamane 1990] Matsuura, Makoto ; Yamane, Seiki: *Biology of the Vespine Wasps*. Berlin : Springer, 1990

[Nissen 1997] Nissen, Volker: *Einführung in Evolutionäre Algorithmen*. Braunschweig : Vieweg Verlag, 1997

[Page et al. 2000] Page, Bernd ; Lechler, Tim ; Claassen, Sönke: *Objektorientierte Simulation in Java mit dem Framework DESMO-J.* Norderstedt : Libri Books on Demand, 2000

[Sigaud und Wilson 2007] Sigaud, Olivier ; Wilson, Stewart W.: Learning Classifier Systems: A Survey. In: *Soft Comput.* 11 (2007), Nr. 11, S. 1065–1078

[Smith 1980] Smith, S. F.: *A Learning System Based on Genetic Algorithms.* Pittsburgh, University of Pittsburgh, Dissertation, 1980

[Sutton 2004] Sutton, Richard S.: *Reinforcement Learning FAQ : Frequently Asked Questions about Reinforcement Learning.* Februar 2004. – URL `http://www.cs.ualberta.ca/~sutton/RL-FAQ.html`

[Sutton und Barto 1998] Sutton, Richard S. ; Barto, Andrew G.: *Reinforcement learning : an introduction.* Cambridge, MA, USA : MIT Press, 1998

[Theraulaz und Bonabeau 1995] Theraulaz, Guy ; Bonabeau, Eric: Modelling the Collective Building of Complex Architectures in Social Insects with Lattice Swarms. In: *Journal of theoretical Biology* 177 (1995), S. 381–400

[Theraulaz et al. 1999] Theraulaz, Guy ; Bonabeau, Eric ; Deneubourg, Jean-Louis: The mechanisms and rules of coordinated building in social insects. In: Detrain, Claire (Hrsg.) ; Deneubourg, Jean-Louis (Hrsg.) ; Pasteels, Jacques M. (Hrsg.): *Information Processing in Social Insects.* Basel : Birkhäuser, 1999, S. 309–330

[Watkins 1989] Watkins, Christopher: *Learning from Delayed Rewards.* Cambridge, England, King's College, Dissertation, 1989

[Weicker 1999] Weicker, Karsten: Evolutionäre Algorithmen. In: Weicker, Karsten (Hrsg.): *Softcomputing - Tagungsband zum ersten Softcomputing-Treffen.* Stuttgart : Informatikverbund Stuttgart, 1999, S. 27–39

[Weicker 2002] Weicker, Karsten: *Evolutionäre Algorithmen.* Stuttgart : Teubner, 2002

[Wenzel 1989] Wenzel, John W.: Endogenous Factors, External Cues, and Eccentric Construction in *Polistes annularis* (Hymenoptera: Vespidae). In: *Journal of Insect Behavior* 2 (1989), Nr. 5, S. 679–699

[Wilson 1971] Wilson, Edward O.: *The Insect Societies.* Cambridge, MA, USA : Belknap Press of Harvard University Press, 1971

[Wilson 1995] Wilson, Stewart W.: Classifier Fitness Based on Accuracy. In: *Evolutionary Computation* 3 (1995), Nr. 2, S. 149–175

[Winston 1991] Winston, Mark L.: *The Biology of the Honey Bee.* Harvard University Press, 1991

Abbildungsverzeichnis

0.1. Wespe beim Nestbau 8

2.1. Aufhängung eines Wespennestes 15
2.2. Leeres Wespennest 16
2.3. Nester verschiedener Wespenarten im Vergleich 17
2.4. Stigmergie beim Wabenbau 18
2.5. Anzahl von Nachbarzellen 19
2.6. Bauwahrscheinlichkeit 19
2.7. Das Animat-Problem 34
2.8. Schema eines *Learning Classifier System* 35

3.1. Ergebnis einer Simulation nach sechs Bauschritten 45
3.2. Initialzustand der Simulation mit bebauten Anfangszellen 45
3.3. Vergleich zwischen Wabenstrukturen 53
3.4. Schema von *Ein-Punkt-Crossover* 57
3.5. Unbebaute sechseckige Gitterstruktur 59
3.6. Wabenstruktur aus drei Bausteintypen 60

4.1. Maximale Fitness ohne Beschränkung der Populationsgröße 67
4.2. Fitness bei Population von 100 und 100 GA-Durchläufen 68
4.3. Entwicklung der Fitness mit Crossover-Wahrscheinlichkeit 69
4.4. Entwicklung der Fitness mit Zurücklegen 70
4.5. Vergleich von Waben von 100 identischen Individuen 72
4.6. Von identischen Individuen gebaute Wabenstrukturen 72
4.7. Ringstruktur bei Simulation mit zwei Regelsätzen 73

5.1. Dreidimensionale Wabenstrukturen 80
5.2. Wabe eines Bienenschwarms 82

Tabellenverzeichnis

2.1. Zahlenwerte in Dezimal-, Standardbinär- und Gray-Codierung 28

3.1. Vom Regelwerk verwendete Zeichen und ihre Bedeutung 49
3.2. Kompletter Regelsatz `RuleSet1` 51
3.3. Kompletter Regelsatz `RuleSet2` (Teil 1) 62
3.4. Kompletter Regelsatz `RuleSet2` (Teil 2) 63

