Fachbereich 4:
Informatik

Institut für Wirtschafts- und Verwaltungsinformatik

Replikation einer Multi-Agenten-Simulationsumgebung zur Überprüfung auf Integrität und Konsistenz

Masterarbeit

zur Erlangung des Grades Master of Science
im Studiengang Wirtschaftsinformatik

vorgelegt von

Florian Zink

Erstgutachter: Prof. Dr. Klaus G. Troitzsch, Institut für Wirtschafts- und Verwaltungsinformatik, Universität Koblenz-Landau

Zweitgutachter: Dr. Michael Möhring, Institut für Wirtschafts- und Verwaltungsinformatik, Universität Koblenz-Landau

Koblenz, im Juli 2011

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden. Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

....

(Ort, Datum) (Florian Zink)

Inhaltsverzeichnis

1 Einleitung..........8
1.1 Sugarscape..........9
1.2 Das Sugarscape Szenario von Iain Weaver..........10
1.2.1 Verschmutzung..........10
1.2.2 Fortpflanzung..........10
1.2.3 Vererbung von Sugar..........11
1.2.4 Übertragung von kulturellen Informationen..........11
1.2.5 Handel betreiben..........11
1.2.6 Kredite..........11
1.2.7 Krankheitsübertragung und Immunsysteme..........12
1.2.8 Regeln für die Spielfelder..........12
1.3 Die Grundlagen und Theorien..........12
1.3.1 Hierarchie..........13
1.3.2 Nachhaltigkeit..........14
1.3.3 Lotka-Volterra..........14
1.3.4 Tragik der Allmende..........15

2 Das Simulationsmodell..........17
2.1 Das König-Modell..........17
2.2 Bedürfnisse..........18
2.2.1 Überleben (survival)..........18
2.2.2 Wohlstand (wealth)..........18
2.2.3 Neugier (curiosity)..........18
2.2.4 Vermehrung (breeding)..........18
2.2.5 Einfluss (influence)..........19
2.3 Aktionen..........19
2.3.1 Nahrung sammeln (gather)..........19
2.3.2 Bewegen (move)..........20
2.3.3 Fortpflanzen (breed)..........20
2.3.4 Koordinierung beginnen (startCoordinating)..........20
2.3.5 Koordinierung beenden (endCoordinating)..........21
2.3.6 Unterordnen (subordinate)..........22
2.3.7 Unterordnung beenden (unsubordinate)..........22
2.3.8 Ruhen (rest)..........22

3 Die NetLogo Replikation..........23
3.1 Grafische Benutzeroberfläche..........23
3.2 Der Programmcode..........27
3.2.1 Das Spielfeld..........27
3.2.2 Die Agenten..........29
3.2.3 Rest..........30
3.2.4 Move..........37
3.2.5 Breed..........41
3.2.6 Coordinate..........43
3.2.7 Endcoordinate..........48
3.2.8 Subordinate..........50
3.2.9 Unsubordinate..........53
3.2.10 Gather..........55
3.3 Wichtige Prozeduren..........59
3.3.1 To decide..........59
3.3.2 To execution..........60

3.3.3 To memory-fill....61
3.3.4 To memory-delete....62
3.3.5 To reproduce....63
3.3.6 To turtle-move....64
3.3.7 To turtle-eat....65
3.3.8 To coordinate....65
3.3.9 To collectTaxes....66
3.3.10 To endCoordinate....67
3.3.11 To subordinate....68
3.3.12 To endSubordinate....69
4 Ergebnisse....70
4.1 Freie Marktwirtschaft vs. Nachhaltigkeit....70
4.2 Hierarchie und Nachhaltigkeit....73
4.3 Hierarchie im Härtetest....77
5 Fazit....81
5.1 Ausblick....82

1 Einleitung

Die Untersuchung menschlichen Verhaltens ist eine komplexe Wissenschaft. Egal ob man dabei von einem wirtschafts- oder einem sozialwissenschaftlichen Betrachtungspunkt ausgeht, so stößt man mit seinen Methoden schnell an Grenzen. Unabhängig von der Betrachtungsweise muss man den Menschen und sein Handeln abstrahieren, da man nicht alle Variablen abdecken und nicht jede Aktion/Reaktion abbilden kann. So betrachtet man in den Wirtschaftswissenschaften den Mensch häufig als Homo Ökonomicus [Kg08]. Diese Version handelt immer rational und prüft gemäß seinem Wissensstand alle möglichen Handlungsalternativen. Nach der Bewertung dieser wählt er die für sich beste davon aus, um seinen Nutzen zu maximieren. Das Ganze geschieht ohne Einflüsse von außen, Emotionen oder moralische Bedenken. Diese Abstraktion des Menschen auf ein rein rational handelndes Wesen wird auch in Simulationen häufig verwendet.

In dieser Master-Arbeit möchte ich zunächst eine Simulation vorstellen, mit der das Verhalten von Agenten untersucht wird, die in einer generierten Welt versuchen zu überleben und dazu einige Handlungsmöglichkeiten zur Auswahl haben. Anschließend werde ich kurz die theoretischen Aspekte beleuchten, welche hier zu Grunde liegen.
Der Hauptteil meiner Arbeit ist meine Replikation einer Simulation, die von Andreas König im Jahr 2000 in Java angefertigt worden ist [Kö2000]. Ich werde hier seine Arbeit in stark verkürzter Form darstellen und anschließend auf meine eigene Entwicklung eingehen.

Im Schlussteil der Arbeit werde ich die Ergebnisse meiner Simulation mit denen von Andreas König vergleichen und die verwendeten Werkzeuge (Java und NetLogo) besprechen. Zum Abschluss werde ich in einem Fazit mein Vorhaben kurz zusammenfassen und berichten was sich umsetzen ließ, was nicht funktioniert hat und warum.

1.1 Sugarscape

Ein Beispiel für eine solche Simulation ist das Sugarscape Szenario von Epstein und Axtell aus dem Jahr 1996 [EA96]. In dieser Simulation gibt es ein vordefiniertes virtuelles Spielfeld, das 50x50 Felder groß ist. Über das ganze Spielfeld verteilt ist der Sugar. Dieses Gut wird von den Agenten benötigt, denn es ist ihre Nahrung und ohne genug Sugar sterben sie. In dem Standardmodell befindet sich im Nordosten und im Südwesten jeweils eine hohe Ansammlung von Sugar auf den Feldern, während auf dem Rest des Spielfelds eher wenig Sugar auf den Feldern vorhanden ist. Die Agenten auf dem Spielfeld können nur horizontal und vertikal um sich blicken, jedoch nicht diagonal. Zusätzlich hat jeder Agent nur eine begrenzte Sichtweite, so dass jeder nur einen gewissen Teil seiner Umgebung wahrnehmen kann. Zu Beginn einer Runde schaut jeder Agent um sich und bewertet die Felder in seiner sichtbaren Umgebung, danach bewegt er sich auf das Feld mit dem meisten Sugar. Da jeder Agent einen zufällig generierten Stoffwechsel hat, muss er sich Runde für Runde nach Feldern mit Sugar umschauen. Erreicht der Sugarwert eines Agenten null, so stirbt er. Tote Agenten werden durch neue ersetzt. Diese werden an einer zufälligen Position auf dem Spielfeld erzeugt.

Jeder Agent muss also für seine Situation (Stoffwechsel, Sichtweite, Felderqualität in seiner Umgebung, Sugarguthaben) eine rein rationale Entscheidung treffen und das beste Feld auswählen, damit er sein Sugarguthaben maximieren kann. Agenten, die es auf einen der Sugarberge (Nordosten/Südwesten) geschafft haben, führen in dieser Simulation ein relativ entspanntes Leben, so lange nicht zu viele Agenten den Weg dorthin finden. Jedoch sterben auch sehr erfolgreiche Agenten, sobald sie ihre vordefinierte Lebensspanne erreicht haben.

Mittlerweile gibt es viele abgeänderte Versionen des Sugarscape Szenarios, die von Leuten in unterschiedlichen Programmiersprachen entwickelt wurden. Ein Beispiel ist das umfangreiche Szenario vom dem Zentrum für interdisziplinäre Wissenschaft an der Universität von Leicester [UOL11]. Dieses Szenario [Wi09] wurde in NetLogo

geschrieben. Dies ist eine Multi-Agenten-Programmiersprache mit integrierter Modellierungsumgebung. Mit NetLogo ist es möglich, in relativ kurzer Zeit eine Simulation zu programmieren, für die man in anderen Sprachen wie z.B. Java oder C ein umfangreiches Wissen an Programmierung und Zeit benötigen würde. NetLogo basiert auf Java und man kann die Simulationen als Java-Applet exportieren. Man benötigt zum Betrachten im Prinzip nur einen Webbrowser. Somit laufen die Simulationen auf einer Vielzahl von verschiedenen Computersystemen. Die einzige Einführung zu NetLogo in Buchform ist im Buch Simulations for the social scientist von Nigel Gilbert und Klaus G. Troitzsch [GT05] enthalten. Ansonsten bleibt einem nur die NetLogo Website mit ihrem Hilfsmaterial [Ln11]

1.2 Das Sugarscape Szenario von Iain Weaver

Neben den schon bekannten Aktionen aus dem Standardmodell von Epstein und Axtell (umschauen, auf ein Feld bewegen, Sugar konsumieren, tote Agenten durch neue ersetzen) enthält dieses Modell [UOL11] [Wi09] eine Reihe weiterer Parameter, die sich auf die gesamte Simulation auswirken.

1.2.1 Verschmutzung

Aktiviert man diesen Parameter, so hinterlassen die Agenten eine Verschmutzung auf dem Feld, auf dem sie essen und ihr Stoffwechsel durchgeführt wird.

1.2.2 Fortpflanzung

Wenn diese Option aktiviert ist, können sich Agenten fortpflanzen. Dabei schaut sich ein Agent seine Nachbarn an und überprüft das Geschlecht. Wenn einer der Nachbarn fruchtbar ist, das gegensätzliche Geschlecht besitzt und an ein freies Feld angrenzt, so können die beiden Agenten gemeinsam ein Kind erzeugen. Dazu wird

von beiden Agenten die Hälfte ihres Sugarwertes abgezogen und ein neuer Agent auf dem freien Nachbarfeld erzeugt.

1.2.3 Vererbung von Sugar

Bei Aktivierung dieser Option wird der gesamte Sugar, den ein Agent besitzt, bei seinem Tod gleichmäßig unter seinen Kindern verteilt.

1.2.4 Übertragung von kulturellen Informationen

Dieser Parameter legt fest, ob Agenten untereinander kulturelle Informationen austauschen können. Wenn aktiviert, teilen benachbarte Agenten sich untereinander ihre kulturellen Informationen mit.

1.2.5 Handel betreiben

In diesem Szenario gibt es zwei Ressourcen auf dem Spielfeld: Sugar und Spice. Das Spielfeld ist so aufgebaut, dass die beiden Ressourcen in einer Region nicht in großen Massen vorhanden sind. Dies zwingt die Agenten ständig zu wandern. Da aber einige Agenten sterben, bevor sie in einer Region angekommen sind, die genügend von der zweiten Ressource beherbergt, können die Agenten untereinander Handel betreiben. Dabei prüfen die Agenten, was ihr Nachbar handeln möchte. Stimmen Angebot und Nachfrage, so wird das entsprechende Gut gehandelt. Es kann so oft gehandelt werden, wie Nachbarn in der Umgebung die gefragte Ressource handeln wollen.

1.2.6 Kredite

Mit Hilfe dieses Parameters ist es den Agenten möglich, sich Sugar von anderen Agenten zu leihen. Dies ist nötig, da in der Simulation oft Agenten, die zu alt sind um sich fortzupflanzen, die Mehrheit des Wohlstands haben. Dadurch haben viele jüngere fruchtbare Agenten zu wenig Sugar, um sich fortzupflanzen. Diese können nun allerdings ihre Nachbarn überprüfen, ob unter ihnen ein Kreditgeber ist. Hat sich ein Kreditgeber gefunden, so wird die Summe vereinbart und nach 10

Runden erfolgt die Rückzahlung. Sollte der Agent nach 10 Runden nicht die geforderte Summe zurückzahlen können, so wird sein Sugarguthaben halbiert und an den Kreditgeber ausgezahlt. Danach wird ein neuer Betrag für den noch fehlenden Sugar berechnet.

1.2.7 Krankheitsübertragung und Immunsysteme

Bei dieser Option gibt es verschiedene Krankheiten, die auftreten können. Ist ein Agent von einer Krankheit befallen, überprüft er sein Immunsystem. Kann sein Immunsystem die aktuelle Krankheit nicht korrekt bekämpfen, so ändert der Agent es ab bis es funktioniert. Außerdem verteilen Agenten in jeder Runde eine zufällige Krankheit an ihre Nachbarn.

1.2.8 Regeln für die Spielfelder

Auch für die Felder gibt es neue Regeln neben dem Nachwachsen der Ressource. Zum einen wird die Verschmutzung, die durch die Agenten erzeugt wurde, an benachbarte Felder weitergegeben. Zum anderen wurden Jahreszeiten implementiert. Das bedeutet, dass die eine Hälfte des Spielfeldes viel länger braucht um Ressourcen nachwachsen zu lassen als die andere Hälfte. Nach einer gewissen Zeit wechselt das Ganze.

1.3 Die Grundlagen und Theorien

Bei seiner Diplomarbeit [Kö2000] ging es Andreas König vor allem darum herauszufinden, ob es in seiner Simulation zu Hierarchiebildung kommt und ob dies einen Mehrwert für die Agenten darstellt oder nicht. Der zweite Punkt der ihn interessierte, war die Nachhaltigkeit, also zu überprüfen, ob die Agenten auch in der Lage sind in größeren Dimensionen zu denken als nur für sich selbst. Zusätzlich zu diesen beiden Aspekten gehören meiner Meinung nach zwei für die Simulation interessante Gesichtspunkte erklärt: Die Lotka-Volterra Regeln und das Allmende-Dilemma. Im folgenden werde ich auf alle vier Punkte eingehen, um sie dem Leser kurz zu erklären.

1.3.1 Hierarchie

Durch sein *„Interesse an gesellschaftlichen Strukturen und der Frage nach dem Sinn bzw. Unsinn von Hierarchien."* [Kö2000, S. 5] motiviert, wollte Andreas König die Hierarchiebildung in seiner Simulation beobachten. Er implementierte in seiner Simulation die Möglichkeit für alle Agenten, sich zu einem Koordinator zu erklären oder sich einem Koordinator unterzuordnen. Sobald sich ein Agent dafür entscheidet, sich als Koordinator zur Verfügung zu stellen, ändern sich zwei Dinge für ihn. Erstens ist er ab sofort für alle anderen Agenten auf dem Spielfeld als Koordinator erkennbar und zweitens erhöht sich sein Stoffwechsel. Dies hängt damit zusammen, dass er als Koordinator für all seine untergeordneten Agenten jede Runde das beste Feld ausarbeiten muss. Hier wird der simple Ansatz - Wer viel arbeitet, benötigt auch mehr Ressourcen – verfolgt. Das ausgearbeitete Feld übermittelt er dann seinen Untergeordneten. Von allem was die Agenten konsumieren, bekommt der Koordinator dann einen gewissen Anteil. Man kann dies quasi als Steuerabgaben betrachten, welche die Untergeordneten für die Nutzung der Dienste ihres Koordinators zahlen müssen.

Um das Ganze jedoch nicht zu komplex werden zu lassen entschied sich Andreas König dazu, nur einstufige Hierarchien zuzulassen. Folglich kann jeder Untergeordnete nur einen Koordinator haben und sobald ein Agent als Koordinator tätig ist, kann er sich niemandem unterordnen.

Jeder Agent kann allerdings nach einer gewissen Zeit seine Rolle als Koordinator oder Untergeordneter beenden. Somit ist es allen Agenten möglich, in jede Rolle zu schlüpfen, je nachdem was ihm persönlich den größten Nutzen bringt.

1.3.2 Nachhaltigkeit

Die Nachhaltigkeit war für Andreas König ein weiterer Schwerpunkt, den er in seine Simulation implementieren wollte. Er schreibt: *„Ich will mit dieser Diplomarbeit eine Simulationsumgebung für sozialwissenschaftliche Studien zur Verfügung stellen, um mögliche Zusammenhänge erforschen zu können und die möglichen Auswirkungen verschiedener Verhaltensweisen deutlich zu machen. Insbesondere soll es möglich sein, nachhaltigen und nicht nachhaltigen Umgang mit natürlichen Ressourcen zu modellieren, und mögliche Folgen anhand der Simulationsergebnisse vor Augen geführt zu bekommen."* [Kö2000, S.11]. Es ist möglich, die Agenten mittels Parametrisierung dazu zu bewegen nachhaltiger zu agieren. Dieses Verhalten wiederum hat Einfluss auf die Entwicklung der Simulation. Die Agenten konsumieren nun nicht mehr den gesamten Sugar der auf einem Feld liegt, sondern lassen einen bestimmten Betrag übrig. Dies hat zur Folge, dass die Felder zwar niemals leergeräumt werden, aber die Agenten nun auch nicht mehr so viel konsumieren wie sie theoretisch könnten. Der Mangel an Nahrung wiederum hat zur Folge, dass die Agenten zum Beispiel eventuell nicht mehr so schnell den Grenzwert an Sugar erreichen um sich fortpflanzen zu können etc.. Des weiteren kann man auch an der Stellschraube des Sugarwachstums drehen und dafür sorgen, dass die Ressource z.B. schneller oder langsamer nachwächst. Die Ergebnisse, die man von den Simulationsdurchläufen erhält, sind interessant und aufschlussreich.