4.1. Stärkewerte des Regelsatzes in Simulation 4.2 75

A. MATLAB-Quellcode

A.1. simulationsumgebung.m

```
1  % Simulationsumgebung für den Wabenbau sozialer Insekten
2  global DX;
3  global DY;
4  global Wabengitter;
5  global Knotenzahl;
6  global zeit;
7  global Grenzen;
8  global MittelPunkte;
9  global MittelWabe;
10 global raumplot;
11 global bebautezellen;
12 global wespenanzahl;
13 global numberofruns;
14 global numberofgaruns;
15 global gadurchlauf;
16 global animation;
17 global ring;
18
19 DX = 41; % Breite des Gitters
20 DY = 41; % Höhe des Gitters
21 wespenanzahl = uint8(1); % Zahl der aktiven Agenten pro einzelner Wabe
22 MaxSteps = uint16(200); % Zahl der Schritte pro Simulationslauf
23 MaxSteps2 = uint16(400); % Zahl der Schritte für zweite Hierarchie
24 numberofruns = uint8(200); % Größe der GA-Population
25 numberofgaruns = uint8(50); % Zahl der Durchläufe des GA
26 rulesetanzahl = uint8(1); % Zahl der unterschiedlichen Regelsätze
27 animation = true; % Parameter, um einzustellen, ob der bauvorgang auch graphisch angezeigt wird
28
29 for ring=uint8(1):rulesetanzahl
30
31 for gadurchlauf=uint8(1):numberofgaruns
```

```
32 close all; % schließt alle vorherigen Fenster
33
34 for run=uint8(1):numberofruns
35
36 Wabengitter = hextop(DX,DY);
37 % Erzeugen des hexagonalen Gitters
38 % Struktur der Matrix:
39 % Zeile 1: X-Koordinate des Knotenpunkts
40 % Zeile 2: Y-Koordinate des Knotenpunkts
41 % Zeile 3: Markierung, ob Knotenpunkt Wabenmittelpunkt ist (1/0)
42 % Zeile 4: Belegung des Feldes (0 = leer, 1 = Baustein Typ 1, ...)
43
44 Knotenzahl = size(Wabengitter,2);
45
46 Position = DX + 2; % Initialisierung - erster Wabenmittelpunkt
47
48
49 % Algorithmus zur Markierung der Wabenmittelpunkte
50 % - legt eine dritte Zeile in der Matrix Wabengitter an und markiert jeden
51 % Knotenpunkt, der einem Wabenmittelpunkt entspricht, mit einer 1
52
53 for Position=DX+2:3:2*DX-1
54 % die zweite Reihe vom zweiten bis zum vorletzten Element
55 % mit horizontalem Abstand 3
56 Position2 = Position;
57 while Position2 < (Knotenzahl - DX) % oberste Zeile ausschließen
58 Wabengitter(3,Position2) = 1; % Setze Markierung für Mittelpunkt
59 Position2 = Position2 + 2 * DX;
60 end
61 Position2 = Position + DX + 2;
62 if (mod(Position2,DX) > 1) && (mod(Position2,DX) < DX)
63 while Position2 < Knotenzahl - DX
64 Wabengitter(3,Position2) = 1;
65 Position2 = Position2 + 2*DX;
66 end
67 end
68 end
69
70 MittelPunkte = find(Wabengitter(3,:));
71 % erzeugt einen Vektor mit Indizes der Wabenmittelpunkte
72 Grenzen = max(Wabengitter,[],2);
73 % ereugt einen Vektor, der die Endkoordinaten des Gitters enthält
74 MittelWabe = MittelPunkte(ceil(length(MittelPunkte) / 2));
75 % Index der annähernd mittleren Wabe (Ausgangspunkt für die Simulation)
```

```
76  mittelnachbarn = nachbarn(MittelWabe);
77  Wabengitter(4,MittelWabe) = 1; % Baustein vom Typ 1 an Mittelpunkt ge-
    setzt
78  % Wird als Anfangspunkt benötigt
79  for nachbarindex=1:6
80  Wabengitter(4,mittelnachbarn(nachbarindex)) = 1;
81  end
82  % die sechs direkten Nachbarn des Mittelpunkts werden als bebaut markiert
83
84  if (gadurchlauf == numberofgaruns)
85   % nur beim letzten Durchlauf grafisch anzeigen
86   scrsz = get(0,'ScreenSize');
87   % erzeuge figure für die graphische Anzeige
88   visual = figure('Name', ...
89  ['GA ' num2str(gadurchlauf) ' RS ' num2str(run)], ...
90  'Position',[1 scrsz(4) scrsz(3) scrsz(4)], ...
91  'NumberTitle','off');
92  raumplot = gca;
93  % verhindern, dass gezeichnetes wieder gelöscht wird:
94  hold on;
95
96  % Betrachtungswinkel einstellen
97  view(3);
98  axis(raumplot,'manual');
99  set(raumplot, 'DataAspectRatio', [1 1 1] ); %, ...
100 axis(raumplot,[-2 (Grenzen(1)+2) -2 (Grenzen(2)+2)]);
101 camlight('right');
102 camproj('perspective');
103 lighting gouraud;
104
105 % leeres Gitter anzeigen
106 plotgitter(Wabengitter, raumplot);
107 plotwabe(MittelWabe, raumplot, 8); % die Anfangswabe wird markiert
108 for nachbarindex=1:6
109  plotwabe(mittelnachbarn(nachbarindex), raumplot, 8);
110  end % die sechs unmittelbaren Nachbarzellen der Mittelwabe werden auch gesetzt
111  % verhindert die Notwendigkeit einer Einserregel
112 end
113
114 if (ring == 1) % nur für den ersten Regelsatz
115
116 if (gadurchlauf == 1) % Nur im ersten Durchlauf des GA...
117 % ...wird die durch wespenanzahl festgelegte Menge von Objekten der
118 % Klasse agent angelegt.
```

```
119   for nummer=uint8(1):wespenanzahl
120     wespe(run,nummer) = ...
121       bauagent(Wabengitter(1:2, MittelWabe)',nummer);
122  % erzeugt ein Objekt der Klasse Bauagent auf den Koordinaten
123  % der mittleren Wabe
124  end
125  global wespe; % 'workaround', um das Array, das die Agenten beinhaltet,
126  % nachträglich alsglobal zu definieren. Gilt als deprecated,
127  % funktioniert aber in der verwendeten Version von MATLAB.
128  % Eine Vorabdeklaration als global funktioniert nicht, weil das Array
129  % dann automatisch als numerisch deklariert wird.
130  else
131  % dieser else-block wird nur in späteren Durchläufen des GA erreicht.
132  % wespe ist dann schon als global deklariert und kann benutzt werden.
133   for nummer=uint8(1):wespenanzahl
134   wespe(run,nummer).BuildCounter = 0;
135   wespe(run,nummer).Position = Wabengitter(1:2, MittelWabe)';
136  end
137  end
138  else
139  % für den zweiten Durchlauf in der Hierarchie mit zwei
140  % hintereinander zu benutzenden Regelsätzen
141  for nummer=uint8(1):wespenanzahl
142   wespe(run,nummer).classifierset = ...
143     bestclassifierset1;
144   wespe(run,nummer).numberofclassifiers = ...
145     length(wespe(run,nummer).classifierset);
146   wespe(run,nummer).classifierfitnessstring = ...
147     strcat(wespe(run,nummer).classifierset.fitnessstring);
148   wespe(run,nummer).BuildCounter = 0;
149   wespe(run,nummer).Position = Wabengitter(1:2, MittelWabe)';
150  end
151  end
152
153
154
155   % Simulationsstart
156   %
157   % Update der Liste der Indizes aller bebauten Zellen
158   bebautezellen = find(Wabengitter(4,:));
159   grafik = gcf;
160
161   % Simulationsschleife zur äußeren Steuerung der Simulation
162   % - hier werden die Agenten aktivert und bewegt
```

```
163   for zeit=uint16(1):MaxSteps
164     for wespennummer=uint8(1):wespenanzahl
165       wespe(run,wespennummer).movebuild();
166     end
167   end
168
169   % Zweiter hierarchischer Regelsatz
170   if ( ring == 2 )
171     for wespennummer=uint8(1):wespenanzahl
172 wespe(run,wespennummer) ...
173   .updateclassifierset(wespe(run,wespennummer).RuleSet2);
174       % Zweites regelwerk wird hier aktiviert
175     end
176     % zweite Zeitschleife für zweite Hierarchieebene
177     for zeit=MaxSteps+1:MaxSteps2
178       for wespennummer=uint8(1):wespenanzahl
179         wespe(run,wespennummer).movebuild();
180       end
181     end
182   end
183
184   % Berechnung des globalen Fitnesswerts der gebauten Wabe
185 [fitnesswert,idealwert,aussenlaenge,wabenanzahl,quotient] ...
186   = fitness(Wabengitter);
187
188   % Legt einen Vektor overallfitness an, in dem für den Phänotyp
189   % jedes Regelsatzes seine Overall-Fitness abgespeichert wird.
190   % Dient als Basis für den GA.
191   overallfitness(run) = quotient;
192 end
193
194 strengthga(overallfitness);
195 % Aufruf des GA zur Erzeugung neuer Regelsätze mit anderen Fitnesswerten
196
197 numberofruns = size(wespe,1);
198 % Update der Anzahl der Durchläufe
199 % - wenn neue Individuen (Regelsätze) dazukommen,
200 % müssen diese auch berücksichtigt werden
201
202 [maxwert, maxindex] = max(overallfitness);
203 [minwert, minindex] = min(overallfitness);
204 % ermittelt den Regelsatz mit bester bzw. niedrigster Fitness
205
206 maxvektor(gadurchlauf,1) = maxwert;
```

```
207 maxvektor(gadurchlauf,2) = maxindex;
208 maxvektor(gadurchlauf,3) = minwert;
209 maxvektor(gadurchlauf,4) = minindex;
210 maxvektor(gadurchlauf,5) = mean(overallfitness);
211 end
212
213 if (ring == 1)
214
215  bestclassifierset1 = ...
216   wespe(uint16(maxvektor(gadurchlauf,2))).classifierset;
217  % besten Regelsatz zur Wiederverwendung kopieren
218
219  bestwespe1 = wespe(uint16(maxvektor(gadurchlauf,2)));
220  bestwespe1.BuildCounter = 0;
221  bestwespe1.Position = Wabengitter(1:2, MittelWabe)';
222 end
223 end
```