1.3.3 Lotka-Volterra

Die erste Lotka-Volterra Regel, die eigentlich dazu dient, die periodische Populationsschwankung in einem Räuber-Beute Szenario zu beschreiben, lässt sich auch in der Simulation von Andreas König finden. Generell besagt diese Regel, dass die Schwankungen der Räuberpopulation phasenverzögert denen der Beutepopulation folgen [Wp11]. In der Simulation von Andreas König kann man sehen, dass die Bevölkerung (= blaue Linie) nicht in Echtzeit mit dem

Nahrungsvorrat (= grüne Linie) wächst. Ist das Nahrungsaufkommen hoch, so sammeln die Agenten erst einmal genug Nahrung an und vermehren sich dann. Nach einiger Zeit übersteigt die Anzahl an Agenten die Zahl, welche das Spielfeld versorgen kann. Daraufhin bricht die Population stark ein bis sich das Nahrungsvorkommen auf den Feldern erholt. Phasenverzögert steigt dann auch wieder die Population usw.. Siehe hierzu folgende Abbildung 1

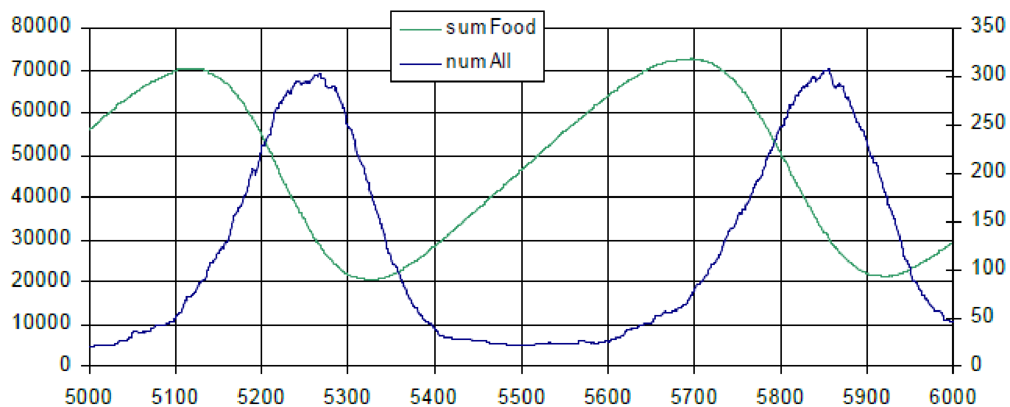

Abbildung 1: Phasenverschiebung aus [Kö2000, S.84]

1.3.4 Tragik der Allmende

Die Tragik der Allmende beschreibt das Verhalten einer Population, die eine vorhandene knappe Ressource gemeinsam nutzt. Garret Hardin schrieb hierzu einen Artikel mit dem Namen „The Tragedy of the Commons" in 1968 [Hg68]. Einige Jahre später griff Elinor Ostrom [Oe99] das Thema auf und verfasste ein ganzes Buch über die Tragik der Allmende.

In seinem Artikel von 1968 schreibt Hardin, dass man sich ein allen frei zugängliches Weideland vorstellen soll. Jeder Hirte wird versuchen, so viel Vieh wie möglich auf diesem Weideland grasen zu lassen, um so seinen Ertrag zu maximieren. Dieses Streben nach Maximierung des Wohlstands des Einzelnen hat zwei Auswirkungen. Erstens eine positive Auswirkung für den Hirten: Er bekommt für jedes Erzeugnis von jedem Tier +1 Wohlstand (z.B. Geld). Das bedeutet, für jedes weitere Tier von ihm, das er auf der Weide grasen lässt, bekommt er +1 zu seinem Wohlstand hinzuaddiert. Die zweite

Auswirkung ist negativer Natur, nämlich die Überanspruchung des Weidelands durch die Hirten. Befinden sich zu viele Tiere auf dem Weideland, ist nicht genügend Nahrung für alle da. Das Resultat ist, dass einige Tiere sterben. Da sich dieses Ereignis aber auf alle Hirten verteilt, bekommt jeder Hirte nur -1 Einheit Wohlstand abgezogen. Aus diesen Auswirkungen schließt der rationale Hirte, dass es für ihn am besten ist, wenn er einfach immer weiter Tiere auf die Weide schickt. Dieses Denken führt nun zu der Tragödie, da jeder Hirte immer mehr Tiere auf die begrenzte Weide schickt. In diesem Gedankenbeispiel wird klar, dass es hier an Regulierung mangelt. Aristoteles beschreibt die Tragik der Allmende kurz und prägnant damit, dass *„dem Gut, das der größten Zahl gemeinsam ist, die geringste Fürsorge zuteil wird. Jeder denkt hauptsächlich an sein eigenes, fast nie an das gemeinsame Interesse."*

Es gibt verschiedene Untersuchungen darüber, welche Reglementierungssysteme in welchen Szenarien besser funktionieren, seien es staatlich verordnete, privatisierte oder durch Stämme geregelte. In der Simulation von Andreas König ist dies durch die Hierarchie abgebildet. Man kann in den Simulationen die Unterschiede erkennen die entstehen, wenn man Hierarchie aktiviert bzw. deaktiviert und einige Parameter wie z.B. die Nachwuchsrate der Nahrung ändert. Siehe hierzu [Kö2000, S.81]

2 Das Simulationsmodell

Bevor ich mit der Dokumentation meiner Simulation in NetLogo beginne, möchte ich dem Leser die Arbeit von Andreas König kurz vorstellen und ihn für den technischen Teil meiner Thesis sensibilisieren.

2.1 Das König-Modell

Das von Andreas König im Jahr 2000 in Java geschriebene Modell basiert auf dem schon im ersten Kapitel erwähnten Sugarscape Szenario. Es werden ein zufälliges Spielfeld generiert und Agenten erzeugt (siehe Abbildung 2). Deren Ziel ist es so viel Nahrung zu sammeln um zu überleben und um die Bedürfnisse zu befriedigen. Andreas König hat im Ganzen fünf Bedürfnisse kreiert, die mit Hilfe von acht verschiedenen Aktionen unterschiedlich befriedigt werden können. Bevor ein Agent jedoch eine Aktion ausführt, wird mittels einer Bedürfnispyramide errechnet, welche Aktion die größte Befriedigung herbeiführt. Da aber durch Auswahl der Alternative mit der Maximalbefriedigung allein das ganze Modell zu vorhersehbar wäre und die Abbildung einer gewissen Irrationalität, die der Mensch des öfteren an den Tag legt, nicht gegeben wäre, wird hier zusätzlich noch ein Zufallsparameter in die Berechnung mit einbezogen. Dadurch wird nicht zwangsläufig die Aktion mit dem höchsten Befriedigungsfaktor gewählt.

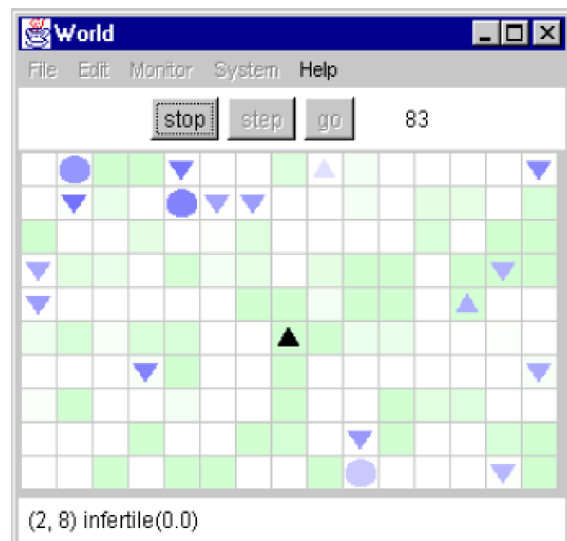

Abbildung 2: Screenshot der König-Simulation aus [Kö2000, S. 42]

Die folgenden Informationen stammen aus [Kö2000, S. 17-20]

2.2 Bedürfnisse

Die fünf Bedürfnisse werden gemäß ihrer Wichtigkeit unterschiedlich bewertet. Je höher der Wert, desto wichtiger ist das Bedürfnis.

2.2.1 Überleben (survival)

Dies ist das wohl wichtigste Bedürfnis und erhält daher im König-Modell auch die höchste Gewichtung mit einem Wert von 0,5. Das Bedürfnis gilt als befriedigt, wenn der Nahrungsvorrat für eine festgelegte Anzahl von Runden ausreicht.

2.2.2 Wohlstand (wealth)

Der Reichtum eines Agenten bemisst sich an seinem Nahrungsvorrat im Vergleich zu dem Durchschnittsvorrat aller Agenten. Je höher sein eigener ist, desto besser. Allerdings kann dieses Bedürfnis nie voll befriedigt sein (also einen Wert von 1 haben). Da der Wohlstand eher ein Luxusbedürfnis darstellt und nicht zwingend zum Überleben benötigt wird, ist es mit einem Faktor von 0,1 gewichtet.

2.2.3 Neugier (curiosity)

Dieses Bedürfnis spiegelt den menschlichen Entdeckungsdrang wider. Die Agenten erkunden gerne und stoßen deshalb in unbekanntes Territorium vor, um dieses Bedürfnis zu befriedigen. Das Bedürfnis ist ebenfalls nicht so wichtig wie das Überleben und wird daher mit einem Faktor von 0,1 gewichtet.

2.2.4 Vermehrung (breeding)

Um sich zu vermehren, benötigt ein Agent keinen Partner. Er erzeugt selbst einen Nachkommen. Dennoch unterliegt das Bedürfnis, sich zu vermehren, einigen Restriktionen in der Simulation.

1. Der Agent muss ein gewisses Alter (Pubertät) erreichen, vorher kann er sich nicht vermehren.
2. Der Agent darf nicht mehr als 10 Kinder haben
3. Der Agent muss genügend Nahrung haben um ein Kind erzeugen zu können

Sind alle Vorgaben erfüllt, so kann ein Agent einen Nachkommen erzeugen. Er stattet diesen mit einem gewissen Grundsatz an Nahrung aus. Danach handelt der neu erzeugte Agent selbständig. Der Faktor der Gewichtung liegt hier bei 0,1.

2.2.5 Einfluss (influence)

Dieses Bedürfnis wird dadurch befriedigt, dass ein Agent die Rolle eines Koordinators einnimmt. Es wird dann die Anzahl seiner untergeordneten Agenten mit der durchschnittlichen Anzahl aller anderen Koordinatoren verglichen. Genau wie bei dem Bedürfnis Wohlstand kann auch hier niemals eine komplette Befriedigung erreicht werden. Der Faktor der Gewichtung beträgt hier 0,2 und ist somit höher als Wohlstand, Neugier und Vermehrung.

2.3 Aktionen

Jeder Agent hat die Möglichkeit, eine von acht Aktionen pro Runde auszuführen. Welche er am Ende ausführt, hängt von der Befriedigung ab, die sie ihm bringt und auch vom Zufall.

2.3.1 Nahrung sammeln (gather)

Jeder Agent, der sich auf einem Feld befindet auf dem es Sugar gibt, kann diesen sammeln. Es wird jedoch nur ein bestimmter Betrag gesammelt. Dieser ist geregelt durch ein Maximum an Sugar, den ein Agent pro Runde aufnehmen kann. Ist auf dem Feld mehr Sugar vorhanden als das Maximum an Sugar, den ein Agent pro Runde sammeln kann, so nimmt er sich nur den Maximalbetrag, andernfalls sammelt er den kompletten Sugar des Feldes ein. Das Sammelverhalten kann durch Parameter beeinflusst werden, so kann z.B. zwecks Nachhaltigkeit das Ganze so konfiguriert werden, dass jeder Agent einen gewissen Teil Sugar auf dem Feld zurücklässt und niemals alles leerräumt.
Falls ein Agent einem Koordinator untergeordnet ist, muss er diesem Abgaben zahlen (Standardmäßig: 10% seiner gesammelten Nahrung).

2.3.2 Bewegen (move)

Die Agenten können sich pro Runde einmal bewegen und das Feld wechseln. Dabei ist ihr Bewegungsradius eingeschränkt und beläuft sich auf die acht benachbarten Felder. Ein Feld, auf dem sich bereits ein anderer Agent befindet, kann nicht angesteuert werden.

Der Agent betrachtet seine acht Nachbarfelder, sucht sich das beste freie Feld davon aus und bewegt sich auf dieses Feld. Ist der Agent jedoch einem Koordinator untergeordnet, so empfängt er von diesem die Koordinaten für das beste Feld. Jetzt geht es für den Agent nur noch darum, das Zielfeld mit so wenig Schritten wie möglich zu erreichen. Hat der Agent es in die unmittelbare Nähe des Feldes geschafft, so schaut er sich dort nach dem besten Feld um und besetzt es. Sobald er von seinem Koordinator ein neues Zielfeld genannt bekommt, beginnt das ganze von vorn.

2.3.3 Fortpflanzen (breed)

Wie schon im Punkt 2.2.4 beschrieben, unterliegen Agenten einer Reihe von Restriktionen, wenn es um die Fortpflanzung geht. Über Parameter kann außerdem festgelegt werden, dass z.B. eines der unmittelbaren Nachbarfelder frei sein muss, andernfalls kann kein Nachkomme erzeugt werden. In der Standardkonfiguration reicht es aus, wenn irgendein Feld frei ist.

2.3.4 Koordinierung beginnen (startCoordinating)

Um Einfluss zu gewinnen, kann ein Agent sich zum Koordinator erklären. Dies ist nur möglich, wenn er sich zur Zeit der Entscheidung nicht koordinieren lässt.

Sobald der Agent zum Koordinator geworden ist, kann sich ihm jeder freie Agent unterordnen. Hat sich ihm ein Agent untergeordnet, übermittelt dieser dem Koordinator sein gesamtes Wissen über die Umwelt. Danach arbeitet der Koordinator in jeder Runde ein Zielfeld für den Agenten aus und übermittelt es an diesen. Allerdings hat der Koordinator durch diesen Mehraufwand auch einen erhöhten

Stoffwechsel. Da aber die untergeordneten Agenten eine Abgabe ihrer gesammelten Nahrung an ihn zahlen, ist es auf lange Sicht trotzdem erstrebenswert Koordinator zu werden. Außerdem wird durch jeden zusätzlichen untergeordneten Agenten der Einfluss erhöht, was wiederum die Gesamtbefriedigung des Agenten steigert.Deshalb ist ihm sehr daran gelegen, die besten Felder für seine untergeordneten Agenten auszuarbeiten. Das beste Feld wird wie folgt ermittelt:

1. Die Nahrungsquantität auf den Feldern wird ermittelt. Dazu wird die Menge der Nahrung auf allen bekannten der neun Felder aufsummiert (das Zentrum zählt dabei doppelt).
2. Die Qualität der Felder wird ermittelt. Ein Feld mit Nahrung ohne Agent zählt zwei Punkte, ein Feld mit Nahrung mit Agent einen Punkt, ein Feld mit Agent ohne Nahrung minus zwei Punkte und ein unbekanntes Feld null Punkte (auch hier zählt das Zentrum doppelt).
3. Der Abstand des Feldes wird ermittelt. Hierzu wird der Quotient aus Reichweite des Agenten und Abstand des Feldes (in Schritten) gebildet.

Diese drei Zahlen werden zu einer Gütemaßzahl aufmultipliziert. Diese bildet die Grundlage der Zielfindung.

2.3.5 Koordinierung beenden (endCoordinating)

Möchte ein Koordinator nicht länger seine Tätigkeit ausüben, so kann er jederzeit damit aufhören. Sobald er aufhört zu koordinieren, ist er für alle Agenten wieder sichtbar als normaler Agent und erhält auch wieder seinen normalen Stoffwechsel. Allerdings fallen auch die Zahlungen der ehemals untergeordneten Agenten weg sowie die Übermittlung der Informationen bezüglich ihrer Umwelt. Die Agenten, die sich ihm untergeordnet hatten, bewegen sich noch bis zum letzten mitgeteilten Zielfeld, aber kümmern sich danach wieder selber um die Nahrungssuche oder ordnen sich einem neuen Koordinator unter.
Der nun „normale“ Agent kann sich jetzt selber Koordinatoren unterordnen.

2.3.6 Unterordnen (subordinate)

Wenn ein Agent weder Koordinator ist und noch niemandem untergeordnet ist, so kann er sich bei Bedarf einem Koordinator unterordnen. Wurde er noch nie koordiniert, so schaut er sich in seinem Sichtradius nach Koordinatoren um und ordnet sich einem unter. Wenn der Agent zu früheren Zeiten schon einmal koordiniert wurde, so prüft er seine damaligen Beziehungen und ordnet sich dem Koordinator unter, dem er die beste Bewertung gegeben hat (sofern dieser noch Koordinator ist). Anschließend erhält der Agent von seinem Koordinator jede Runde die Koordinaten für ein Zielfeld übermittelt und gibt einen Teil seiner gesammelten Nahrung an seinen Koordinator ab. Bei dem Beginn der Beziehung merkt sich der Agent seinen Nahrungsvorrat. Wenn der Agent die Beziehung zu seinem Koordinator bewerten will, so berechnet er die Differenz zwischen dem gemerkten Nahrungsvorrat und dem aktuellen. Je höher die Differenz, desto besser ist der Koordinator.

2.3.7 Unterordnung beenden (unsubordinate)

Ein untergeordneter Agent kann nach Ablauf einer Sperrfrist die Beziehung kündigen. Diese Restriktion wurde von Andreas König implementiert, da man als untergeordneter Agent ab Beginn der Beziehung Abgaben entrichten muss, auch wenn man noch kein Zielfeld übermittelt bekommen hat bzw. wenn man sich noch nicht im Zielgebiet befindet. Um die Beziehung in den ersten Runden vor einem Abbruch zu schützen, wurde eine Probezeit festgelegt. Erst nach Ablauf dieser kann ein untergeordneter Agent die Beziehung beenden.

2.3.8 Ruhen (rest)

Diese Aktion ist für die Agenten jederzeit möglich und kann auch ausgewählt werden. Hier wird einfach nichts getan und der Agent bleibt auf seinem Feld sitzen.

3 Die NetLogo Replikation

Meine Replikation von Andreas Königs Diplomarbeit habe ich in NetLogo Version 4.1.2 realisiert. In diesem Kapitel werde ich auf die implementierten Funktionen eingehen und den Code dazu erklären. Beginnen werde ich mit der grafischen Benutzeroberfläche, die man sieht, wenn man das Programm mit NetLogo öffnet.

3.1 Grafische Benutzeroberfläche

Während man das Java Programm von Andreas König noch aufwendig per DOS-Shell starten musste, genügt bei einem installierten NetLogo ein Doppelklick auf das Programm. Außerdem muss man hier nicht wie bei Java die gewünschte Einstellung der Parameter per Befehlserweiterung übergeben, sondern kann diese bequem, wie in Abbildung 3 zu sehen ist, über einige Schiebregler und on/off-Schalter einstellen.

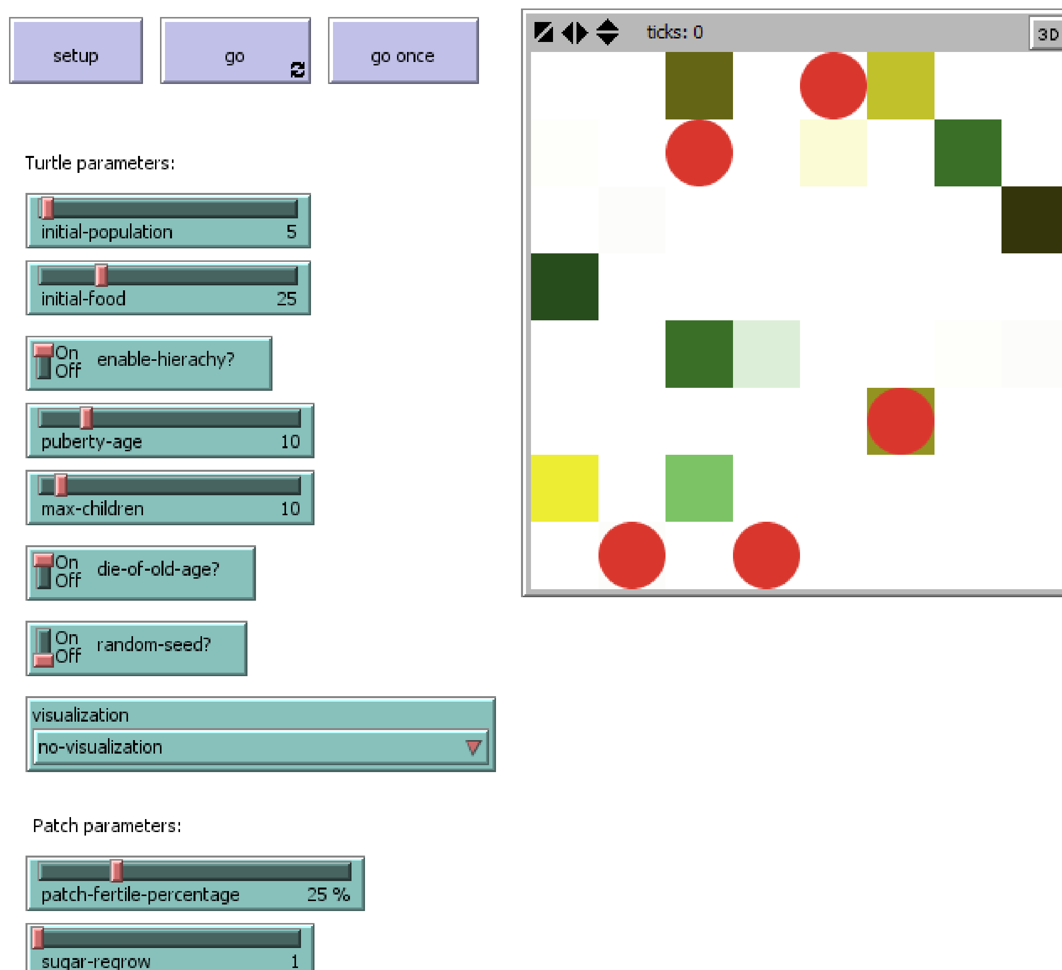

Abbildung 3: GUI der Simulation mit Schiebreglern und on/off-Schaltern

Mittels der Schiebregler der Turtle[1] parameters lassen sich einige Werte einstellen, die für die Agenten relevant sind.

– Initial-population legt fest, mit wie vielen Turtles die Simulation starten soll
– Initial-food legt den Sugarwert fest, mit dem die Anfangsturtles ausgestattet werden
– Enable-hierachy? Mit diesem Schalter legt man fest, ob es möglich sein soll, Hierarchien zu bilden (Koordination/Subordination) oder nicht
– Puberty-age legt fest, ab welchem Alter sich die Turtles vermehren können
– Max-children legt die maximale Anzahl an Nachkommen fest, die ein Turtle insgesamt erzeugen kann
– Die-of-old-age? Mit diesem Schalter stellt man ein, ob Turtles nach einer gewissen Zeit sterben, wenn sie ihr festgelegtes Alter erreicht haben[2] oder ob sie ewig leben können
– Random-seed? Mit diesem Schalter kann man einstellen, ob man für jeden Durchlauf neue Zufallszahlen generiert haben möchte oder ob immer die gleichen Zahlen generiert werden sollen. Dies ist nützlich, wenn man verschiedene Einstellungen unter den gleichen Bedingungen testen will. Off bedeutet hier: Es werden jedesmal neue zufällige Zahlen generiert. On bedeutet, dass immer die gleichen Zufallszahlen generiert werden
– Visualization Mit Hilfe der Einstellung „color-agents-by-metabolism“ färbt NetLogo die Turtles gemäß ihres Stoffwechsels in verschiedene Rottöne ein. Ein hoher Stoffwechsel lässt den Turtle in hellem Rot erscheinen, ein niedriger Stoffwechsel färbt ihn in dunklem Rot ein

[1] In NetLogo werden die Agenten Turtles genannt und Felder heissen Patches

[2] Das Alter eines Turtles wird in der Variable life-expectancy per Zufallszahl generiert

Auch die Felder (Patches) kann man über Parameter steuern

– Patch-fertile-percentage legt fest, wieviel Prozent aller Felder fruchtbar sein sollen
– Sugar-regrow legt fest, wieviel Einheiten Sugar pro Runde auf den Patches nachwächst

Es ist möglich, noch viele weitere Parameter verfügbar zu machen auf der grafischen Benutzeroberfläche, jedoch wird die Ansicht mit wachsender Anzahl an Schiebreglern und Schaltern auch zunehmend unübersichtlicher. Aus diesem Grund habe ich nur ein paar grundlegende Einstellungsmöglichkeiten implementiert.

In Abbildung 4 sind vier sogenannte Plots zu sehen, die Informationen über die laufende Simulation in Echtzeit geben.

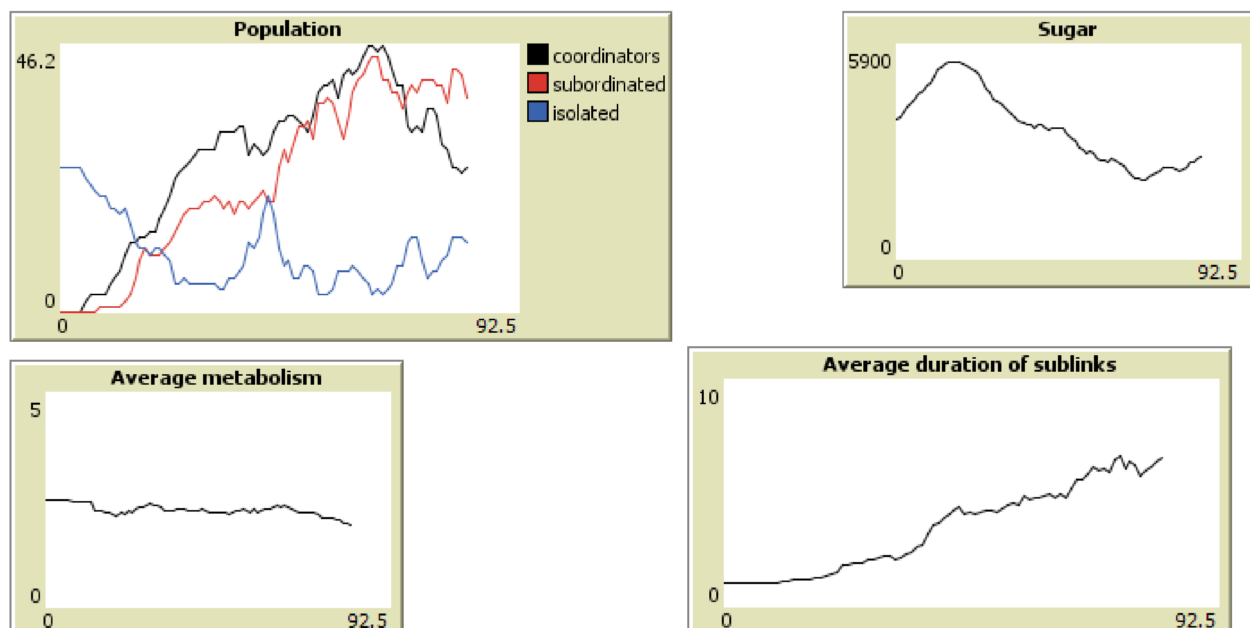

Abbildung 4: Die Plots der Simulation

Der Population Plot zeigt den Verlauf der einzelnen Arten der Bevölkerung an. Man kann genau sehen, wie sich welche Gruppe entwickelt. Bewegt man die Maus über den Graph, so zeigt einem NetLogo die exakte Bevölkerungsanzahl zu diesem Zeitpunkt an. Mit Hilfe dieses Monitors lassen sich Schwankungen sehr gut erkennen und die gesamte Entwicklung über einen langen Zeitraum verfolgen.

Der zweite Plot Average metabolism zeigt den durchschnittlichen Stoffwechsel der Bevölkerung an. Der Stoffwechsel der Turtles wird

über einen Zufallszahlengenerator gesteuert, der Zahlen zwischen 1 und 4 generiert.
Der Sugarplot summiert den Sugar aller Patches auf und zeigt so den Verlauf dieser Ressource während der Simulation.
Im vierten Plot Average duration of sublinks wird die durchschnittliche Lebensdauer der Verbindungen von den untergeordneten Turtles zu ihren Koordinatoren abgebildet.

Hat man alle Einstellungen getroffen, muss der Setup-Button gedrückt werden. Danach hat man die Wahl, die Simulation entweder in einzelnen Schritten zu durchlaufen (dafür muss jede Runde der go-once-Button einmal gedrückt werden), oder die Simulation selbständig laufen zu lassen. Dazu drückt man den go-Button und die Simulation wird so lange laufen, bis sie das Abbruchkriterium erreicht. In diesem Fall lautet das Abbruchkriterium: halte die Simulation an sobald keine Turtles mehr existieren.

Um tiefgreifende Änderungen vorzunehmen, muss man in den Programmcode eingreifen. Dieser befindet sich bei NetLogo 4.1.2 hinter dem Register „Procedures“. Im nächsten Kapitel wird der Code ausführlich erklärt.

3.2 Der Programmcode

3.2.1 Das Spielfeld

Das Spielfeld besteht bei NetLogo aus den Patches. In meiner Simulation ist dieses Spielfeld 50x50 Patches groß. Man kann den Patches verschiedene Variablen zuweisen, um später damit zu arbeiten. In meinem Fall haben alle Patches drei Variablen, wie im folgenden Listing 1 zu sehen ist.

```
01    patches-own
02    [
03      psugar
04      max-psugar
05      growth-rate
06    ]
```

Listing 1: Initialisierung der Patch Variablen

Die Variable PSUGAR speichert den Sugarwert, der sich aktuell auf dem Patch befindet. Die zweite Variable MAX-PSUGAR gibt an, wie viel Sugar sich maximal auf einem Patch befinden darf. Mittels der Variable GROWTH-RATE wird festgelegt, wie viel Einheiten Sugar pro Runde nachwachsen. Die verschiedenen Variablen werden bei NetLogo für die jeweiligen Agenten (Patches, Turtles, Links, etc.), wie in Listing 1 zu sehen ist, in den AGENTEN-OWN Prozeduren festgehalten, Patch Variablen in der PATCHES-OWN, Turtle Variablen in TURTLES-OWN und so weiter. Allerdings sollte man in diese Bereiche nur Variablen eintragen, die man später im ganzen Programm an unterschiedlichen Stellen wieder aufrufen muss. Benötigt man dagegen eine gewisse Variable nur innerhalb einer Prozedur, ist es sinnvoller, diese Variable temporär in der Prozedur zu deklarieren. Mehr dazu in Kapitel 3.2.3 „Rest“. In einer zweiten Prozedur werden den Variablen dann Werte zugewiesen. In diesem Fall trägt die Prozedur den Namen TO SETUP-PATCHES und ist in Listing 2 dargestellt.

```
01 to setup-patches
02  ask patches
03  [ set pcolor (white)]
04
05  ask n-of (count patches * (patch-fertile-percentage / 100))
06  patches
07   [
08    set max-psugar 20
09    set psugar random (21 - 1) + 1
010   set pcolor (green + 5.9 – psugar)
011   set growth-rate sugar-regrow
012  ]
013 end
```

Listing 2: Der Aufbau des Spielfelds mit Wertzuweisung der Variablen

In der TO SETUP-PATCHES Prozedur werden zuerst alle Patches gefragt (aufgerufen) und ihre Farbe auf weiß gesetzt. Im nächsten Schritt wird dann eine zufällige Anzahl an Patches angewiesen, den maximalen Sugarwert auf 20 zu setzen und einen Zufallsstartwert für den Sugar, der auf ihnen zu Beginn der Simulation sein soll zu generieren. Dieser Wert soll zwischen 1 und 20 liegen. In Zeile 010 werden die Patches angewiesen, ihre Farbe zu ändern auf grün + 5,9 – psugar. Dies hat den Effekt, dass die Patches unterschiedliche Farben aufweisen. Ein Patch, auf dem viel Sugar liegt, ist dunkler als ein Patch, auf dem wenig Sugar liegt. Somit lassen sich für den Benutzer schnell erkennen, welche Patches ertragreich sind und welche eher weniger. Als letzter Punkt wird die Nachwuchsrate des Sugars pro Runde auf den Wert von SUGAR-REGROW gesetzt. Diesen Wert sowie die Anzahl an zufälligen Patches, die angesprochen werden sollen in Zeile 05, werden vom Benutzer auf der grafischen Oberfläche mittels zweier Schiebregler eingestellt. Siehe hierzu Abbildung 3, patch-fertile-percentage und sugar-regrow.

Da die TO SETUP-PATCHES Prozedur allerdings nur am Anfang der Simulation aufgerufen wird, müssen noch zwei weitere Prozeduren eingebaut werden, damit die Patches ihre Farbe jede Runde dem aktuellen Sugarwert anpassen, und vor allem muss auch der Sugar

jede Runde nachwachsen. Diese beiden Funktionen werden in den Prozeduren `TO PATCH-RECOLOR` und `TO PATCH-GROWBACK` abgebildet. Siehe hierzu folgendes Listing 3.

```
01    to patch-recolor
02       if (pcolor != white)
03       [ set pcolor (green + 5.9 - psugar) ]
04    end
05
06    to patch-growback
07       if (psugar < max-psugar)
08       [ set psugar (psugar + growth-rate)]
09    end
```

Listing 3: Einfärbung der Patches und Sugarnachwuchs pro Runde

In der `TO PATCH-RECOLOR` Prozedur werden die Patches gefragt, ob ihre Farbe nicht weiß ist. Trifft das zu, so aktualisieren sie ihre Farbe und passen sie dem aktuellen psugar Wert an.

Bei der `TO PATCH-GROWBACK` Prozedur werden die Patches darauf überprüft, ob sie bereits den maximalen Sugarwert erreicht haben. Ist dem nicht so, wird der Sugarwert um den vom Benutzer eingestellten Wert erhöht.

Diese beiden Prozeduren werden jede Runde aufgerufen.

3.2.2 Die Agenten

In dieser Version des Sugarscape Szenarios haben die Agenten fünf verschiedene Bedürfnisse, die sie mittels acht unterschiedlichen Aktionen befriedigen können. Die Bedürfnisse wurden bereits in Kapitel 2.2 „Bedürfnisse“ beschrieben, die Aktionen in Kapitel 2.3 „Aktionen“. Die nachfolgende Tabelle 1 listet alle Aktionen auf sowie die Werte, die für die Berechnung ermittelt werden müssen. Im Anschluss an die Tabelle werde ich auf die jeweiligen Prozeduren im Code eingehen und diese erklären.

	survival	**wealth**	**breeding**	**influence**	**curiosity**
rest	$survival_{DEF}$	$wealth_{DEF}$	$breeding_{DEF}$	$influence_{DEF}$	$curiosity_{DEF}$
move	$survival_{M}$	$wealth_{M}$	$breeding_{DEF}$	$influence_{DEF}$	$curiosity_{M}$
breed	$survival_{B}$	$wealth_{B}$	$breeding_{B}$	$influence_{DEF}$	$curiosity_{DEF}$
coordinate	$survival_{SC}$	$wealth_{SC}$	$breeding_{DEF}$	$influence_{SC}$	$curiosity_{SC}$
endcoordinate	$survival_{EC}$	$wealth_{EC}$	$breeding_{DEF}$	$influence_{EC}$	$curiosity_{EC}$
subordinate	$survival_{SUB}$	$wealth_{SUB}$	$breeding_{DEF}$	$influence_{DEF}$	$curiosity_{DEF}$
unsubordinate	$survival_{UNSUB}$	$wealth_{UNSUB}$	$breeding_{DEF}$	$influence_{DEF}$	$curiosity_{DEF}$
gather	$survival_{G}$	$wealth_{G}$	$breeding_{DEF}$	$influence_{DEF}$	$curiosity_{DEF}$

Tabelle 1: Bedürfnisse/Aktionen und die benötigten Werte

3.2.3 Rest

Um die `DEF` Werte für die Bedürfnisse zu errechnen, muss jeder Turtle die sechs folgenden Prozeduren aufrufen.