A.2. bauagent.m

```
1  classdef bauagent < handle
2  %BAUAGENT Diese Klasse definiert die Eigenschaften des einzelnen Agenten
3  % Hier werden die Eigenschaften festgelegt, die den einzelnen Agenten
4  % ausmachen, der für den Bau der Wabenstruktur zuständig ist.
5  % Für jeden einzelnen Agenten wird ein Objekt dieser Klasse erzeugt.
6
7  properties
8  Position % Vektor, welcher die Position des Agenten in
9  % X-Y(-Z)-Koordinaten enthält. Je nach Umgebung
10 % zwei- oder dreidimensional.
11 Ausrichtung % Blickrichtung des Agenten im Gradmaß (0 bis 360)
12 % Null entspricht Blick waagerecht nach rechts
13 RuleSet1
14 % Enthält die Ausgangsregeln, nach denen gebaut oder nicht
15 % gebaut wird. Hier werden nur Regeln festgelegt, mit
16 % denen begonnen wird, danach werden die Regeln in das
17 % Array classifierset kopiert und nur noch von dort
18 % ausgelesen und auch durch den GA verändert.
19 % Könnte auch aus Datei gelesen werden...
20 RuleSet2 % Regelsatz der zweiten Hierarchieebene
21 BuildCounter % Zählt die Anzahl der gesetzten Bausteine.
22 Abstand % Ein Abstandsvektor, der die absoluten Abstandswerte zu
23 % allen Zellenmittelpunkten enthält
```

```
24 Speed = 1; % relative Geschwindigkeit, mit der sich der Agent bewegt
25
26 Markierung % graphisches Objekt zur Positionsmarkierung
27 Index % enthält die laufende Nummer des Agenten
28 Farbe % die Farbe zur Markierung
29 classifierset % ein Array, in dem die Objekte vom Typ 'classifier'
30 % abgelegt werden
31 numberofclassifiers % Anzahl der vorhandenen Regeln
32 matchset % Array in dem die Indizes aller passenden Klassifikatoren für einen
   Zeitschritt abgelegt werden
33 % - wird nach jedem Zeitschritt gelöscht
34 bucket % enthält den Index-Wert des letzten benutzten Klassifikators
35 % wird für den bucket-brigade-Algorithmus benötigt
36 classifierfitnessstring % enthält die Fitnessstrings aller Regeln im Gray
   code hintereinander
37 end
38
39 methods
40
41 % Konstruktor
42 function obj = bauagent(InitialPosition, indexNummer)
43 global Grenzen;
44 global ring;
45
46 obj.Ausrichtung = 0;
47
48 obj.RuleSet1 = ['BBBBBB1'; 'BBBBB01'; '0BBBBB1'; ...
49                 'B0BBBB1'; 'BB0BBB1'; 'BBB0BB1'; ...
50                 'BBBB0B1'; 'BBBB001'; '0BBBB01'; ...
51                 '00BBBB1'; 'B00BBB1'; 'BB00BB1'; ...
52                 'BBB00B1'; 'BBB0001'; '0BBB001'; ...
53                 '00BBB01'; '000BBB1'; 'B000BB1'; ...
54                 'BB000B1'; 'BB00001'; '0BB0001'; ...
55       '00BB001'; '000BB01'; '0000BB1'; 'B0000B1'; ...
56                 'B000000'; '0B00000'; '00B0000'; ...
57                 '000B000'; '0000B00'; '00000B0'];
58
59 obj.RuleSet2 = ['1100002'; '0110002'; '0011002'; ...
60                 '0001102'; '0000112'; '1000012'; ...
61                 '2110002'; '0211002'; '0021102'; ...
62                 '0002112'; '1000212'; '1100022'; ...
63                 '1120002'; '0112002'; '0011202'; ...
64                 '0001122'; '2000112'; '1200012'; ...
65                 '2112002'; '0211202'; '0021122'; ...
```

```
66                  '2002112'; '1200212'; '1120022'; ...
67                  '2120002'; '0212002'; '0021202'; ...
68                  '0002122'; '2000212'; '1200022'; ...
69                  '1200002'; '0120002'; '0012002'; ...
70                  '0001202'; '0000122'; '2000012'; ...
71                  '2100002'; '0210002'; '0021002'; ...
72                  '0002102'; '0000212'; '1000022'];
73 if (ring == 1)
74 obj.updateclassifierset(obj.RuleSet1);
75 elseif (ring == 2)
76 obj.updateclassifierset(obj.RuleSet2);
77 end
78
79
80 if nargin > 0
81 obj.Index = indexNummer; % übergibt den Index aus dem Vektor
82 % von Agenten als Eigenschaft
83 else
84 obj.Index = 0;
85 end
86
87 if (mod(obj.Index, 7) == 1)
88 obj.Farbe = 'blue';
89 elseif (mod(obj.Index, 7) == 2)
90 obj.Farbe = 'cyan';
91 elseif (mod(obj.Index, 7) == 3)
92 obj.Farbe = 'green';
93 elseif (mod(obj.Index, 7) == 4)
94 obj.Farbe = 'white';
95 elseif (mod(obj.Index, 7) == 5)
96 obj.Farbe = 'black';
97 elseif (mod(obj.Index, 7) == 6)
98 obj.Farbe = 'yellow';
99 else
100 obj.Farbe = 'magenta';
101 end
102
103 obj.BuildCounter = 0; % Anzahl der gebauten Bausteine auf Null
104
105 if nargin > 0
106 obj.Position = InitialPosition;
107 else
108 % obj.Position = rand(1,2);
109 % obj.Position(1,1) = obj.Position(1,1) * Grenzen(1); obj.Position(1,1) = 0;
```

```
110 obj.Position(1,2) = 0;
111 end
112
113 obj.Markierung = line(obj.Position(1), obj.Position(2), ...
114 1.1, 'Marker','.', 'MarkerFaceColor',obj.Farbe, ...
115 'Color',obj.Farbe );
116 % aktuelle Position des Agenten wird graphisch
117 % markiert
118 % set(obj.Markierung, 'XDataSource','agent.Position(1)');
119 % set(obj.Markierung, 'YDataSource','agent.Position(2)');
120 % Verknüpfung des Grafikobjekts mit der Position
121 end
122
123 % function obj = bauagent()
124 function obj = movebuild(agent)
125 % Diese Funktion wird einmal pro Zeitschritt für jeden Agenten aufgerufen
126 % Die Umgebung wird verarbeitet und nach den Regeln des Agenten
127 % entschieden, ob der Agent an der aktuellen Position eine Wabe baut
128 % oder sich weiterbewegt.
129 global Wabengitter;
130 global Knotenzahl;
131 global Grenzen;
132 global raumplot;
133 global bebautezellen;
134 global animation;
135 global gadurchlauf;
136 global numberofgaruns;
137
138 % matchset muss gelöscht werden (empty matrix)
139 agent.matchset = [];
140
141 % dreht den Agenten in eine zufällige Richtung
142 agent.Ausrichtung = mod(agent.Ausrichtung + rand*360, 360);
143
144 % Index der Zelle, auf dessen Fläche sich der Agent befindet
145 Zelle = agent.indexPosition;
146 % Erzeugt in Nachbarn eine Liste der Indizes
147 % der den Agenten unmittelbar umgebenden Zellen
148 Nachbarn = agent.nachbarn(Zelle);
149
150 for nachbarZelle=1:6
151 builtFlag = 0;
152 validFlag = 1; % Initialisierung
153
```

```
154 bauPosition = Nachbarn(nachbarZelle);
155 % setze potentielle Position für neuen Baustein auf einen
156 % der sechs Nachbarzellen der aktuellen Position
157
158 if ( (bauPosition < 1) || (bauPosition > Knotenzahl) || ...
159  % Grenzen des Gitters
160  (Wabengitter(4,bauPosition) ~= 0) || ...
161  % Bebauung der Zelle
162  (Wabengitter(3,bauPosition) == 0) )
163  % und ist die Wabe noch im Gitter?
164  validFlag = 0;
165
166 end
167
168 % erzeugt Liste der unmittelbaren Nachbarn der potentiellen Bauposition
169 bauNachbarn = agent.nachbarn(bauPosition);
170 % jeder Klassifikator wird auf Validität geprüft
171 for ruleNumber=1:agent.numberofclassifiers
172  % wenn sich die Position außerhalb des
173  % Gitters befindet oder schon bebaut ist,
174  % wird die nächste Position geprüft
175  if (validFlag == 0)
176  break;
177 end
178
179 validFlag = 1;
180 % jede Position der Regel wrd einzeln geprüft
181 for ruleCellNumber=1:6
182 if ( (validFlag == 0) || ...
183 (bauNachbarn(ruleCellNumber) > Knotenzahl) || ...
184 (bauNachbarn(ruleCellNumber) < 1) )
185 break;
186 % Wenn schon die letzte Stelle der Regel ungültig war,
187 % braucht diese nicht mehr getestet zu werden.
188 % Wenn die Position außerhalb des Gitters liegt, auch nicht
189 end
190
191 if ...
192 strcmp(agent.classifierset(ruleNumber). ...
193         detector(ruleCellNumber),'B')
194 if ( Wabengitter(4,bauNachbarn(ruleCellNumber)) == 1 ...
195   || Wabengitter(4,bauNachbarn(ruleCellNumber)) == 2 ...
196   || Wabengitter(4,bauNachbarn(ruleCellNumber)) == 3 ...
197   || Wabengitter(4,bauNachbarn(ruleCellNumber)) == 4 ...
```

```
198     || Wabengitter(4,bauNachbarn(ruleCellNumber)) == 5  ...
199     || Wabengitter(4,bauNachbarn(ruleCellNumber)) == 6  ...
200     || Wabengitter(4,bauNachbarn(ruleCellNumber)) == 7  )
201     validFlag = 1;
202     % wenn die Regel an dieser Stelle 'B' enthält,
203     % wird validFlag 'true' gesetzt, falls ein
204     % beliebiger Baustein (1-7) vorhanden ist.
205   else
206    validFlag = 0;
207   end
208   elseif ...
209   strcmp(agent.classifierset(ruleNumber). ...
210           detector(ruleCellNumber),'*')
211   validFlag = 1;