1. To baseValueFoodForField

Diese Prozedur dient hauptsächlich dazu, den Hilfswert `bfood` zu errechnen. Dieser Wert dient zur Bewertung der Umgebung des Turtles. Des weiteren werden in dieser Prozedur auch die Werte `curiositydef`, `curiositysc` und `curiosityec` berechnet. Die letzten beiden Werte werde ich später in den dazugehörigen Kapiteln 3.2.6 und 3.2.7 wieder aufgreifen. Zuerst werden einige temporäre Variablen angelegt, zu sehen an dem Wort `let` vor dem Variablennamen. Der Vorteil an temporären Variablen ist, dass man sie nicht im Kopf des Programms (bspw. In `TURTLE-OWN`) deklarieren muss, sondern nur innerhalb der benötigten Prozedur. Dadurch wird der Code übersichtlicher und verständlicher. Allerdings gelten diese Variablen auch nur innerhalb der Prozedur, in der sie deklariert wurden und können nicht von anderen Prozeduren aufgerufen werden. Der Wert von `bfood` entsteht, indem der Turtle, der diese Prozedur aufruft, die Patches in seiner `planning-range` (Standardmäßig: 5) befragt und überprüft, ob diese in seinem Gedächtnis vorhanden sind. Findet der Turtle die Koordinaten des beobachteten Patches in seinem Gedächtnis, so berechnet er die Distanz zu diesem Patch und

summiert das Sugar-Distanzverhältnis auf (Zeile 018). Findet er den beobachteten Patch nicht in seinem Gedächtnis, so wird die Distanz berechnet und das Unbekannt-Distanzverhältnis aufsummiert (Zeile 023). Siehe hierzu Listing 4.

```
01 to baseValueFoodForField
02   let sumUnknown 0
03   let sumFood 0
04   let count-known 0
05   let known-places mem-pos
06   let known-food mem-ts
07   let my-current-x pxcor
08   let my-current-y pycor
09
010        ask patches in-radius planning-range
011        [
012          ifelse member? list pxcor pycor known-places
013          [
014            let dist distancexy my-current-x my-current-y
015            let pos position list pxcor pycor known-places
016            let food last item pos known-food
017              if food >= 0
018                [set sumFood sumFood + food / (2 ^ dist)]
019            set count-known count-known + 1
020          ]
021          [
022            let dist distancexy my-current-x my-current-y
023            set sumUnknown sumUnknown + 1 / (2 ^ dist)
024          ]
025         ]
026        set sumFood sumFood / count-known
027
028          if (sumFood > 1)
029          [ set sumFood 1]
030        set bfood sumFood
031          if (sumUnknown > 1)
032          [ set sumUnknown 1]
033        set curiositydef sumUnknown
034        set curiositysc (curiositydef + (1 - curiositydef) * fsc)
035        set curiosityec (curiositydef * (1 - fec))
036        end
```

Listing 4: baseValueFoodForField Prozedur

In den Zeilen 05 und 06 werden zwei temporäre Listen angelegt namens KNOWN-PLACES und KNOWN-FOOD. Deren Inhalt sind die zuvor in der Prozedur TO MEMORY-FILL angelegten Listen der Umgebung eines Turtles. Mehr zu dieser Prozedur in Kapitel X.X.X. Die Listen beinhalten die Koordinaten der Patches in Sichtweite des Turtles, den Tick[3], an dem sie sich umgeschaut haben und den Sugarwert des Patches. Es ist in der baseValueFoodForField Prozedur nötig, diese von den Turtles besessenen Listen in neutrale temporäre Listen zu packen, da in Zeile 010 die Patches gefragt werden, in die Listen zu schauen. Jedoch besitzen Patches nicht das Recht, in Turtle Variablen hinein zu schauen. Das Gleiche gilt andersherum genauso. In die neutralen temporären Listen, die von niemandem besessen werden, können die Patches jedoch ohne weiteres hinein schauen. In Zeile 012 wird gefragt ob, der aktuelle Patch in der Gedächtnisliste vorhanden ist. Trifft dies zu, wird eine temporäre Variable namens DIST angelegt, welche die Distanz zwischen dem Patch, auf dem der Turtle aktuell sitzt, und dem angefragten Patch berechnet. Danach wird eine Variable erstellt, welche die Position der X/Y Koordinaten innerhalb der Liste enthält. Mit Hilfe dieser Variable wird nun in der zweiten Gedächtnisliste ermittelt, wie viel Sugar sich laut Eintrag auf dem Patch befindet. Falls der Sugarwert größer als 0 ist, wird der Summenwert SUMFOOD erhöht um den Foodwert / 2 hoch DIST. Diese Berechnung hat zur Folge, dass Patches, die weiter weg sind, vom Turtle eine schwächere Bewertung bekommen, da sie kleinere Werte produzieren.

Wenn der angefragte Patch allerdings nicht in der Gedächtnisliste enthalten ist, so wird wieder eine Distanzvariable erzeugt und der Summenwert SUMUNKNOWN erhöht um 1 / 2 hoch DIST. Hierbei geht es darum herauszufinden, wie viele Felder in der Umgebung des Turtles unbekannt sind. Auch hier gilt: je weiter das Feld vom Turtle entfernt ist, desto unattraktiver ist es. Da in die Berechnungen nur Werte zwischen 0 und 1 einfließen sollen, wird zum Schluss noch überprüft, ob die aufsummierten Werte größer als 1 sind. Trifft dies zu, so

[3] Der Tick gibt die aktuelle Rundennummer an und startet bei 0

werden die Ergebnisse auf den Wert 1 (Maximum) gesetzt. Der Wert `bfood` fasst also die Nahrungsmenge in der Umgebung zusammen und gibt an, ob der Turtle sich in einer guten Umgebung befindet oder nicht. Der Wert von `curiositydef` gibt an, ob der Turtle sich in einer Umgebung befindet, in der es viele unbekannte Patches gibt (gut für die Befriedigung von curiosity) oder ob ihm alle Patches bekannt sind.

2. To survival-def

`SURVIVALDEF` ist der erste Wert, der für die endgültige Befriedigungsabschätzung von `REST` benötigt wird. Dieser wird in der `to survival-def` Prozedur ermittelt. Das nachfolgende Listing 5 beinhaltet die komplette Prozedur.

```
01    to survival-def
02          set roundstosurvive 20
03          let tempErg 0
04          let temp 0
05          let resultsurvivaldef 0
06
07          set temp (sugar / (metabolism * roundstosurvive))
08          ifelse (temp > 1)
09          [ set tempErg 1 ]
010         [ set tempErg temp ]
011
012        set resultsurvivaldef ((0.95 * tempErg) + (0.1 * bfood))
013         ifelse (resultsurvivaldef > 1)
014         [ set survivaldef 1]
015         [ set survivaldef resultsurvivaldef]
016
017         set survivalec (survivaldef * (1 - fec))
018
019   end
```

Listing 5: Die to survival-def Prozedur

In der Prozedur wird der Turtle Variable `ROUNDSTOSURVIVE` der Wert 20 zugewiesen und ein paar temporäre Variablen deklariert. Danach folgt

die erste Berechnung, in welcher der aktuelle Sugarvorrat des Turtles durch seinen Stoffwechsel (`metabolism`) mal die Anzahl an Runden, für die das Überleben des Turtles gesichert sein soll, geteilt wird. Anschließend erfolgt wieder die Überprüfung, ob sich der Wert zwischen 0 und 1 befindet und die Sicherstellung, dass der Wert maximal 1 beträgt. Danach folgt die zweite Berechnung (Zeile 012), bei der auf den zuvor berechneten Wert `BFOOD` aus der `baseValueFoodForField` Prozedur zugegriffen wird. Auch danach erfolgt wieder die Überprüfung und Sicherstellung, dass das Ergebnis maximal einen Wert von 1 hat.

3. To saturation

In dieser kurzen Prozedur wird der Arkustangens vom aktuellen Sugarvorrat des Turtles und des durchschnittlichen Wohlstands der Turtles errechnet. Als Ergebnis erhält man hier wie gewohnt einen Wert zwischen 0 und 1. Im nachfolgenden Listing 6 ist die Prozedur aufgeführt.

```
01    to saturation
02      set avwealth 75
03      set resultsaturation 0
04      set resultsaturation ((atan sugar avwealth) / 180)
05    end
```

Listing 6: Die to saturation Prozedur

4. To wealth-def

Um den Wert für `wealthdef` auszurechnen, greift diese Prozedur auf zwei Werte anderer Prozeduren zu, nämlich `BFOOD` und `resultsaturation`. Mit Hilfe dieser Werte kann das Ergebnis berechnet werden. Wie im Listing 7 zu sehen ist, wird auch hier nach der Berechnung sofort überprüft, ob sich der Wert im festgelegten Spektrum zwischen 0 und 1 befindet.

```
01    to wealth-def
02       let resultwealthdef 0
03       set resultwealthdef ((0.9 * resultsaturation) + (0.1 *      bfood))
04       ifelse (resultwealthdef > 1)
05       [set wealthdef 1]
06       [set wealthdef resultwealthdef]
07
08      set wealthec (wealthdef * (1 - fec))
09    end
```

Listing 7: Die to wealth-def Prozedur

5. To breeding-def

In dieser Prozedur wird der Standardwert für `breedingdef` (Fortpflanzen) berechnet. Dabei wird zunächst das Alter des Turtles überprüft. Ist er zu jung um sich fortzupflanzen, so gilt das Bedürfnis sich fortzupflanzen als voll befriedigt und erhält den Wert 1. Wenn das Alter des Turtles genau dem Alter entspricht, ab dem man sich fortpflanzen kann, dann wird der Variable `breedingdef` ein zufälliger Wert zwischen 0 und 1 zugewiesen. Ansonsten wird der Wert von `breedingdef` pro Runde um den Wert von `birthrecreationfactor` verringert. Diese rundenweise Herabstufung der Befriedigung von `breedingdef` veranlasst den Turtle früher oder später Nachkommen zu erzeugen, um das Bedürfnis wieder voll zu befriedigen. Im nachfolgenden Listing 8 ist die Prozedur dargestellt.

```
01    to breeding-def
02      set birthrecreationfactor 0.95
03
04      ifelse ((ticks - birthday) < pubertyage)
05      [ set breedingdef 1 ]
06      [
07        ifelse ((ticks - birthday) = pubertyage)
08        [ set breedingdef (random (10) + 1) / 10 ]
09        [ set breedingdef breedingdef * birthrecreationfactor ]
010     ]
011   end
```

Listing 8: Die to breeding-def Prozedur

6. To saturation-i

Dies ist die letzte Prozedur, die benötigt wird, um einen Wert für `REST` zu berechnen. Hier wird `influencedef` berechnet. Im folgenden Listing 9 ist die Prozedur zu sehen.

```
01          to saturation-i
02                let subsagent 0
03                let avsubs 5
04                set influencedef 0
05
06                set subsagent (count my-out-coord-to-sub)
07                set influencedef ((atan subsagent avsubs) / 180)
08          end
```

Listing 9: Die to saturation-i Prozedur

Um den Einfluss (influence) eines Turtles festzustellen, muss zunächst überprüft werden, wie viele untergeordnete Turtles er koordiniert. Dazu wird in Zeile 06 die Anzahl an ausgehenden Koordinatorverbindungen gezählt und der Variable `subsagent` zugewiesen. Danach erfolgt die Berechnung mittels Arkustangens für die Werte `subsagent` und `avsubs`, wobei `avsubs` hier die durchschnittliche Anzahl an untergeordneten Turtles pro Koordinator darstellt. Standardmäßig beträgt dieser Wert 5.

Sind alle benötigten Werte für die Aktion von `REST` berechnet, können sie in der `to-report sat-rest` Prozedur verwendet werden. Siehe Listing 10.

```
01          to-report sat-rest
02          report ((survivaldef * 0.5) + (wealthdef * 0.1) + (breedingdef
            * 0.1) + (influencedef * 0.2) + (curiositydef * 0.1))
03          end
```

Listing 10: Die to-report sat-rest Prozedur

In der Endkalkulation werden die zuvor errechneten Werte mit festgelegten Gewichtungen (siehe [Kö2000, S. 33]) versehen und multipliziert. Dies geschieht für jede Aktion, so dass man am Ende sieben Werte erhält, die angeben, wie viel Befriedigung sie dem Turtle zuführen können. Zum Beispiel könnte move eine Gesamtbefriedigung

von 0,5 bieten, gather eine von 0,4 etc.. Diese Werte werden dann im Auswahlverfahren benötigt und beeinflussen, welche Aktion der Turtle am Ende ausführt.

3.2.4 Move

In dieser Aktion geht es darum, das beste Feld für den Turtle ausfindig zu machen, so dass er, wenn er sich für die Aktion move entscheidet, zu diesem Feld bewegt. Die Zielsuche hängt hier davon ab, ob der Turtle koordiniert wird – in diesem Fall arbeitet der Koordinator ein Feld für den Turtle aus – oder ob er selbständig ist. In dem Fall arbeitet er selber ein Feld für sich aus. Jeder Turtle führt zur Berechnung die folgenden Prozeduren aus.

1. to baseValueFoodForField-m

In dieser Prozedur wird:

- das beste Feld in der Umgebung für den Turtle ausfindig gemacht
- BFOODM berechnet
- CURIOSITYM berechnet

Um das beste Feld zu finden, werden zuerst alle Patches in der PLANNING-RANGE angewiesen, ihre X/Y Koordinaten in eine temporäre Liste einzutragen. Danach werden die Patches erneut angefragt, diesmal fragen diese allerdings ihre acht Nachbarpatches und diese müssen ihre Koordinaten in die temporäre Liste eintragen. Sind deren Koordinaten in der Liste schon enthalten, so ersetzen sie den Eintrag. Dadurch erhält man eine Liste mit den Koordinaten der 25 Patches, die innerhalb der PLANNING-RANGE des Turtles liegen. Im Anschluss wird dann jedes Element aus der Liste gefragt, ob es auch im Gedächtnis des Turtles steht (Zeile 020), also ob es bekannt ist. Ist das der Fall, so wird die Distanz berechnet zwischen dem Turtle und dem Patch, die Koordinaten werden in eine extra Liste geschrieben und der Patch wird nach seinem Sugarwert befragt. Dieser Wert fließt in eine Berechnung ein, deren Ergebnis wiederum in eine weitere Liste geschrieben wird (Zeilen 022-028). Falls der angefragte Patch nicht in der Gedächtnisliste steht, werden die gleichen Funktionen ausgeführt

wie bei der anderen Bedingung, zusätzlich wird jedoch noch die Summe der unbekannten Felder um eins erhöht und es fließt noch das Distanzverhältnis dazu ein (Zeilen 033-040). Am Ende wird wieder überprüft, ob die Summenwerte den Maximalwert von 1 nicht überschreiten. Danach wird das Ergebnis von SUMFOOD in eine Ergebnisliste geschrieben (Zeilen 043-052).

Hat der Turtle diese Berechnung für alle Patches in seiner Liste durchgeführt, sucht er sich den höchsten Eintrag aus der Ergebnisliste heraus und weist BFOODM dessen Wert zu. Außerdem wird noch der höchste Wert aus der PSUGARLIST ermittelt und die Position, an der dieser in der Liste steht, gespeichert. Mittels der Position des Wertes lassen sich die X/Y Koordinaten des dazugehörigen Patches aus der COORDSLIST entnehmen, die zuvor angelegt wurde. Diese Koordinaten stellen den besten Patch dar für den Turtle und werden deshalb in den Turtle Variablen BESTFIELDX und BESTFIELDY zur späteren Verwendung gespeichert (Zeilen 055-059). Der komplette Code der Prozedur ist nachfolgend im Listing 11 dargestellt.

```
01     to baseValueFoodForField-m 1 von 2
02          ask patches in-radius planning-range
03          [
04          set templist lput list pxcor pycor templist
05          ]
06
07          ask patches in-radius planning-range
08          [
09          ask neighbors
010         [
011          ifelse member? list pxcor pycor templist
012           [ let insert-position position list pxcor pycor templist
013               set templist replace-item insert-position templist list
                  pxcor pycor
014           ]
015           [ set templist lput list pxcor pycor templist ]
016         ]
017         ]
018         foreach templist
019         [
020          ifelse member? ? known-places
021           [
022               let dist distancexy first ? last ?
023               set coordslist lput ? coordslist
024               ask patch first ? last ?
025               [
026                set psugarlist lput (psugar / (2 ^ dist)) psugarlist
027             if psugar >= 0
028             [set sumFood sumFood + psugar / (2 ^ dist)]
029           ]
030           set count-fields count-fields + 1
031         ]
032         [
033           let dist distancexy first ? last ?
034           set coordslist lput ? coordslist
035           ask patch first ? last ?
036           [ set psugarlist lput (psugar / (2 ^ dist)) psugarlist
037             if psugar >= 0
038             [set sumFood sumFood + psugar / (2 ^ dist)]
039           ]
040           set sumUnknown sumUnknown + 1 / (2 ^ dist)
041           set count-fields count-fields + 1
042         ]
043      set sumFood sumFood / count-fields
044        if (sumFood > 1)
045       [ set sumFood 1]
```

```
to baseValueFoodForField-m 2 von 2
046   if (sumUnknown > 1)
047   [ set sumUnknown 1 ]
048
049   set curiositym sumUnknown
050   set tempErg sumFood
051   set bfoodmErg tempErg
052   set Erglist lput bfoodmErg Erglist
053   ]
054
055   set bfoodm max Erglist
056   set maxfood max psugarlist
057   set pos position maxfood psugarlist
058   set bestfieldx first item pos coordslist
059   set bestfieldy last item pos coordslist
060   end
```

Listing 11: Die baseValueFoodForField-m Prozedur

2. To survival-m

Die `to survival-m` Prozedur ist im Grunde gleich aufgebaut wie die `to survival-def` Prozedur, nur dass hier anstatt `BFOOD` die Variable `BFOODM` für die Berechnung benutzt wird. Siehe hierzu Listing 12.

```
01    to survival-m
02          set roundstosurvive 20
03          let tempErg 0
04          let temp 0
05          let resultsurvivaldef 0
06
07          set temp (sugar / (metabolism * roundstosurvive))
08          ifelse (temp > 1)
09          [ set tempErg 1 ]
010         [ set tempErg temp ]
011
012         set resultsurvivaldef ((0.95 * tempErg) + (0.1 * bfoodm))
013         ifelse (resultsurvivaldef > 1)
014         [ set survivalm 1]
015         [ set survivalm resultsurvivaldef]
016   end
```

Listing 12: Die to survival-m Prozedur

3. To wealth-m

Auch in dieser Prozedur wird nun der BFOODM Wert verwendet, der Rest ist identisch mit der `to wealth-def` Prozedur. Es wird auch wieder auf die Prozedur `to saturation` zugegriffen, die, wie schon auf Seite 34 beschrieben, den Arkustangens berechnet. Siehe nachfolgendes Listing 13.

```
01    to wealth-m
02       let resultwealthdef 0
03       set resultwealthdef ((0.9 * resultsaturation) + (0.1 * bfoodm))
04       ifelse (resultwealthdef > 1)
05       [set wealthm 1]
06       [set wealthm resultwealthdef]
07    end
```

Listing 13: Die to wealth-m Prozedur

Die Berechnung der Werte BREEDINGDEF und INFLUENCEDEF wurde bereits im Kapitel 3.2.3 „Rest“ beschrieben.
Somit sind nun alle Werte erarbeitet, die für die Berechnung der Befriedigung des Turtles notwendig sind. Im folgenden Listing 14 ist zu sehen wie der Endwert kalkuliert wird, der in die Entscheidungsprozedur einfließt.

```
01          to-report sat-move
02          report ((survivalm * 0.5) + (wealthm * 0.1) + (breedingdef
            * 0.1) + (influencedef * 0.2) + (curiositym * 0.1))
03          end
```

Listing 14: Die to-report sat-move Prozedur

3.2.5 Breed

Für die Aktion Breed müssen survival, wealth und breed neu berechnet werden, da sich durch die Aktion breed (Fortpflanzung) der Nahrungsvorrat des Turtles verringert. Am Ende erhält man als Ergebnis den Wert der Befriedigung, den diese Aktion herbeiführen würde. Für die Berechnung der drei oben genannten Werte muss der Turtle folgende drei Prozeduren aufrufen:

1. To survival-b

Diese Prozedur ist gleich aufgebaut wie z.B. die `to survival-m` Prozedur, jedoch wird hier, wie im Listing 15 zu sehen ist, der Nahrungsvorrat des Turtles um den Wert von `CHILDFOOD` in der Berechnung verringert und der Standardwert `BFOOD` genutzt.

```
01    to survival-b
02          set roundstosurvive 20
03          let tempErg 0
04          let temp 0
05          let resultsurvivaldef 0
06
07          set temp (sugar - childfood / (metabolism * roundstosurvive))
08          ifelse (temp > 1)
09          [ set tempErg 1 ]
010         [ set tempErg temp ]
011
012         set resultsurvivaldef ((0.95 * tempErg) + (0.1 * bfood))
013         ifelse (resultsurvivaldef > 1)
014         [ set survivalb 1]
015         [ set survivalb resultsurvivaldef]
016   end
```

Listing 15: Die to survival-b Prozedur

2. To saturation-b

Auch in dieser Prozedur wird der Nahrungsvorrat des Turtles um den Wert von `CHILDFOOD` verringert und anschließend der Arkustangens berechnet. Siehe hierzu Listing 16.

```
01    to saturation-b
02          set avwealth 75
03          set resultsaturationb 0
04          set resultsaturationb ((atan (sugar – childfood) avwealth /180)
05     end
```

Listing 16: Die to saturation-b Prozedur

3. To wealth-b

In dieser Prozedur wird das Ergebnis aus `to saturation-b` verwendet, um die Auswirkungen auf den Wohlstand des Turtles auszurechnen. Siehe Listing 17.

```
01  to wealth-b
02     let resultwealthdef 0
03     set resultwealthdef ((0.9 * resultsaturationb) + (0.1 * bfood))
04     ifelse (resultwealthdef > 1)
05     [set wealthb 1]
06     [set wealthb resultwealthdef]
07  end
```

Listing 17: Die to wealth-b Prozedur

Nachdem die Berechnung für alle Werte erfolgt ist, kann die `to-report sat-breed` Prozedur aufgerufen werden. Hier werden die verschiedenen Werte zu einem Endergebnis zusammengerechnet. Die Darstellung ist in Listing 18 zu sehen.

```
01      to-report sat-breed
02      report ((survivalb * 0.5) + (wealthb * 0.1) + (breedingb
        * 0.1) + (influencedef * 0.2) + (curiositydef * 0.1))
03      end
```

Listing 18: Die to-report sat-breed Prozedur

3.2.6 Coordinate

Bei dieser Berechnung ist es das Ziel herauszufinden, wie viel Befriedigung die Einnahme der Koordinatoren-Rolle für den Turtle herbeiführen würde. Hierzu müssen alle Werte bis auf breed neu berechnet werden. Für die Fortpflanzung wird hier auf den Wert von `breedingdef` zugegriffen. Um die restlichen Werte zu berechnen, muss der Turtle folgende sechs Prozeduren aufrufen:

1. To saturation-sc0

In dieser Prozedur prüft der Turtle zunächst, wie viele der Turtles in seiner Sichtweite Koordinatoren sind (`subsc`) und wie viele „normale“ sind (`subsnc`). Danach berechnet er das Verhältnis zwischen diesen beiden Populationen und verwendet das Ergebnis zusammen mit der

Variable AVSUBS - welche die durchschnittliche Anzahl an untergeordneten Turtles pro Koordinator darstellt - um den Arkustangens zu berechnen. Dieses Ergebnis wird dann noch mit dem Wert von COORDINATORBSTRENGTH multipliziert. Diese Variable dient dazu, survival, wealth und curiosity aufzuwerten und dadurch den Wechsel in die Koordinatorenrolle für den Turtle interessanter zu machen. Dies wird dadurch erreicht, dass das Ergebnis FSC in die Berechnungen für SURVIVALSC, WEALTHSC und CURIOSITYSC einfließt. Die komplette Prozedur ist im nachfolgenden Listing 19 dargestellt.

```
01      to saturation-sc0
02           let coordinatorBStrength 0.5
03           let avsubs 5
04           let subsc 0
05           let subsnc 0
06           let subspot 0
07       ifelse (isCoordinator = false)
08       [
09           set subsnc count other turtles with [isCoordinator = false] in-
             radius vision
010          set subsc count turtles with [isCoordinator = true] in-radius
             vision + 1
011      ]
012      [
013          set subsnc count turtles with [isCoordinator = false] in-radius
             vision
014          set subsc count turtles with [isCoordinator = true] in-radius
             vision
015      ]
016      ifelse subsc = 0 or subsnc = 0
017          [ set subspot 0 ]
018          [ set subspot subsnc / subsc ]
019          set fsc 0
020          set fsc ((atan subspot avsubs) / 180) * coordinatorBStrength
021     end
```

Listing 19: Die to saturation-sc0 Prozedur

2. To survival-sc

In dieser Prozedur wird der Überlebenswert ermittelt. Zu beachten ist hier, dass, wie im Listing 20, Zeile 07 zu sehen, der Stoffwechsel (metabolism) um den Wert COORDFOODPRFACTOR erhöht wird. Der

Stoffwechsel wird deswegen erhöht, weil ein Turtle als Koordinator nicht nur für sich selbst sorgen muss, sondern auch noch für jeden seiner untergeordneten Turtles Felder ausarbeiten muss (siehe Kapitel 2.3.4). In die Endberechnung wird hier auch der in `to saturation-sc0` berechnete Wert `FSC` mit einbezogen (Zeile 018).

```
01    to survival-sc
02      set roundstosurvive 20
03      let tempErg 0
04      let temp 0
05      let resultsurvivalsc 0
06      let survivalsc0 0
07      set temp (sugar  / ((metabolism * coordFoodPRFactor) *
        roundstosurvive))
08
09      ifelse (temp > 1)
010     [ set tempErg 1 ]
011     [ set tempErg temp ]
012
013     set resultsurvivalsc ((0.95 * tempErg) + (0.1 * bfood))
014     ifelse (resultsurvivalsc > 1)
015     [ set survivalsc0 1]
016     [ set survivalsc0 resultsurvivalsc]
017
018     set survivalsc (survivalsc0 + (1 - survivalsc0) * fsc)
019
020   end
```

Listing 20: Die to survival-sc Prozedur

3. To wealth-sc

In dieser Prozedur wird wieder auf den Standardwert `BFOOD` und die Standardfunktion `to saturation` zugegriffen. Des Weiteren fließt hier der Wert `COORDINATORMALUS` in die Berechnung ein. Dieser Wert ist ein prozentualer Abzug des Wohlstands (wealth), den ein Turtle zu Beginn der Koordinatorenrolle durch den erhöhten Stoffwechsel erleidet. Auch in dieser Prozedur wird der Wert `FSC` in der Endkalkulation verwendet, wie in Listing 21, Zeile 011 zu sehen ist.

```
01     to wealth-sc
02      let wealthsc0 0
03      let coordinatorMalus 0.1
04      let resultwealthdef 0
05      set resultwealthdef ((0.9 * resultsaturation) + (0.1 * bfood))
06      ifelse (resultwealthdef > 1)
07       [set wealthdef 1]
08       [set wealthdef resultwealthdef]
09
010     set wealthsc0 wealthdef * (1 - coordinatorMalus)
011     set wealthsc (wealthsc0 + (1 - wealthdef) * fsc)
012   end
```

Listing 21: Die to wealth-sc Prozedur

4. To saturation-sc

In dieser Prozedur wird wie schon in `to saturation-sc0` zuerst überprüft, wie viele Koordinatoren und wie viele „normale“ Turtles sich in der Umgebung befinden. Dann wird das Verhältnis ausgerechnet und der Arkustangens der potentiellen Untergeordneten und der durchschnittlichen Anzahl an untergeordneten Turtles berechnet. Hierbei wird auf den `COORDINATORBSTRENGTH` Faktor verzichtet, da es diesmal nur um den möglichen zu erhaltenen Einfluss (`influence`) geht. Die komplette Prozedur ist im nachfolgenden Listing 22 abgebildet.

```
01    to saturation-sc
02     let avsubs 5
03     let subsc 0
04     let subsnc 0
05     let subspot 0
06
07     ifelse (isCoordinator = false)
08      [
09       set subsnc count other turtles with [isCoordinator = false] in-
         radius vision
010      set subsc count turtles with [isCoordinator = true] in-radius
         vision + 1
011     ]
012     [
013      set subsnc count turtles with [isCoordinator = false] in-radius
         vision
014      set subsc count turtles with [isCoordinator = true] in-radius
         vision
015     ]
016    ifelse subsc = 0 or subsnc = 0
017     [ set subspot 0 ]
018     [ set subspot subsnc / subsc ]
019
020    set resultsaturationsc 0
021    set resultsaturationsc ((atan subspot avsubs) / 180)
022   end
```

Listing 22: Die to saturaion-sc Prozedur

5. To influence-sc

Diese Prozedur dient dazu, falls der Turtle kein Koordinator ist, dem Einflussfaktor einen Wert zuzuweisen. Dieser kommt von der oben beschriebenen `to saturation-sc` Prozedur und fließt in die Endkalkulation von `sat-coord` ein. Die Prozedur ist im folgenden Listing 23 abgebildet.

```
01    to influence-sc
02
03     if isCoordinator = false
04     [set influencesc resultsaturationsc]
05
06    end
```

Listing 23: Die to influence-sc Prozedur

6. To baseValueFoodForField

Die vollständige Prozedur ist beschrieben in Kapitel 3.2.3 „Rest“. In dieser Prozedur wird auch der Wert von `curiositysc` berechnet (Zeile 041). Hier wird auch auf den Wert `FSC` aus der `to saturation-sc0` Prozedur zugegriffen.

Nach Berechnung aller Werte werden diese in der to-report sat-coord Prozedur zu einer Zahl zusammengerechnet. Siehe hierzu Listing 24.

```
01    to-report sat-coord
02     report ((survivalsc * 0.5) + (wealthsc * 0.1) + (breedingdef * 0.1)
03     + (influencesc * 0.2) + (curiositysc * 0.1))
04    end
```

Listing 24: Die to-report sat-coord Prozedur

3.2.7 Endcoordinate

Bei der Berechnung des Endcoordinate Wertes wird ermittelt, wie viel Befriedigung es dem Turtle bringen würde, sein Dasein als Koordinator aufzugeben und als „normaler“ Turtle weiter zu existieren. Auf der Pro-Seite steht hier, dass er als normaler Turtle auch wieder einen niedrigen Stoffwechsel besitzt. Als Contra-Punkte stehen dem aber der Informationsverlust und der Einflussverlust gegenüber. Da der Turtle, sobald er sein Dasein als Koordinator beendet, keine Informationen über die Umgebung von seinen Untergebenen bekommt, kann er sich nur noch auf seine eigenen Informationen berufen. Außerdem wird sein Bedürfnis nach Einfluss durch den Wegfall der Untergebenen nicht mehr befriedigt. Die benötigten Werte werden in den folgenden Prozeduren berechnet.

1. To saturation-ec

So wie bei dem Beginn einer Koordinatorenrolle ein Faktor berechnet wird der Wohlstand, Überleben, Einfluss und Neugier positiv beeinflusst, wird nun ein Faktor berechnet, der *„den zukünftigen Wegfall der Untergebenen und seinen Einfluss auf die Bedürfnisse abschätzt“* [Kö2000, S. 27]. Siehe hierzu Listing 25.

```
01    to saturation-ec
02      let subsagent 0
03      let avsubs 5
04      let malus 0.5
05      set fec 0
06
07      set subsagent (count my-out-coord-to-sub)
08      set fec ((atan subsagent avsubs) / 180) * malus
09     end
```

Listing 25: Die to saturation-ec Prozedur

2. To survival-def

Der in `to saturation-ec` berechnete Wert `FEC` wird nun bei der Berechnung der anderen Werte benutzt. Der Wert von `survivalec` wird in der Prozedur `to survival-def` mit ausgerechnet, da er nur die Werte `survivaldef` und `FEC` benötigt, wie im Listing 26 zu sehen ist.

```
01    to survival-def
02    ...
03          ifelse (resultsurvivaldef > 1)
04          [ set survivaldef 1]
05          [ set survivaldef resultsurvivaldef]
06
07          set survivalec (survivaldef * (1 - fec))
08    end
```

Listing 26: Die verkürzte to survival-def Prozedur

2. To wealth-def

Ebenso wie `survivalec` wird der Wert von `wealthec` in der `default` Prozedur ausgerechnet. Die verkürzte Fassung der Prozedur ist in Listing 27 dargestellt.

```
01    to wealth-def
02    ...
03       [set wealthdef resultwealthdef]
04
05       set wealthec (wealthdef * (1 - fec))
06    end
```

Listing 27: Die verkürzte to wealth-def Prozedur

3. To baseValueFoodForField

Wie auch `curiositysc` wird der Wert von `curiosityec` innerhalb der `to baseValueFoodForField` Prozedur berechnet. Im folgenden Listing 28 ist der Ausschnitt des Codes zu sehen, in dem die Berechnung stattfindet.

```
01    to baseValueFoodForField
02      ...
03      set curiosityec (curiositydef * (1 - fec))
04    end
```

Listing 28: baseValueFoodForField Prozedur

Nach erfolgreicher Berechnung sämtlicher Werte wird die Endkalkulation für die Beendigung der Koordinatorenrolle durchgeführt. Siehe Listing 29.

```
01    to-report sat-coord
02      report ((survivalsc * 0.5) + (wealthsc * 0.1) + (breedingdef * 0.1)
03      + (influencesc * 0.2) + (curiositysc * 0.1))
04    end
```

Listing 29: Die to-report sat-coord Prozedur

3.2.8 Subordinate

Bei der Berechnung der Werte für die Aktion subordinate wird ermittelt, wie sehr es den Turtle befriedigen würde, wenn er sich einem Koordinator unterordnet. Hier gibt es Vorteile sowie Nachteile, die der Turtle abwägen muss. Auf der einen Seite wird der Koordinator jede Runde ein Zielfeld für den Turtle ausarbeiten, auf dem in der Regel ein sehr hoher Nahrungsvorrat liegt. Dies wiederum bedeutet, dass der Turtle eventuell öfter in unbekanntes Gebiet vorstößt, was seine Neugier befriedigt. Allerdings ist sein Bedürfnis nach Einfluss zu null Prozent befriedigt. Hinzu kommt noch, dass er als untergeordneter Turtle seinem Koordinator jedesmal, wenn er Sugar konsumiert, einen Teil der Nahrung abgeben muss. Außerdem vertraut er dem Koordinator quasi blind und es kann auch vorkommen, dass das Feld, auf das er geschickt wird, schon leer oder besetzt ist, bis er dort

ankommt. Für die Aktion subordinate müssen lediglich die Werte survivalsub und wealthsub neu berechnet werden. Bei den drei anderen wird auf die Ergebnisse der Standardberechnung zugegriffen.

1. To saturation-sub

Zunächst muss diese Prozedur aufgerufen werden, um den später benötigten Wert FSUB zu berechnen. Im folgenden Listing 30 ist die Prozedur dargestellt.

```
01    to saturation-sub
02     let coordeval 0
03     set avwealth 75
04     let subordCoordValuesInfluence 1
05
06     ifelse ([coord-value] of my-out-sub-to-coord = [] )
07      [ set coordeval 10 + random-float 10.0 ]
08      [ set coordeval (max [coord-value] of my-out-sub-to-coord) ]
09
010    let resultsaturationsub 0
011    set fsub 0
012    set resultsaturationsub ((atan coordeval avwealth) / 180) *
013    subordCoordValuesInfluence
014    set fsub (abs coordeval) * resultsaturationsub
015
016   end
```

Listing 30: Die to saturation-sub Prozedur

Falls der Turtle in der Vergangenheit schon einmal koordiniert wurde, hat er die Bewertung des Koordinators in der Variable COORDEVAL gespeichert. Deshalb wird in Zeile 06 zunächst das Maximum dieses Wertes (also der beste Koordinator) ermittelt. Ist der Turtle noch nie koordiniert worden, so wird in Zeile 07 ein Zufallswert generiert der zwischen 10 und 20 liegt. Nach dieser Abfrage wird dann zunächst der Arkustangens für die Werte COORDEVAL und AVWEALTH berechnet. Der Wert AVWEALTH stellt dabei den durchschnittlichen Wohlstand der Turtles dar. Anschließend wird der FSUB Faktor berechnet, der die Werte von survival und wealth modifiziert.

2. To survival-sub

In dieser Prozedur wird der zuvor berechnete Faktor FSUB eingesetzt um den Wert für SURVIVALSUB auszurechnen. Siehe hierzu Listing 31.

```
01    to survival-sub
02      ifelse fsub >= 0
03        [ set survivalsub (survivaldef + (1 - survivaldef)) * fsub ]
04        [ set survivalsub (survivaldef * (1 + fsub)) ]
05    end
```

Listing 31: Die to survival-sub Prozedur

3. To wealth-sub

Im nachfolgenden Listing 32 ist die to wealth-sub Prozedur aufgeführt, in der man sehen kann, wie der Faktor FSUB den Wohlstandswert beeinflusst.

```
01    to wealth-sub
02       ifelse (fsub >= 0)
03         [ set wealthsub (wealthdef + (1 - wealthdef) * fsub) ]
04         [ set wealthsub (wealthdef * (1 + fsub)) ]
05    end
```

Listing 32: Die to weatlh-sub Prozedur

Wie schon am Anfang dieses Kapitels beschrieben, müssen hier nur die beiden Werte für Überleben und Wohlstand neu berechnet werden. In der Endkalkulation (siehe Listing 33) werden dann alle Ergebnisse gewichtet und das Endergebnis ausgerechnet.

```
01    to-report sat-subord
02     report ((survivalsub * 0.5) + (wealthsub * 0.1) + (breedingdef *
03     0.1) + (influencedef * 0.2) + (curiositydef * 0.1))
04    end
```

Listing 33: Die to-report sat-subord Prozedur

3.2.9 Unsubordinate

Bei der Beendigung des untergeordneten Verhältnisses müssen ebenso nur die Werte von `survivalunsub` und `wealthunsub` neu berechnet werden. Während bei der Überlegung zur Unterordnung gilt: je schlechter die Koordinatoren mich in der Vergangenheit betreut haben, desto schlechter wird die Aktion der Unterordnung eingeschätzt, gilt nun: Je besser die Bewertung meines Koordinators ist, desto schlechter wird die Aktion des Verlassens eingeschätzt.