212   % enthält die Regel an der betreffenden Stelle ein '*'
213   % so ist der Zustand der Zelle egal.
214   else
215   validFlag = ...
216   strcmp(num2str(Wabengitter(4,bauNachbarn(ruleCellNumber))), ...
217    agent.classifierset(ruleNumber).detector(ruleCellNumber));
218   % in allen anderen Fällen (genaue
219   % Spezifikation in der Regel) muss
220   % der Zelleninhalt exakt mit der
221   % Regelposition übereinstimmen.
222   end
223
224   end
225
226   if (validFlag == 1)
227
228   agent.classifierset(ruleNumber).bid = ...
229   % Rückkonvertierung von Gray
230   gray2bin(agent.classifierset(ruleNumber). ...
231    strength,'fsk',256) ...
232    * log2(agent.classifierset(ruleNumber).specificity + 1);
233   % bid-Wert wird berechnet:
234   % bid(r) = strength(r) * log2(specificity(r))
235
236   agent.matchset(length(agent.matchset)+1) = ruleNumber;
237   % wenn die Regel des Klassifikators gültig ist,
238   % wird seine Indexnummer in matchset übernommen
239
240   end
241
```

```
242 validFlag = 1; % sonst wird die nächste Regel nicht angewendet
243 end
244 % an dieser Position sind alle Regeln ausgewertet und
245 % matchset sollte alle passenden Regeln enthalten
246
247
248 overallbid = 0;
249 cumulativebid = 0;
250 chosenrule = 0;
251
252 if ~isempty(agent.matchset)
253 % bei leerem matchset muss nichts ausgewählt werden
254 for clsf=1:length(agent.matchset)
255 overallbid = ...
256  overallbid + agent.classifierset(agent.matchset(clsf)).bid;
257 end
258
259 for clsf=1:length(agent.matchset)
260 normalbid = ...
261  agent.classifierset(agent.matchset(clsf)).bid / ...
262   overallbid;
263 % Normalisierung des bid-Wertes auf Wert zwischen 0 und 1
264 % (Anteil an der Summe aller bid-Werte aller Regeln)
265 agent.classifierset(agent.matchset(clsf)).minfit = ...
266  cumulativebid;
267 cumulativebid = cumulativebid + normalbid;
268 agent.classifierset(agent.matchset(clsf)).maxfit = ...
269  cumulativebid;
270 % Unter- und Obergrenzen werden gesetzt
271 end
272
273 decision = rand;
274 for clsf=1:length(agent.matchset)
275 if ...
276 (decision>=agent.classifierset(agent.matchset(clsf)).minfit...
277 &&decision<=agent.classifierset(agent.matchset(clsf)).maxfit)
278 % hier wird die zu verwendende Regel ausgewählt und ihr Index in chosenrule abge-
    speichert
279 % hack um zu verhindern, dass jede regel
280 % unbedingt ausgewählt wird
281 if ( (rand * 255) <= agent.classifierset(agent.matchset(clsf)).strength
282      chosenrule = agent.matchset(clsf);
283 end
284 break;
```

```
285 end
286 end
287 end
288
289 % Wenn eine Regel (chosenrule) ausgewählt wurde, muss nach
290 % dieser Regel gebaut werden:
291 if (chosenrule)
292 Wabengitter(4,bauPosition) = ...
293   agent.classifierset(chosenrule).effector;
294 % *hier wird gebaut*
295
296 agent.bucket = chosenrule;
297 % benutzte Regel wird für BBA
298 % zwischengespeichert
299
300 agent.BuildCounter = agent.BuildCounter + 1;
301 % Counter erhöhen
302
303 builtFlag = 1; % zeigt an, dass in diesem Zeitschritt gebaut wurde
304
305 if (gadurchlauf == numberofgaruns)
306 plotwabe(bauPosition, raumplot, Wabengitter(4,bauPosition));
307 % die gebaute Wabe wird graphisch markiert
308 % Parameter:
309 % bauPosition = Index der zu bebauenden
310 % Zelle
311 % raumplot = Handle des zu benutzenden
312 % Achsenobjekts
313 % Wabengitter(...) = Typ des Bausteins
314 % (definiert die Farbe)
315 end
316 % Update der Liste der Indizes aller bebauten Zellen
317 bebautezellen = find(Wabengitter(4,:));
318 end
319
320 if true(builtFlag)
321 builtFlag = 0;
322 break;
323 % jeder Agent soll höchstens einmal pro Zeitschritt bauen
324 end
325 end
326
327 % Bewegung in x-Richtung
328 agent.Position(1) = ...
```

```
329   agent.Position(1) + cosd(agent.Ausrichtung * agent.Speed);
330  % Bewegung in y-Richtung
331  agent.Position(2) = ...
332   agent.Position(2) + sind(agent.Ausrichtung * agent.Speed);
333
334  zelle = agent.indexPosition;
335  % wenn Position unbebaut ist...
336  if ( (Wabengitter(4,zelle) == 0) || ...
337  (agent.Position(1) > Grenzen(1)) || ...
338  (agent.Position(1) < 0) ||...
339  (agent.Position(2) > Grenzen(2)) || ...
340  (agent.Position(2) < 0) )
341  % .. oder außerhalb des Gitters liegt
342  % ...dann mache Positionsänderung rückgängig
343  agent.Position(1) = agent.Position(1) + ...
344  cosd(mod((agent.Ausrichtung+180),360)*agent.Speed);
345  agent.Position(2) = agent.Position(2) + ...
346  sind(mod((agent.Ausrichtung+180),360)*agent.Speed);
347  end
348
349  if (gadurchlauf == numberofgaruns)
350  agent.Markierung=line(agent.Position(1),agent.Position(2), ...
351   1.1, 'Marker','.', 'MarkerFaceColor',agent.Farbe, ...
352   'Color',agent.Farbe, 'Parent',raumplot );
353   % neue Markierung des Agenten wird gesetzt
354  end
355
356  if (animation)
357   drawnow;
358  end
359  % zeigt die animierte Bausequenz, wenn animation-flag gesetzt
360  end
361
362  function indexnummer = indexPosition(agent)
363  % Gibt für jeden Agenten den Index des Mittelpunkts derjenigen
364  % Wabe zurück, auf deren Fläche sich der Agent nach seinen
365  % Koordinaten befindet
366  global Wabengitter;
367  global MittelPunkte;
368
369  agent.Abstand = ...
370   [abs(Wabengitter(1,MittelPunkte) - agent.Position(1)); ...
371    abs(Wabengitter(2,MittelPunkte) - agent.Position(2))];
372  kandidatenX = find(agent.Abstand(1,:) <= 1);
```

```
373 kandidatenY = find(agent.Abstand(2,:) <= 1);
374
375 indexnummer = 0;
376 for kandX=1:length(kandidatenX)
377 for kandY=1:length(kandidatenY)
378 if (kandidatenX(kandX) == kandidatenY(kandY))
379 indexnummer = MittelPunkte(kandidatenX(kandX));
380 break;
381 end
382 end
383 if (indexnummer ~= 0)
384 break;
385 end
386 end
387
388 % Der folgede Code ist zu langsam und wird nur verwendet,
389 % wenn der andere zu keinem Ergebnis kommt
390 %
391 if indexnummer == 0
392 agent.Abstand = sqrt((Wabengitter(1,MittelPunkte) - ...
393  (agent.Position(1,1))).^2 + (Wabengitter(2,MittelPunkte) ...
394  - (agent.Position(1,2))).^2 );
395 % erzeugt einen Vektor mit den euklidischen Abständen des Agenten
396 % von allen Wabenmittelpunkten:
397 % D(X,Y) = sqrt(x1 - y1)^2 + (x2 - y2)^2
398
399 [abstand, I] = min(agent.Abstand);
400 % Gibt den Abstand zum nächstgelegenen
401 % Wabenmittelpunkt (abstand) und den dazugehörigen
402 % Index im Vektor Abstand (I) zurück
403 indexnummer = MittelPunkte(I);
404 % Der nächste Mittelpunkt ist der
405 % mit Index I im Vektor Abstand, also ist sein
406 % Index im Array Wabengitter der Wert an Stelle i
407 % im Vektor Abstand
408 end
409 end
410
411 function updatefitness(agent)
412 % Kopiert den Gesamt-Fitnesstring in die Stärkewerte der
413 % einzelnen Regeln.
414 for regelindex=1:agent.numberofclassifiers
415 agent.classifierset(regelindex).fitnessstring = ...
416 agent.classifierfitnessstring(((regelindex-1)*8)+1: ...
```

```
417   ((regelindex-1)*8)+8);
418  graystrength = ...
419   bin2dec(agent.classifierset(regelindex).fitnessstring);
420  % Konvertierung von Binärstring in Dezimalzahl(Graycodiert)
421  agent.classifierset(regelindex).strength = ...
422   gray2bin(graystrength,'fsk',256);
423  % konvertiert von Gray nach Standard-Binär und speichert im strength-Feld
424  end
425  end
426
427  function updateclassifierset(agent, ruleset)
428  % Erzeugt aus den Regeldefinitionen in ruleset einen neuen
429  % Regelsatz für den Agenten und speichert diesen in
430  % classifierset ab. Wird beim Erzeugen des Agenten aufgerufen
431  % und danach bei jedem Wechsel in eine neue Hierarchieebene
432  % zum Wechsel zum neuen Regelsatz.
433  for rule=1:size(ruleset,1)
434  classifiers(rule) = ...
435   classifier(ruleset(rule,:),uint8(rand*255));
436  end
437  % Erzeugung des Regelsatzes -
438  % Für jede Regel eine Zufallszahl zwischen
439  % 0 und 255, um genau mit 8-bit Gray code arbeiten zu können
440
441  agent.classifierset = classifiers;
442  agent.numberofclassifiers = length(agent.classifierset);
443  agent.classifierfitnessstring = ...
444   strcat(agent.classifierset.fitnessstring);
445  end
446
447  end
448
449  end
```

A.3. classifier.m

```
1  classdef classifier
2  %CLASSIFIER Definiert die Struktur eines Klassifikators
3  % Klassendefinition für die Klassifikatoren
4  % die das Verhalten der Agenten steuern
5  %
6  % Ein Klassifikator in LCS besteht aus einer Regel, die
7  % einen Detektor (hier als String implementiert) und
```

```
8  % einen Effektor miteinander verknüpft.
9  % Außerdem werden noch verschiedene zusätzliche Parameter
10 % wie Stärke(Fitness) und Spezifizität der Regel als
11 % Bestandteil des Klassifikators definiert.
12
13 properties
14 detector % enthält den String zur Definition des Detektors zur
15 % Verarbeitung der Umwelt
16 % B = beliebiger Baustein an Position
17 % * = don't care
18 % 0,1,2,3,... = bestimmter (bzw. kein) Baustein an Position
19
20 effector % Definiert die Aktion des Agenten - in diesem Modell also
21 % den Typ des zu bauenden Bausteins als Zahlenwert (1,2,3,
22 % ...) bzw. auch eben KEINEN Baustein zu setzen (0)
23
24 strength % skalarer Wert zur Bestimmung der Stärke oder Fitness
25 % - normiert auf 0 ≤ Wert ≤ 255
26 % wichtig wegen interner Codierung in 8bit Gray code
27 % für GA
28
29 fitnessstring % Fitnesswert in 8bit Gray code als Textstring z.B. '00110110'
30
31 specificity % Wert zur Bestimmung der Genauigkeit der Regel
32 % entspricht der Zahl der Elemente des Detektor-Strings
33 % die nicht "*don't-care entsprechen
34
35 age % wird in jedem Zeitschritt erhöht, bei Benutzung des Klassifikators erniedrigt
36 % nach Modell von Holland(1978)
37
38 frequency % Häufigkeit der Aktivierung des Klassifikators
39 % Wird bei jeder Aktivierung um 1 erhöht
40
41 attenuation % Dämpfungskoeffizient
42 % wird bei der Berechnung des Feedback aus der Umgebung
43 % miteinbezogen
44
45 detectorlength = 6; % Länge des Detektors (im zweidimensionalen Modell 6,
46 % im dreidimensionalen 20
47
48 bid % skalarer Wert, der aus strength und specificity in jedem Zeitschritt neu be-
   rechnet wird
49 % wird nur berechnet, wenn Klassifikator im Matchset auftaucht
50 % Je höher der bid-Wert des Klassifikators, desto höher seine
```

```
51 % Chance auf Ausführung. Pro Zeitschritt wird nur ein
52 % Klassifikator angewendet.
53
54 minfit % Untergrenze für Roulette-Wheel-Selection
55 maxfit % Obergrenze für Roulette-Wheel-Selection
56
57
58 end
59
60 methods
61 function obj = classifier( rulestring, fitness ) % Konstruktor
62 obj.detector = rulestring(1:obj.detectorlength);
63 % kopiert die relevanten Teile des Strings in den Detektor
64
65 % Konvertierung in 8bit Gray code
66 obj.strength = bin2gray(fitness,'fsk',256);
67 % Darstellung der Fitness als Bitstring für GA
68 obj.fitnessstring = dec2bin(obj.strength,8);
69
70 obj.specificity = 0;
71 % Spezifizität der Regel wird berechnet
72 for pos=1:obj.detectorlength
73 if (~(strcmp(obj.detector(pos), '*')))
74 obj.specificity = obj.specificity + 1;
75 end
76 end
77
78 obj.effector = str2double(rulestring(obj.detectorlength + 1));
79 % letzte Position des RuleString ist der Effektor
80
81 % Initialisierung von age, frequency und attenuation
82 % Anfangswerte bei allen gleich Null
83 obj.age = 0;
84 obj.frequency = 0;
85 obj.attenuation = 0;
86
87 end
88
89 end
90
91 end
```