1. To saturation-unsub

Im nachfolgenden Listing 34 ist die Berechnung des Faktors `FUNSUB` dargestellt. Zuvor wird allerdings noch die Bewertung des aktuellen Koordinators berechnet, sofern es einen gibt (Zeilen 07 bis 011).

```
01    to saturation-unsub
02      let coordeval 0
03      set avwealth 75
04      let unsubordCoordValuesInfluence 1
05      let tempSugar sugar
06
07      ask my-out-sub-to-coord with [active = 1]
08       [
09        set current-value tempSugar
010       set coord-value (current-value - start-value)
011       set coordeval coord-value
012      ]
013
014     let resultsaturationunsub 0
015     set funsub 0
016     set resultsaturationunsub ((atan coordeval avwealth) / 180) *
017     unsubordCoordValuesInfluence
018     set funsub (abs coordeval) * resultsaturationunsub
019    end
```

Listing 34: Die to saturation-unsub Prozedur

2. To survival-unsub

Im folgenden Listing 35 ist die `to survival-unsub` Prozedur dargestellt. Hier fließt der `FUNSUB` Faktor aus der `to saturation-unsub` Prozedur ein.

```
01    to survival-unsub
02     ifelse funsub < 0
03     [ set survivalunsub (survivaldef + (1 - survivaldef) * (- funsub)) ]
04     [ set survivalunsub (survivaldef * (1 - funsub)) ]
05    end
```

Listing 35: Die to survival-unsub Prozedur

3. To wealth-unsub

Das folgende Listing 36 beinhaltet die `to wealth-unsub` Prozedur, in der die Auswirkungen auf den Wohlstand ausgerechnet werden, wenn der Turtle sein Untergeordneten-Verhältnis beendet.

```
01    to wealth-unsub
02     ifelse funsub < 0
03     [ set wealthunsub (wealthdef + (1 - wealthdef) * (- funsub)) ]
04     [ set wealthunsub (wealthdef * (1 - funsub)) ]
05    end
```

Listing 36: Die to wealth-unsub Prozedur

Nach der Berechnung der benötigten Werte wird die Endkalkulation `to-report sat-unsubord` aufgerufen. Siehe hierzu Listing 37.

```
01    to-report sat-unsubord
02     report ((survivalunsub * 0.5) + (wealthunsub * 0.1) + (breedingdef *
03     0.1) + (influencedef * 0.2) + (curiositydef * 0.1))
04    end
```

Listing 37: Die to-report sat-unsubord Prozedur

3.2.10 Gather

Für diese Aktion müssen survival und wealth neu berechnet werden, da sich der Nahrungsvorrat des Turtles und die Nahrungsmenge auf dem Feld ändert. Bevor diese berechnet werden können, muss allerdings zuerst ein neuer `BFOOD` Wert ausgerechnet werden, bei dem das Ausgangsfeld gesondert behandelt wird.

1. To baseValueFoodForField-g

In dieser Prozedur wird, wie schon in der normalen `baseValueFoodForField` Variante, überprüft, ob die Patches in der `planning-range` des Turtles in seinem Gedächtnis sind. Für alle Patches, für die dies zutrifft, werden die Distanz und der Nahrungsvorrat berechnet. Zusätzlich wird in dieser Prozedur das Feld, auf dem sich der Turtle befindet, nur mit der Hälfte des tatsächlichen Nahrungsvorrates bewertet. Am Ende erhält der Turtle eine modifizierte Bewertung seiner Umgebung. Die komplette Prozedur ist in folgendem Listing 38 abgebildet.

```
01    to baseValueFoodForField-g
02     let ownFood 0
03     let sumFood 0
04     let count-known 0
05     let known-places mem-pos
06     let known-food mem-ts
07     let my-current-x pxcor
08     let my-current-y pycor
09
010    ask patches in-radius planning-range
011    [
012     if member? list pxcor pycor known-places
013     [
014      let dist distancexy my-current-x my-current-y
015      let pos position list pxcor pycor known-places
016      let food last item pos known-food
017
018      if food >= 0
019      [
020       ifelse dist = 0
021       [set ownFood (food / 2)]
022       [set sumFood sumFood + food / (2 ^ dist)]
023      ]
024      set count-known count-known + 1
025     ]
026    ]
027   set sumFood ownFood + SumFood
028   set sumFood sumFood / count-known
029
030   if (sumFood > 1)
031   [ set sumFood 1]
032
033   set bfoodg sumFood
034   end
```

Listing 38: Die to baseValueFoodForField-g

2. To survival-g

In dieser Prozedur wird zunächst überprüft, ob der Turtle einem Koordinator untergeordnet ist oder nicht (Zeilen 07 bis 010). Dies hat Einfluss auf die maximale Menge an Sugar, die er sammeln kann. Danach erfolgt die eigentliche Berechnung. Wie in Zeile 016 zu sehen ist, fließt hier auch der neue Wert von BFOODG mit ein. Siehe hierzu Listing 39.

```
01    to survival-g
02     set roundstosurvive 20
03     let tempErg 0
04     let temp 0
05     let resultsurvivaldef 0
06
07     ifelse subordinated = true
08     [ set temp (sugar + maxgather * (1 - taxes) / (metabolism *
09      roundstosurvive)) ]
010    [ set temp (sugar + maxgather / (metabolism * roundstosurvive)) ]
011
012    ifelse (temp > 1)
013    [ set tempErg 1 ]
014    [ set tempErg temp ]
015
016    set resultsurvivaldef ((0.95 * tempErg) + (0.1 * bfoodg))
017    ifelse (resultsurvivaldef > 1)
018    [ set survivalg 1]
019    [ set survivalg resultsurvivaldef]
020   end
```

Listing 39: Die to survival-g Prozedur

3. To saturation-g

In dieser Prozedur wird zunächst auch überprüft, ob der Turtle einem Koordinator untergeordnet ist. Danach wird dann mittels Arkustangens das Ergebnis berechnet.

```
01    to saturation-g
02       set avwealth 75
03       let foodmaxgather 0
04       ifelse subordinated = true
05          [set foodmaxgather (sugar + maxgather * (1 - taxes))]
06          [set foodmaxgather (sugar + maxgather) ]
07
08       set resultsaturationg 0
09       set resultsaturationg ((atan foodmaxgather avwealth) / 180)
010   end
```

Listing 40: Die to saturation-g Prozedur

4. To wealth-g

In der `to wealth-g` Prozedur werden die Werte `resultsaturationg` und `bfoodg` verwendet um zu berechnen, wie sich die Aktion Gather auf den Wohlstand auswirken würde. Die Prozedur ist im Listing 41 abgebildet.

```
01    to wealth-g
02       let resultwealthdef 0
03       set resultwealthdef ((0.9 * resultsaturationg) + (0.1 * bfoodg))
04       ifelse (resultwealthdef > 1)
05          [set wealthg 1]
06          [set wealthg resultwealthdef]
07
08    end
```

Listing 41: Die to wealth-g Prozedur

Nach der Berechnung der neuen Werte für Überleben und Wohlstand fließen alle benötigten Ergebnisse in die `to-report sat-gather` Prozedur ein. Hier wird wie gewohnt das Endergebnis für die Aktion berechnet. Siehe Listing 42.

```
01    to-report sat-gather
02     report ((survivalg * 0.5) + (wealthg * 0.1) + (breedingdef * 0.1) +
03     (influencedef * 0.2) + (curiositydef * 0.1))
04    end
```

Listing 42: Die to-report sat-gather Prozedur

3.3 Wichtige Prozeduren

In Kapitel 3.2. „Der Programmcode“ bin ich auf den Aufbau des Spielfeldes und die verwendeten Prozeduren zur Berechnung der auszuführenden Aktion eingegangen. In diesem Kapitel möchte ich einige Prozeduren vorstellen, die die tatsächlichen Aktionen beinhalten und den Turtle somit „handfeste“ Dinge ausführen lassen.
Beginnen werde ich jedoch mit der Entscheidungsfindung.

3.3.1 To decide

Im vorherigen Kapitel ist zu sehen, dass am Ende der Berechnungen immer eine `TO REPORT sat-XXX` Prozedur aufgerufen wird. Sie liefert jeweils das Endergebnis der Befriedigung der jeweiligen Aktion. Dieser Wert wird in der `TO DECIDE` Prozedur genutzt um zu berechnen, welche Aktion der Turtle als nächstes ausführen wird.
Zu Anfang der Prozedur wird überprüft, ob der Benutzer die Hierarchie aktiviert oder deaktiviert hat mittels Schieberegler enable-hierachy? auf der grafischen Benutzeroberfläche.
Ist die Hierarchie aktiviert, so wird zunächst eine Liste namens `SATISFACTIONS` angelegt. Diese Liste enthält die ganzen berechneten Endwerte (sat-gather, sat-move, etc.) aus Kapitel 3.2. .
Im zweiten Schritt wird von jedem Wert aus der `SATISFACTIONS` Liste der kleinste Wert abgezogen. Dies geschieht, da der Turtle die Befriedigung diesen Wertes auf jeden Fall erwarten kann, egal welche Aktion er ausführt. König schreibt hierzu *„Um den Effekt einer Bedürfnispyramide erzielen zu können (siehe 2.3.5) wird diese Sockelbefriedigung von allen Gesamtbefriedigungen abgezogen. Die so erhaltenen Werte werden normiert, so dass sie in der Summe eins ergeben. Diese normierten Werte entsprechen der Wahrscheinlichkeit, dass die entsprechende Aktion bei der abschliessenden Zufallsauswahl gezogen wird.“* [Kö2000, S. 28]. Genau so wird in der `TO DECIDE` Prozedur auch vorgegangen. Nachdem das Minimum von allen Werten abgezogen wurde, werden im nächsten Schritt die Werte jeweils durch die Summe aller Werte geteilt. Danach wird die Summe neu berechnet. Es wird

eine Zufallszahl zwischen 0,1 und 1,0 generiert. Diese wird dann mit der neu gebildeten Summe multipliziert. Im letzten Schritt der Prozedur wird dann überprüft, ob der Zufallswert kleiner ist als der erste Wert aus der Liste. Falls ja, wird die Position des Wertes in der Liste gespeichert in der Variable ACTION und die Prozedur wird mittels STOP Anweisung verlassen. Sollte die Zufallszahl nicht kleiner sein als der erste Wert aus der Liste, so werden der erste und zweite Wert addiert und erneut überprüft usw. .

Hat der Benutzer die Hierarchie deaktiviert, werden die gleichen Berechnungen ausgeführt, jedoch mit einer geänderten Liste. Diese kleinere Liste beinhaltet nur noch die Aktionen gather, move, breed und rest.
In jedem Fall wird am Schluss die auszuführende Aktion in der ACTION Variable gespeichert und in der Prozedur TO EXECUTION verwendet.

3.3.2 To execution

In dieser Prozedur wird mittels IF-Konstrukten überprüft, welchen Wert die Variable ACTION hat und dementsprechend der Turtle angewiesen, eine gewisse Aktion auszuführen. Dabei werden bei bestimmten Aktionen zuvor noch einige Parameter überprüft, wie im Ausschnitt in Listing 43 zu sehen ist.

```
01    if action = 3 and isCoordinator = false and subordinated = false
02    and enable-hierachy? = true
03      [
04        set isCoordinator true
05        set shape "triangle"
06        set probationStart ticks
07        stop
08      ]
```

Listing 43: Codeausschnitt der to execution Prozedur

Kann der Turtle die IF-Bedingungen seiner ausgewählten Aktion nicht erfüllen, so greift ein Sicherheitsmechanismus und er wird auf jeden Fall die Aktionen move und eat ausführen, damit er nicht stirbt.

3.3.3 To memory-fill

In dieser Prozedur wird das Gedächtnis des Turtles befüllt. Hier werden zwei Listen angelegt. In die eine Liste (`mem-pos`) werden die X/Y Koordinaten der Patches eingetragen, in die zweite Liste (`mem-ts`) werden der Tick, zu dem das Ganze aufgerufen wurde, und der Sugarwert des befragten Patches geschrieben. Um Redundanzen zu vermeiden, werden schon vorhandene Einträge in den Listen ersetzt durch neuere. In Listing 44 ist die komplette Prozedur abgebildet.

```
01    to memory-fill
02      let pxpy (list)
03      let ts (list)
04
05      ask patches in-radius vision
06      [
07        set pxpy lput list pxcor pycor pxpy
08        set ts lput list ticks psugar ts
09      ]
010
011     (foreach pxpy ts
012       [
013         ifelse member? ?1 mem-pos
014         [
015           let insert-position position ?1 mem-pos
016           set mem-pos replace-item insert-position mem-pos ?1
017
018           ifelse (any? turtles-on patch first ?1 last ?1 = true)
019           [ set mem-ts replace-item insert-position mem-ts list
020             first ?2 ((last ?2)  / 2) ]
021           [ set mem-ts replace-item insert-position mem-ts ?2 ]
022         ]
023         [
024           set mem-pos lput ?1 mem-pos
025           ifelse (any? turtles-on patch first ?1 last ?1 = true)
026           [ set mem-ts lput list first ?2 ((last ?2)  / 2) mem-ts]
027           [ set mem-ts lput ?2 mem-ts]
028         ]
029       ]
030     )
031   end
```

Listing 44: Die to memory-fill Prozedur

In den Zeilen 02 und 03 werden zunächst zwei temporäre Listen angelegt. Danach werden die Patches in der Sichtweite des Turtles aufgerufen. Diese tragen dann ihre Koordinaten, den Tick und den Sugarwert in die temporären Listen ein.

Danach werden die X/Y Koordinaten aus der temporären Liste überprüft, ob sie schon in der `mem-pos` Liste enthalten sind. Ist dem so, wird sich die Position in der Liste gemerkt, an der sie stehen (Zeile 015) und danach werden sie ersetzt (Zeile 016). Anschließend wird überprüft, ob schon ein Turtle auf dem Patch mit den X/Y Koordinaten sitzt. Wenn ja, wird auch hier der Eintrag in der `mem-ts` Liste ersetzt, und zwar mit dem aktuellen Tick und dem halbierten Sugarwert. Befindet sich kein Turtle auf dem angegebenen Patch, wird der komplette Sugarwert in die Liste geschrieben (Zeilen 018-021).

Stehen die X/Y Koordinaten noch nicht in der `mem-pos` Liste, so werden sie eingefügt und anschließend wird wieder ermittelt, ob ein Turtle auf dem Patch sitzt oder nicht und der Tick und der Sugarwert in die `mem-ts` Liste eingefügt (Zeilen 024-027).

3.3.4 To memory-delete

Diese Prozedur dient dazu, Informationen, die älter sind als der Wert von `MEMORY-LIFETIME`, aus dem Gedächtnis des Turtles zu löschen. Die komplette Prozedur ist im Listing 45 dargestellt.

```
01    to memory-delete
02      let memory-lifetime 40
03
04      (foreach mem-pos mem-ts
05        [
06          if (ticks - (first ?2)) >= memory-lifetime
07          [
08            let remove-position position ?2 mem-ts
09            set mem-pos remove-item remove-position mem-pos
010           set mem-ts remove-item remove-position mem-ts
011         ]
012       ]
013     )
014   end
```

Listing 45: Die to memory-delete Prozedur

Dazu wird in der IF-Abfrage überprüft, ob der aktuelle Tick minus den Wert aus der Liste größer gleich dem Wert von `MEMORY-LIFETIME` ist (Zeile 06). Trifft dies zu, so wird die Position, an der dieser Eintrag in der `mem-ts` Liste steht, in die Variable `REMOVE-POISITION` geschrieben. In den Zeilen 09 und 010 werden dann die Einträge in den beiden Listen an der entsprechenden Position gelöscht.

3.3.5 To reproduce

Diese Prozedur spiegelt das Fortpflanzen wider. Hier werden erst eine Reihe von Restriktionen überprüft (siehe Listing 46, Zeilen 03-04). Nur wenn alle eingehalten werden, kann der Turtle einen Nachkommen erschaffen. Die Nachkommen müssen mit den gleichen Variablen ausgestattet werden, die auch ihre Eltern besitzen. Dies erfolgt in den Zeilen 08 bis 015. Anschließend wird der Kinderzähler des Elternturtles um eins erhöht und die Befriedigung der Fortpflanzung auf 1 (also voll befriedigt) gesetzt (Zeilen 017-018).

```
01    to reproduce
02
03      if (children < 10) and (sugar > childfood) and ((ticks - birthday)
04      >= pubertyage)
05        [
06          set sugar (sugar - childfood)
07          hatch 1 [
08                     set sugar childfood
09                     set birthday ticks
010                    set life-expectancy random (101 - 50) + 50
011                    set maxgather 50
012                    set maxchildren 10
013                    set metabolism random (5 - 1) + 1
014                    set children 0
015                    set shape "circle"
016                 ]
017         set children (children + 1)
018         set breedingdef 1
019       ]
020   end
```

Listing 46: Die to reproduce Prozedur

3.3.6 To turtle-move

In dieser Prozedur wird das beste Feld für den Turtle ermittelt und anschließend begibt er sich dann dorthin, siehe Listing 47.
Zunächst wird in Zeile 05 überprüft, ob der Turtle koordiniert wird. Falls dies der Fall ist, fragt dieser Turtle seinen eingehenden `coord-to-sub` Link und nimmt sich die X/Y Koordinaten für das beste Feld heraus. Er überprüft dann, ob schon ein anderer Turtle auf diesem Patch sitzt (Zeile 013). Falls es frei ist, bewegt er sich dorthin. Ist der Turtle keinem Koordinator untergeordnet, hat er sich vorher in der `baseValueFoodForField-m` Prozedur sein eigenes bestes Feld ausgearbeitet. Aber auch hier überprüft er kurz, ob nicht schon jemand dieses Feld besetzt (Zeile 017). Wenn es frei ist, geht er auf diesen Patch.

```
01    to turtle-move
02      let tempx 0
03      let tempy 0
04
05      ifelse subordinated = true
06      [
07        ask my-in-coord-to-sub
08        [
09          set tempx targetx
010         set tempy targety
011       ]
012
013       if not any? other turtles-on patch tempx tempy
014       [setxy tempx tempy]
015     ]
016     [
017       if not any? turtles-on patch bestfieldx bestfieldy
018       [ setxy bestfieldx bestfieldy ]
019     ]
020   end
```

Listing 47: Die to turtle-move Prozedur

3.3.7 To turtle-eat

Die to turtle-eat Prozedur steuert die Nahrungsaufnahme der Turtles. Hier wird zunächst überprüft, ob der Turtle einem Koordinator untergeordnet ist. Trifft dies zu, so gehen Steuern von seiner aufgenommenen Nahrung ab, die dann später vom Koordinator eingesammelt werden.