A.4. fitness.m

```
1  function [fitnesswert,idealwert,aussenlaenge, ...
2            wabenanzahl,fitnessquotient] = fitness(zustandsraum)
3  %FITNESS Berechnet den Fitnesswert für eine beliebige Wabenstruktur
4  % Diese Version berechnet die Fitness anhand der
5  % Kompaktheit der Struktur: man berechnet die
6  % Gesamtlänge der Außenkanten und teilt sie durch die
7  % Anzahl der bebauten Zellen. Je höher dieser Wert, desto
8  % kompakter die Struktur - Idealfall: Kreisform
9  %
10 % Löcher in der Struktur führen zur Abwertung der Fitness
11 %
12 % Von der Funktion werden folgende Werte zurückgegeben:
13 % - berechneter Fitnesswert für die Struktur
14 % - Idealwert eines Kreises mit gleicher Fläche
15 % - Gesamtlänge der Außenwand
16 % - Anzahl der bebauten Waben
17 % - Quotient aus berechneter Fitness und idealer Fitness:
18 % dieser Wert wird als Fitnesswert für strengthga benutzt
19 global Knotenzahl;
20
21 zellenflaeche = ( 3 / 2 ) * sqrt( 3 );
22 bebautezellen = find(zustandsraum(4,:));
23 wabenanzahl = length(bebautezellen);
24 aussenlaenge = 0;
25 for zellindex=1:wabenanzahl
26 nachbarliste = nachbarn(bebautezellen(zellindex));
27 for zindex=1:6
28 if (nachbarliste(zindex) < 1) || ...
29  (nachbarliste(zindex) > Knotenzahl)
30 bebauung = 0;
31 else
32 bebauung = zustandsraum(4,nachbarliste(zindex));
33 end
34 if bebauung == 0
35 aussenlaenge = aussenlaenge + 1;
36 end
37 end
38 end
39
40 % Kantenlänge des Sechsecks ist 1
41 % Nimmt man als Fitnesswert das Verhältnis von Fläche A zu Umfang U der
```

```
42 % Struktur (möglichst groß), so ergibt sich der Idealwert für die Fitness
43 % bei einer Kreisform, für die folgende Gleichungen gelten:
44 % - A = r^2 · π
45 % - U = 2 · r · π
46 % - A = U / (2 · π)^2 · π = U^2 / 4 · π
47 % - A / U = U / 4 · π
48 idealwert = aussenlaenge / ( 4 * pi );
49 fitnesswert = (wabenanzahl * zellenflaeche) / aussenlaenge;
50 fitnessquotient = fitnesswert / idealwert;
51
52 end
```