Falls der aktuelle Turtle ein Koordinator ist, kommt in dieser Prozedur sein erhöhter Stoffwechsel ins Spiel.

Ist der Turtle ein ganz normaler Agent, der weder Koordinator noch untergeordnet ist, so konsumiert er einfach entweder so viel wie der aktuelle Patch hergibt, oder so viel wie ihm noch fehlt, um seinen maximalen Nahrungsvorrat pro Runde zu erreichen.

3.3.8 To coordinate

Bevor ich die Prozedur beschreibe, möchte ich zunächst die Verbindungen (Links), die zwischen den Koordinatoren und ihren untergeordneten Turtles existieren, erklären. Wenn sich ein Turtle einem Koordinator unterordnet, erstellt er einen `SUBLINK`, der von ihm aus zum Koordinator geht. In diesen Link werden einige Informationen geschrieben, unter anderem auch, ob der Link aktiv ist oder nicht (`active 0/1`). Der Koordinator hingegen erstellt jede Runde einen neuen `COORDLINK` zu seinen untergeordneten Turtles, in die er die für diese Runde besten X/Y Koordinaten einträgt. In der nächsten Runde vernichtet er zunächst den alten Link und erstellt dann wieder einen neuen Link mit neuen Koordinaten.

In dieser Prozedur überprüft der Koordinator zunächst all seine aktiven untergeordneten Verbindungen und arbeitet für jede dieser Verbindungen ein bestes Feld aus. Dem liegt eine Punktbewertung zu Grunde, die von Andreas König übernommen wurde (siehe Kapitel 2.3.4 „Koordinierung beginnen“). Sobald das Feld ausgearbeitet ist, schreibt der Koordinator die Koordinaten in die `COORDLINK` Verbindung, die von ihm zum untergeordneten Turtle geht. Das folgende Listing 48 beinhaltet die Überprüfung der aktiven untergeordneten Links (Zeilen

02 bis 05) und am Schluss die Erstellung eines neuen COORDLINK, in den die X/Y Koordinaten des für diesen Turtle besten Feldes eingetragen werden (Zeilen 08-09).

```
01    ...
02    ask my-in-sub-to-coord with [active = 1]
03      [
04        set tempRange range
05        ask other-end [set mySubordinate lput who mySubordinate]
06      ]
07    ...
08    create-coordlink-to turtle ?
09    [ set targetx temptargetx set targety temptargety]
```

Listing 48: Codeausschnitt aus der to coordinate Prozedur

3.3.9 To collectTaxes

Diese Prozedur dient der Steuereintreibung für die Koordinatoren. Wie schon in Kapitel 2.3.6 „Unterordnen“ beschrieben, muss jeder koordinierte Turtle jede Runde einen Teil seiner Nahrung an seinen Koordinator abgeben. Die to collectTaxes Prozedur macht genau dies. Sie wird nur von Koordinatoren aufgerufen. Wie in Listing 49 (Zeilen 01 bis 03) zu sehen ist, prüft der Koordinator zuerst all seine aktiven SUBLINKS und weist die untergeordneten Turtles an, ihre ID in die Liste mySubordinate zu schreiben. Danach geht er die Liste durch und weist die jeweiligen Turtles an, ihren Wert aus der Variable TAXES

```
01     ask my-in-sub-to-coord with [active = 1]
02      [
03        ask other-end [set mySubordinate lput who mySubordinate]
04      ]
05
06      foreach mySubordinate
07      [
08        ask turtle ?
09        [ set sumSugar sumSugar + taxes ]
010     ]
011     set sugar sugar + sumSugar
```

Listing 49: Codeausschnitt aus der to collectTaxes Prozedur

(wird berechnet in to turtle-eat) aufzusummieren. Nachdem alle untergeordneten Turtles ihre Steuern aufsummiert haben, addiert der Koordinator in Zeile 011 diese Einnahmen zu seinem Nahrungsvorrat dazu.

3.3.10 To endCoordinate

Diese Prozedur (genau wie `to endSubordinate`) darf nur ausgeführt werden, wenn die Sperrfrist[4] (`PROBATIONTIME`) abgelaufen ist. Besteht die Beziehung länger als die Sperrfrist, kann der Koordinator sich dazu entschließen seine Tätigkeit einzustellen. Dazu ermittelt er zunächst wieder all seine aktiven untergeordneten Turtles. Er veranlasst diese dann dazu, in ihre `SUBLINKS` eine Bewertung seiner Arbeit zu schreiben und die Verbindung inaktiv zu setzen (`active 0`), siehe Listing 50 Zeilen 07-011. Danach setzen die untergeordneten Turtles ihren Status auf „normal“ zurück und nehmen auch die normale Form an (Zeilen 013-015). Anschließend zerstört der Koordinator alle seine `COORDLINKS`, und setzt seinen Status ebenfalls wieder auf „normal“ zurück und nimmt auch die Kreisform wieder an (Zeilen 018-021).

[4] Siehe Kapitel 2.3.7 „Unterordnung beenden“

```
01    foreach mySubordinate
02      [
03        ask turtle ?
04        [
05          set tempSugar sugar
06
07          ask my-out-sub-to-coord
08          [
09            set current-value tempSugar
010           set coord-value (current-value - start-value)
011           set active 0
012         ]
013        set subordinated false
014        set duration-sublinks 0
015        set shape "circle"
016       ]
017     ]
018     ask my-out-coord-to-sub
019     [die]
020     set isCoordinator false
021     set shape "circle"
```

Listing 50: Codeausschnitt aus der to endCoordinate Prozedur

3.3.11 To subordinate

In dieser Prozedur ordnet sich ein Turtle einem Koordinator unter. Dazu überprüft er zunächst, ob er über alte bestehende Links verfügt, die inaktiv sind. Falls dies der Fall ist, sucht er sich den Koordinator, aus dem er damals die beste Bewertung gegeben hat. Er fragt nun diesen Turtle, ob er noch immer Koordinator ist. Wenn dem so ist, erstellt der untergeordnete Turtle einen `SUBLINK` zu dem Koordinator, schreibt ein paar Variablen in den Link und setzt seinen Status auf untergeordnet.

Falls der Turtle jedoch über keine alten Links verfügt (also noch nie koordiniert wurde), schaut er sich in seiner Umgebung nach Koordinatoren um. Findet er einen, so ordnet er sich diesem unter und erstellt ebenfalls einen `SUBLINK`, schreibt einige Variablen hinein und setzt seinen Status auf untergeordnet (`subordinated true`).

3.3.12 To endSubordinate

Wie schon bei der `to endCoordinate` Prozedur wird auch hier zunächst überprüft, ob die Sperrfrist bereits überschritten wurde oder nicht. Siehe Listing 51, Zeilen 03-04. In den Zeilen 010 bis 014 wird dann der aktive `SUBLINK` aufgerufen und auf inaktiv gesetzt sowie die Bewertung des Koordinators vorgenommen. Anschließend setzt der Turtle seinen Status auf „normal“ (Zeile 017) und nimmt die Kreisform wieder an (Zeile 019).

```
01    to endSubordinate
02
03      if (ticks - probationStart < probationTime)
04      [stop]
05
06      if isCoordinator = true
07      [stop]
08
09      let tempSugar sugar
010     ask my-out-sub-to-coord with [active = 1]
011     [
012       set active 0
013       set current-value tempSugar
014       set coord-value (current-value - start-value)
015     ]
016
017     set subordinated false
018     set duration-sublinks 0
019     set shape "circle"
020   end
```

Listing 51: Die to endSubordinate Prozedur

4 Ergebnisse

In diesem Kapitel stelle ich die drei Simulationen, die Andreas König in seiner Diplomarbeit behandelt hat, nach und vergleiche seine Ergebnisse mit denen meiner Simulationen. Es ist zu beachten, dass nicht alle Funktionen von Andreas König exakt repliziert werden konnten. Daher kann es durchaus zu unterschiedlichen Ergebnissen kommen.

4.1 Freie Marktwirtschaft vs. Nachhaltigkeit

Bei diesem Modell sieht die Konfiguration wie folgt aus in NetLogo:

– Initial-population 15
– Initial-food 25
– enable-hierachy? Off
– puberty-age 10
– max-children 99
– die-of-old-age? Off
– random-seed? Off
– patch-fertile-percentage 75%
– sugar-regrow 0,5
– max-psugar 40
– maxgather 50 / random zwischen 10 und 100
– gatherrest 10 / random zwischen 0 und 20

Das Referenzmodell (siehe Abbildung 5, orange Linie) ist also komplett statisch eingestellt. Die Turtles essen soviel Sugar von einem Patch, bis nur noch 10 Einheiten vorhanden sind. Diese lassen sie jedoch unangetastet. Da der Sugar nur um 0,5 Einheiten pro Runde nachwächst, bedarf es einiger Runden, bevor es sich für einen Turtle wieder lohnt, auf das gleiche Feld zu gehen. Dadurch wirtschaften die Turtles zwar sehr nachhaltig, aber im Gegenzug können nur ungefähr 40 – 50 Turtles zur selben Zeit auf dem Spielfeld existieren.

Bei der zweiten Simulation (siehe Abbildung 5, blaue Linie) kann der Wert GATHERREST mutieren. Dadurch kommt es zu einer unterschiedlichen Behandlung der Patches. Während der eine Turtle z.B. einen GATHERREST Wert von 10 errechnet hat, gilt für einen anderen z.B. ein Wert von 20. Durch die Mutation der Werte haben es die Turtles mit einem hohen GATHERREST Wert viel schwieriger, da die Turtles mit niedrigen GATHERREST Werten die Patches unter deren Minimum leer essen und sie so länger warten müssen, bis sie wieder Nahrung konsumieren können. Zusätzlich ist die Mutation in diesem Szenario sehr gefährlich, da der Sugar auf den Patches nur ab einem Wert von >= 5 nachwächst. Isst ein Turtle also soviel Sugar von einem Patch, dass dessen Wert unter 5 fällt, so gilt das Feld als „verdorrt“ und es wächst kein Sugar mehr nach.

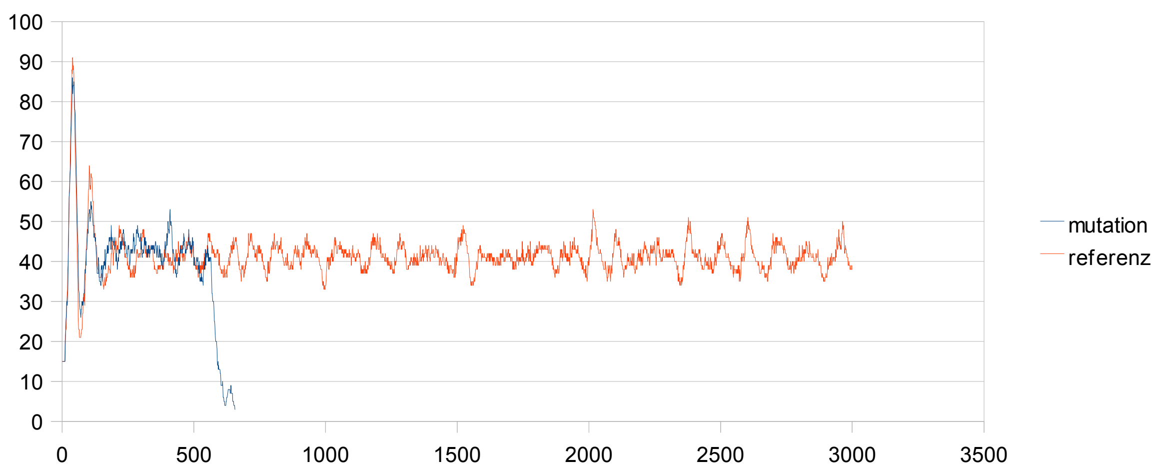

Abbildung 5: Simulationsergebnis aus NetLogo

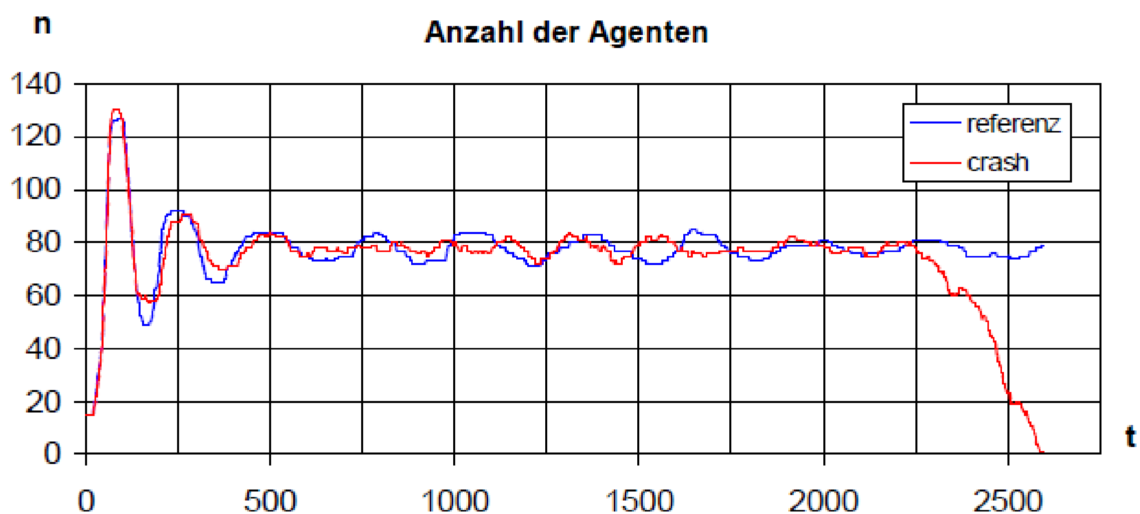

Abbildung 6: Simulationsergebnis aus [Kö2000, S. 75]

In Abbildung 6 ist das Ergebnis von Andreas König dargestellt. Hier ist zu sehen, dass die Population der Turtles im Modell mit Mutationen bei ihm erst nach ungefähr 2600 Ticks komplett aussterben. Der durchschnittliche GATHERREST Wert sinkt bei seiner Population langsamer, deswegen existiert sie länger, stirbt aber am Ende ebenfalls komplett aus. Ein weiterer Unterschied zwischen den Modellen von mir und Andreas König ist, dass die Populationen in seiner Simulation am Anfang bis auf ca. 130 anwachsen und sich dann später im Bereich von ~80 einpendeln. In meiner Simulation ist solch ein Amplitudencharakter am Anfang auch zu sehen, allerdings liegt hier das Bevölkerungsmaximum bei 90 und pendelt sich im Laufe der Simulation bei ~45 ein.

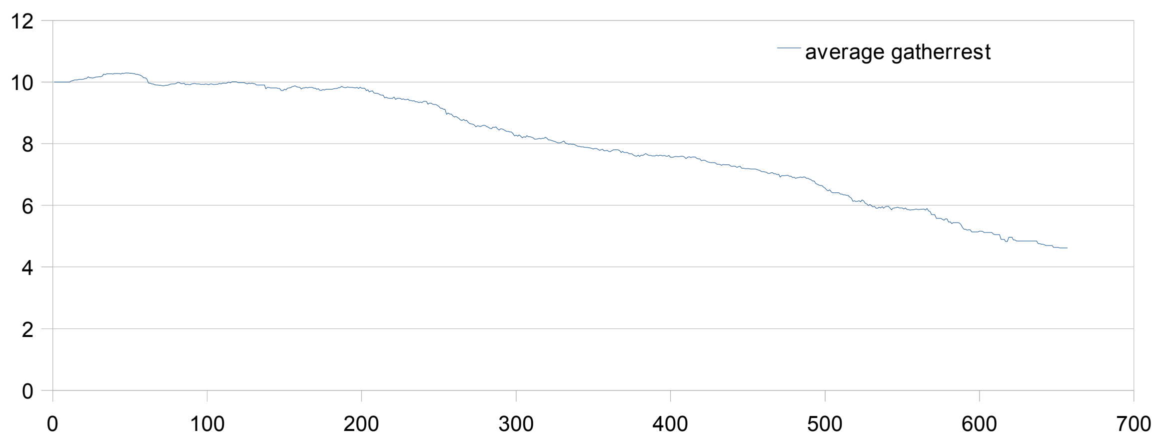

Abbildung 7: Simulationsergebnis aus NetLogo

4.2 Hierarchie und Nachhaltigkeit

In diesem Szenario wird deutlich, wie lohnend bzw. überlebenswichtig es ist, sich unterzuordnen. Die Konfiguration in NetLogo sieht wie folgt aus:

- growth-rate 0,05
- probationtime 100
- childfood 100
- birthrecreationfactor 0,999
- enable-hierachy? false / true
- patch-fertile-percentage 100%
- initial-population 300
- initial-food 25
- pubertyage 10
- max-children 100
- gatherrest 0

Bei dieser Simulation ist jeder Patch des Spielfeldes fruchtbar. Dies stellt für die Turtles am Beginn quasi das perfekte Spielfeld dar. Zugleich dürfen sie auch noch so viel Sugar konsumieren wie sie finden können (gatherrest 0). Außerdem sind in dem Szenario, in dem die Hierarchiebildung aktiviert ist, die Beziehungen auf Stabilität ausgelegt (probationtime 100). Das ist auch nötig, da der Sugar auf den Patches nur um 0,05 Einheiten pro Runde nachwächst.