A.5. EckPunkte.m

```
1  function eckpunkte = EckPunkte(MittelPunkt)
2  %ECKPUNKTE Berechnet die Eckpunkte einer Zelle
3  % Für einen gegebenen Zellmittelpunkt werden die Eckpunkte der Zelle als
4  % Vektor zurückgegeben. Reihenfolge vom linken Eckpunkt im
5  % Uhrzeigersinn.
6  global DX;
7
8  eckpunkte = [ MittelPunkt-1, ...
9   MittelPunkt+DX-1+mod(idivide(MittelPunkt,uint32(DX)),2), ...
10  MittelPunkt+DX+mod(idivide(MittelPunkt,uint32(DX)),2), ...
11  MittelPunkt+1, ...
12  MittelPunkt-DX+mod(idivide(MittelPunkt,uint32(DX)),2), ...
13  MittelPunkt-DX-1+mod(idivide(MittelPunkt,uint32(DX)),2) ];
14
15 end
```

A.6. strengthga.m

```
1  function [] = strengthga( overallfitness )
2  %STRENGTHGA Genetischer Algorithmus zur Anpassung der Fitnesswerte
3  % Dieser GA soll nach mehreren erfolgten Simulationsläufen mit dem
4  % gleichen Regelsatz aber unterschiedlichen Fitnesswerten der einzelnen
5  % Regeln die Ergebnisse der Läufe miteinander vergleichen und aufgrund
6  % der globalen Fitness der entstandenen Strukturen die einzelnen
7  % Stärkewerte der Regeln genetisch rekombinieren. Dadurch sollen neue
8  % Regelsätze erzeugt werden, die sich nur durch die Fitness der einzelnen
```

```
% Regeln von den alten Regelsätzen unterscheiden.

global wespe;
global Wabengitter;
global MittelWabe;

% variable Parameter:
probcrossover = 0.6; % Crossoverwahrscheinlichkeit
probmutation = 0.2; % Mutationswahrscheinlichkeit
elternzahl = 100; % Zahl der auszuwählenden Individuen pro GA-Durchlauf
                  % - Muss ein Vielfaches von 2 sein

% wichtig für Roulette-Wheel-Selection:
%
% Die reziproken Werte der Gesamtfitness
% zur Berechnung der zu löschenden Elemente
overallrezip = 1 ./ overallfitness;
%
% Normierung auf Gesamtsumme 1
overallrezip = overallrezip / sum(overallrezip);
%
% Normierung auf Gesamtfitness 1
overallfitness = overallfitness / sum(overallfitness);

% Es werden zwei Vektoren (maxfitness und maxrezip) erzeugt, in denen für
% jeden Regelsatz in wespe ein Wert abgespeichert ist, der der Position des
% jeweiligen Abschnitts auf dem Roulette-Wheel entspricht. Der Vektor ist
% normiert auf 1, d.h. der letzte Wert im Vektor beträgt 1.0.
cumulfitness = 0;
cumulrezip = 0;
for regelsatz=1:length(overallfitness)
cumulfitness = cumulfitness + overallfitness(regelsatz);
maxfitness(regelsatz) = cumulfitness;
cumulrezip = cumulrezip + overallrezip(regelsatz);
maxrezip(regelsatz) = cumulrezip;
end

% Selektion
eltern = roulettemz(overallfitness, maxfitness);

% Rekombination
newstrings = onepointxover(eltern);

% Mutation
```

```
53 newstrings = mutation(newstrings);
54
55 % Selektion der zu löschenden bauagent-Objekte
56 % - Es wird das gleiche Prinzip zur Roulette-Wheel-Selektion angewendet wie
57 % bei der Eltern-Selektion, nur dass auf den Kehrwerten der Fitnesswerte
58 % gearbeitet wird und deshalb die Selektionswahrscheinlichkeit mit
59 % steigender Fitness abnimmt, d.h. dass weniger fitte bauagent-Objekte mit
60 % höherer Wahrscheinlichkeit gelöscht werden.
61 todelete = roulette(overallrezip, maxrezip);
62
63 % Erzeugen von zwei zusätzlichen bauagent-Objekten in wespe
64 % - nicht mehr nötig, da die Populationsgröße konstant gehalten werden soll
65 % und für die neuen Objekte alte überschrieben werden
66 % Stattdessen direktes Ersetzen der zu löschenden Objekte und Einfügen der
67 % neuen Fitnessstrings
68
69 % rulesetanzahl = size(wespe,1);
70 for elternindex=1:elternzahl
71 wespe(todelete(elternindex),1) = ...
72  bauagent(Wabengitter(1:2, MittelWabe)', 1);
73 wespe(todelete(elternindex),1).classifierfitnessstring ...
74  = newstrings(1,:);
75 wespe(todelete(elternindex),1).updatefitness;
76 end
77
78
79 function index = roulette(fitnessvektor, maxvektor)
80 % Roulette-Wheel-Selection ohne Zurücklegen - wird zur Auswahl der aus
81 % der Population zu löschenden Individuen verwendet. Jedes Individuum kann
82 % nur einmal gelöscht werden.
83 for candid=1:elternzahl
84 a = rand;
85 for ruleset=length(fitnessvektor):-1:2
86 if ( (a >= maxvektor(ruleset-1)) && ...
87  (fitnessvektor(ruleset) > 0.0) )
88 index(candid) = ruleset
89 break;
90 end
91 end
92 if ( a < maxvektor(1) )
93 index(candid) = 1
94 end
95 %
96 % Fitness des gewählten Individuums auf 0 setzen
```

```
97  fitnessvektor(index(candid)) = 0.0;
98  % neue Normierung
99  fitnessvektor = fitnessvektor / sum(fitnessvektor);
100 %
101 cumul = 0;
102 % neuen Vektor mit Maximalwerten für Roulette-Selection erzeugen
103 for regel=1:length(fitnessvektor)
104 cumul = cumul + fitnessvektor(regel);
105 maxvektor(regel) = cumul;
106 end
107 end
108 end
109
110 function index = roulettemz(fitnessvektor, maxvektor)
111 % Roulette-Wheel-Selection mit Zurücklegen - wird zur Auswahl der
112 % Eltern-Individuen verwendet. Jedes Individuum soll mehrmals ausgewählt
113 % werden können.
114 for candid=1:elternzahl
115 a = rand;
116 for ruleset=length(fitnessvektor):-1:2
117 if ( (a >= maxvektor(ruleset-1)) && ...
118  (fitnessvektor(ruleset) > 0.0) )
119 index(candid) = ruleset
120 break;
121 end
122 end
123 if ( a < maxvektor(1) )
124 index(candid) = 1
125 end
126 end
127 end
128
129 function kinder = onepointxover(eltern)
130 for xoverindex=1:uint8(elternzahl/2)
131 string1 = ...
132  wespe(eltern(xoverindex),1).classifierfitnessstring;
133 string2 = ...
134  wespe(eltern(xoverindex+1),1).classifierfitnessstring;
135 % kopiert die Fitness-Strings der ausgewählten Eltern in Graycode
136 % in zwei lokale String-Variablen
137 % Es wird der Gesamtstring (Stärke aller Regeln
138 % hintereinander) verwendet
139 xoverpoint = uint16(rand * length(string1));
140 % wählt einen zufälligen Crossoverpunkt
```

```
141
142 if (rand <= probcrossover)
143 % Nur dann Crossover, wenn Zufallszahl kleiner als
144 % Crossover-Wahrscheinlichkeit ist.
145 neustring1 = ...
146  [string1(1:xoverpoint) string2(xoverpoint+1:length(string2))];
147 neustring2 = ...
148  [string2(1:xoverpoint) string1(xoverpoint+1:length(string1))];
149 else % sonst werden die Elternstrings einfach kopiert
150 neustring1 = string1;
151 neustring2 = string2;
152 end
153
154 kinder(xoverindex,:) = neustring1;
155 kinder(xoverindex+1,:) = neustring2;
156 end
157 end
158
159 function newbitstring = mutation(bitstring)
160 for mutindex=1:uint8(elternzahl/2)
161 bitstring1 = bitstring(mutindex,:);
162 bitstring2 = bitstring(mutindex+1,:);
163
164 if (rand < probmutation)
165  % wenn Zufallszahl unter Mutationswahrscheinlichkeit
166 mutbit = uint16(ceil(rand * length(bitstring1)));
167 if strcmp(bitstring1(mutbit),'0')
168 bitstring1(mutbit) = '1';
169 end
170 if strcmp(bitstring1(mutbit),'1')
171 bitstring1(mutbit) = '0';
172 end
173 end % für bitstring1
174
175 if (rand <= probmutation)
176  % wenn Zufallszahl unter Mutationswahrscheinlichkeit
177 mutbit = uint16(ceil(rand * length(bitstring2)));
178 if strcmp(bitstring2(mutbit),'0')
179 bitstring2(mutbit) = '1';
180 end
181 if strcmp(bitstring2(mutbit),'1')
182 bitstring2(mutbit) = '0';
183 end
184 end % für bitstring2
```

```
185
186 newbitstring(mutindex,:) = bitstring1;
187 newbitstring(mutindex+1,:) = bitstring2;
188 end
189 end
190
191 end
```

A.7. nachbarn.m

```
1  function liste = nachbarn(mittelpunkt)
2  %NACHBARN Gibt für einen beliebigen Zellenindex eine Liste der Indizes der direk-
   ten
3  %Nachbarzellen zurück
4  % Die zurückgegebene Liste enthält *immer* sechs Indizes, auch wenn diese
5  % außerhalb des Gitters liegen. Ob ein Index gültig ist, muss separat
6  % geprüft werden.Die Reihenfolge ist vom oberen Nachbarn beginnend im
7  % Uhrzeigersinn.
8
9  global DX;
10
11 liste = [ mittelpunkt+2*DX, ...
12  mittelpunkt+DX+1+mod(idivide(mittelpunkt,uint32(DX)),2), ...
13  mittelpunkt-DX+1+mod(idivide(mittelpunkt,uint32(DX)),2), ...
14  mittelpunkt-2*DX, ...
15  mittelpunkt-DX-2+mod(idivide(mittelpunkt,uint32(DX)),2), ...
16  mittelpunkt+DX-2+mod(idivide(mittelpunkt,uint32(DX)),2) ];
17
18 end
```

A.8. plotgitter.m

```
1  function [] = plotgitter(zustandsraum,plotachsen)
2  %PLOTGITTER Zeichnet das leere Wabengitter
3  % Zeichnet ein Gitternetz auf den Koordinaten, die in den ersten beiden
4  % Reihen von zustandsraum definiert sind.
5  global MittelPunkte;
6  global animation;
7
8  for wabenNummer=1:length(MittelPunkte)
9      ecken = EckPunkte(MittelPunkte(wabenNummer));
10     patch(zustandsraum(1, ecken), zustandsraum(2, ecken), ...
```

```
11      'white', 'EdgeColor','black', 'Parent',plotachsen);
12  end
13
14  if (animation)
15      drawnow;
16  end   % sofort zeichnen, falls Animation an
17
18  end
```

A.9. plotwabe.m

```
1   function [] = plotwabe(wabenIndex,plotachse,farbe)
2   %PLOTWABE Markiert eine einzige Wabe graphisch als bebaut
3   % Detailed explanation goes here
4   global Wabengitter;
5   global animation;
6
7   ecken = EckPunkte(wabenIndex);
8
9   knoten = [ Wabengitter(1,ecken) Wabengitter(1,ecken); ...
10              Wabengitter(2,ecken) Wabengitter(2,ecken); ...
11              0 0 0 0 0 0 1 1 1 1 1 1 ]';
12
13  seiten = [ 1 2 3 4 5 6; ...
14             7 8 9 10 11 12; ...
15             1 2 8 7 NaN NaN;
16             2 3 9 8 NaN NaN;
17             3 4 10 9 NaN NaN;
18             4 5 11 10 NaN NaN;
19             5 6 12 11 NaN NaN;
20             6 1 7 12 NaN NaN ];
21
22  if (farbe == 1)
23  wabenFarbe = 'red';
24  elseif (farbe == 2)
25  wabenFarbe = 'green';
26  elseif (farbe == 3)
27  wabenFarbe = 'blue';
28  elseif (farbe == 4)
29  wabenFarbe = 'yellow';
30  elseif (farbe == 5)
31  wabenFarbe = 'magenta';
32  elseif (farbe == 6)
```

```
33 wabenFarbe = 'cyan';
34 else
35 wabenFarbe = 'black';
36 end
37
38 if (farbe ~= 0) % Null steht für unbebaut, also wird nichts gezeichnet
39 patch('Vertices',knoten, 'Faces',seiten, ...
40       'FaceColor',wabenFarbe, ...
41       'EdgeColor','black', 'LineWidth',1, 'Parent',plotachse );
42 end
43
44 if (animation)
45 drawnow;
46 end
47 % sofort zeichnen, falls Animation an
48
49 end
```

A.10. plotwaben.m

```
1  function [] = plotwaben(zustandsraum)
2  %PLOTWABEN Graphische Anzeige der Ergebnisse
3  % Markiert alle bebauten Waben mit einer Farbe
4  global raumplot;
5  global bebautezellen;
6  global animation;
7
8  % Vektor mit Indizes aller bebauten Zellen
9  bebautezellen = find(zustandsraum(4,:));
10
11 for wabe=1:length(bebautezellen)
12 % hier wird für jede bebaute Zelle plotwabe aufgerufen
13 plotwabe(wabe,raumplot,zustandsraum(4,wabe));
14 end
15
16 if (animation)
17 drawnow;
18 end % sofort zeichnen, falls Animation an
19
20 end
```

Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Departments Informatik einverstanden.

Hamburg, den 1. Oktober 2009

_____

Marius Zirngibl