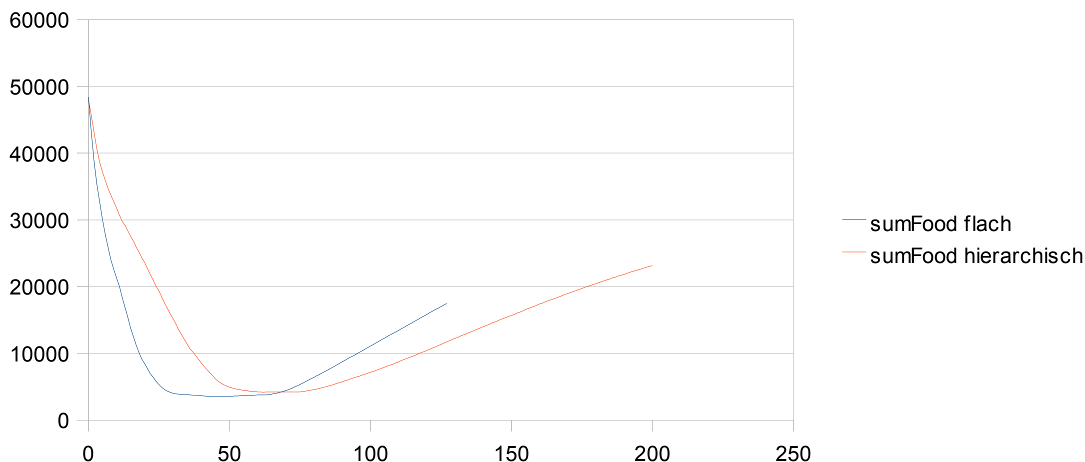

Abbildung 8: Durchschnittlicher Sugar auf dem Spielfeld

Bei dem Nachwuchs wird hier Wert gelegt auf eine sehr gute Grundausstattung an Nahrung, die bei der Geburt übergeben wird (childfood 100).

Der Verlauf der beiden Simulationen (enable-hierachy? false bzw. true) zeigt in diesem Szenario deutlich, dass eine Population ohne Koordinierung nicht auf Dauer überlebensfähig ist.

In Abbildung 9 (blaue Linie) ist zu sehen, dass die Bevölkerung ohne Hierarchie bereits nach ungefähr 130 Ticks komplett ausgestorben ist. Ab dem Beginn der Simulation sammeln die Turtles hier ungebremst soviel Sugar wie sie können, was zu einer sehr raschen Verknappung führt (siehe Abbildung 8, blaue Linie). Obwohl der Sugar ab dem 60. Tick wieder zu steigen beginnt, schafft es die Bevölkerung nicht zu überleben. Bei der Simulation mit aktivierter Hierarchie hingegen ist in Abbildung 8 (orange Linie) deutlich zu sehen, dass die Turtles hier den Sugar nicht ganz so schnell abernten wie in der Version ohne Hierarchie. So erreicht der Sugar hier erst in Tick 50 sein Minimum, wohingegen in der anderen Simulation dieser Wert schon im 30. Tick erreicht wurde. In der Abbildung 9 (orange Linie) ist zu sehen, wie sich die Bevölkerung verhält. In meinen Augen liegt hier der Schlüssel zum Erfolg. Da die Population hier am Anfang nicht so stark ansteigt wie die der unkoordinierten Bevölkerung, wird der Sugar langsamer abgebaut. Das bedeutet, dass sich der Sugar auf mehreren Feldern erholt und früher wieder genug Nahrung zum Überleben bieten kann.

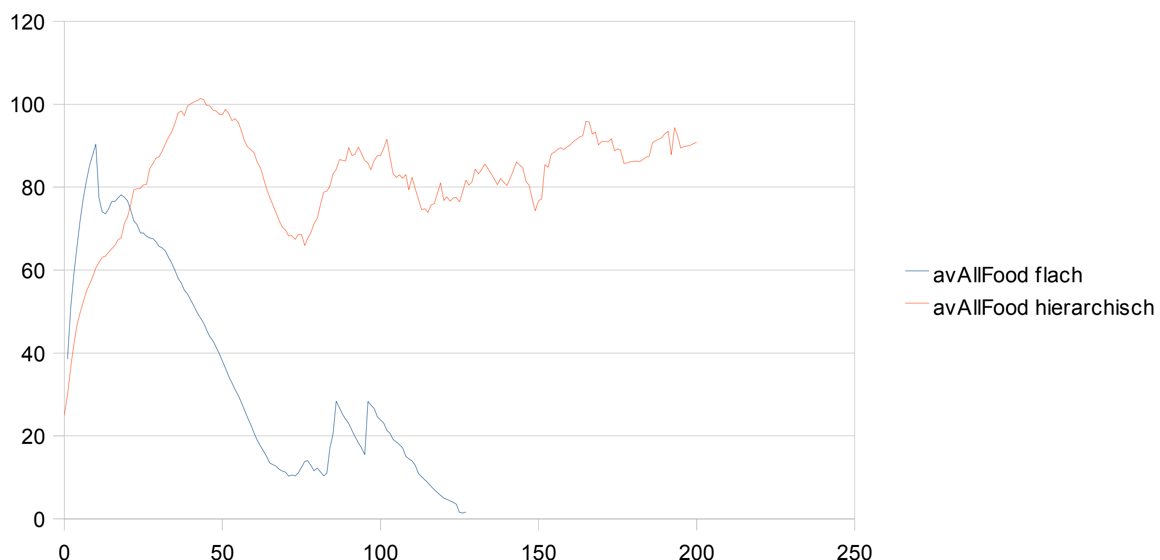

Abbildung 9: Durchschnittlicher Sugar aller Turtles auf dem Spielfeld

Dies erzeugt einen „weicheren Fall“ im weiteren Verlauf der Simulation, so dass sich ab dem 150. Tick die Bevölkerungsanzahl bei ca. 40 einpendelt. Abbildung 10 zeigt den durchschnittlichen Sugar der gesamten Bevölkerung an. Hier ist sehr gut zu erkennen, dass sich das Unterordnen/Koordinieren auszahlt. Während sich bei der hierarchischen Bevölkerung der Wert zwischen 60 und 100 bewegt, steigt der Wert bei der unkoordinierten Bevölkerung am Anfang kurz auf ca. 90 an und fällt dann rapide auf ca. 16. Danach gibt es noch zwei kurze Steigungen und schließlich sinkt er auf 0 ab.

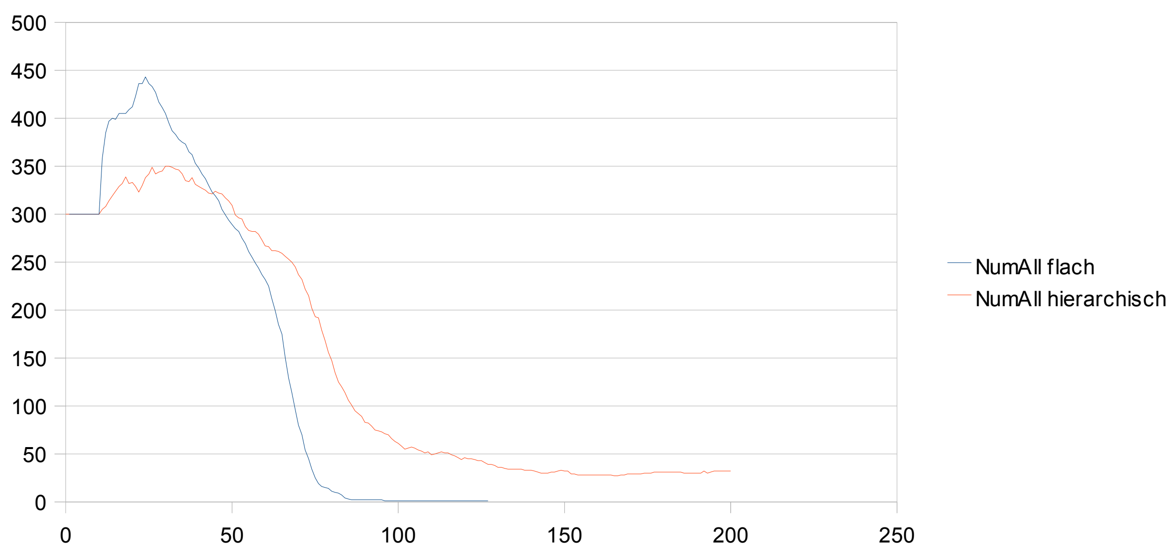

Abbildung 10: Anzahl an Turtles

Die folgenden drei Abbildungen (11,12,13) zeigen die Ergebnisse von Andreas König zu diesem Modell.

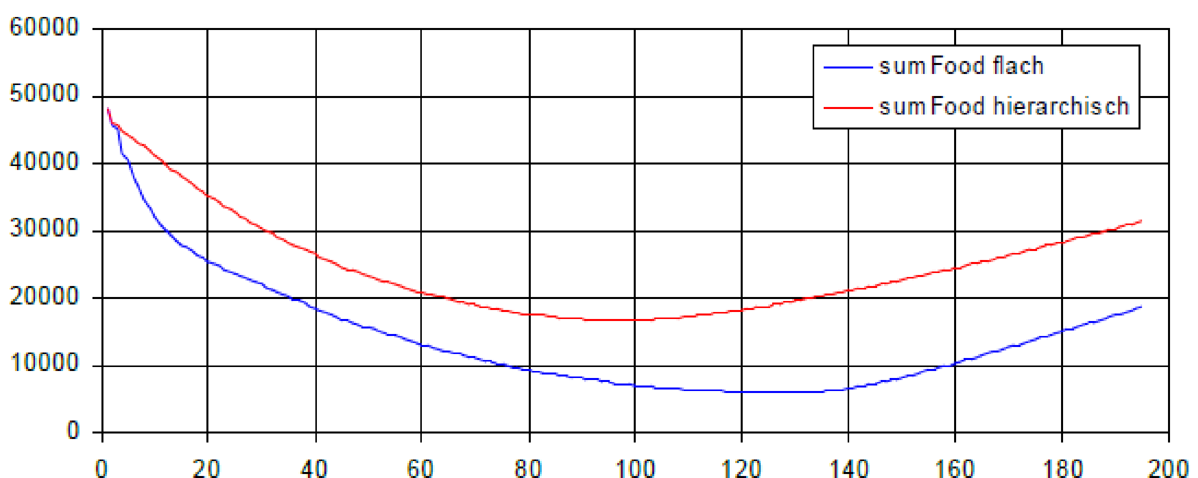

Abbildung 11: Ergebnis des Szenarios aus [Kö2000, S. 81]

Wenn man unsere Ergebnisse miteinander vergleicht stellt man fest, dass sich die Werte ein wenig unterschiedlich entwickeln, im Prinzip aber den gleichen Ausgang haben. Qualitativ gelangen wir beide zu ungefähr denselben Ergebnissen. Es scheint aber so, als ob NetLogo schneller läuft und so quantitative Änderungen schneller sichtbar werden.

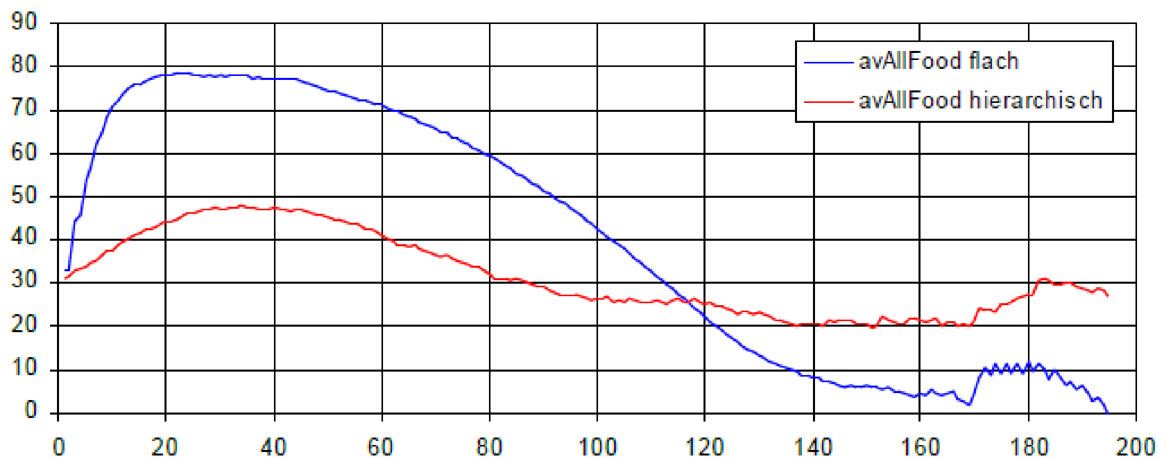

Abbildung 12: Ergebnis des Szenarios aus [Kö2000, S. 82]

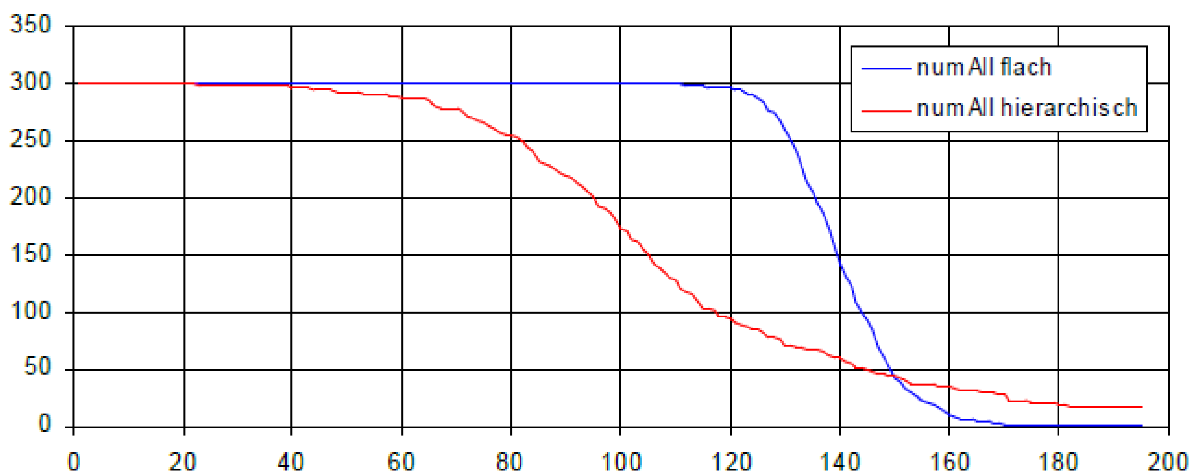

Abbildung 13: Ergebnis des Szenarios aus [Kö2000, S. 82]

4.3 Hierarchie im Härtetest

In diesem Szenario stellt der einzige Unterschied die Hierarchie dar. Die Konfiguration sieht wie folgt aus:

- patch-fertile-percentage 20%
- growth-rate 0,75
- metabolism mutable 0.1 (1-3)
- vision mutable 0.1 (1-5)
- planning-range mutable 0.1 (2-6)
- survival weight mutable 0.1 (0-1)
- breeding weight mutable 0.1 (0-1)
- influence weight mutable 0.1 (0-1)
- curiosity weight mutable 0.1 (0-1)
- wealth weight mutable 0.1 (0-1)
- enable-hierachy? false/true

Das Spielfeld für diese Simulation beträgt nur 20x30 Felder und ist somit recht klein gehalten. Bei beiden Modellen sind einige Werte zur Mutation frei gegeben, dabei sind die Grenzen in den beiden Varianten gleich. Lediglich die Hierarchie ist bei der einen Simulation aktiviert und bei der anderen deaktiviert. Die Simulationen liefen über 10.000 Schritte.

Das Ergebnis ist, dass die hierarchische Bevölkerung etwas unter der flachen Bevölkerung liegt (siehe Abbildung 14, blaue Linie).

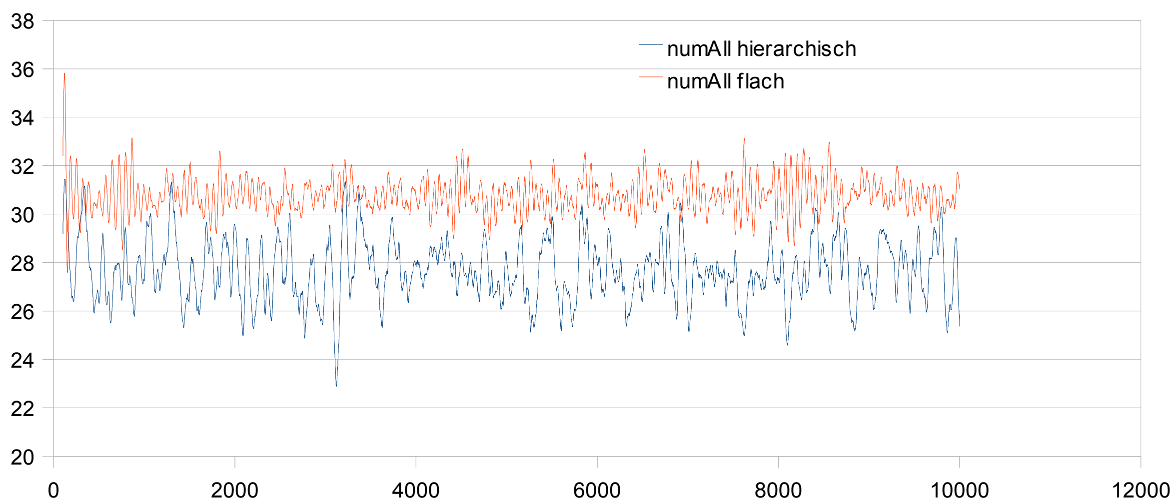

Abbildung 14: Anzahl der Turtles

Die Bevölkerung ohne Hierarchie steigt hier wie im vorherigen Experiment am Anfang wieder rasant an, fällt dann jedoch schnell wieder ab. Da in dieser Simulation der Sugar allerdings mit 0,75 Einheiten pro Runde nachwächst anstatt mit nur 0,05, ist die flache Bevölkerung jedoch in der Lage, weiter zu existieren und pendelt sich zwischen 20 und 45 Turtles ein. Bei der hierarchischen Bevölkerung ist zu erkennen, dass es am Beginn der Simulation nur einen vergleichsweise geringen Anstieg der Population gibt und sich das Ganze sehr schnell zwischen 15 und 40 Turtles einpendelt. Die Population verhält sich eher moderater als die flache Bevölkerung.
Bei den Werten metabolism und vision-range (Abbildungen 15 und 16), ist dagegen kaum ein Unterschied auszumachen. Bei beiden Bevölkerungen bewegt sich der Stoffwechsel zwischen 1,9 und 2,03 (Abbildung 15) und die Sichtweite zwischen 2,95 und 3,1. Hier findet nicht so viel Veränderung statt wie in der Simulation von Andreas König (siehe Abbildungen 17,18,19). Dies ist eventuell zurück zu führen auf verschiedene interne Berechnungen (Java <> NetLogo) bei der Mutation.

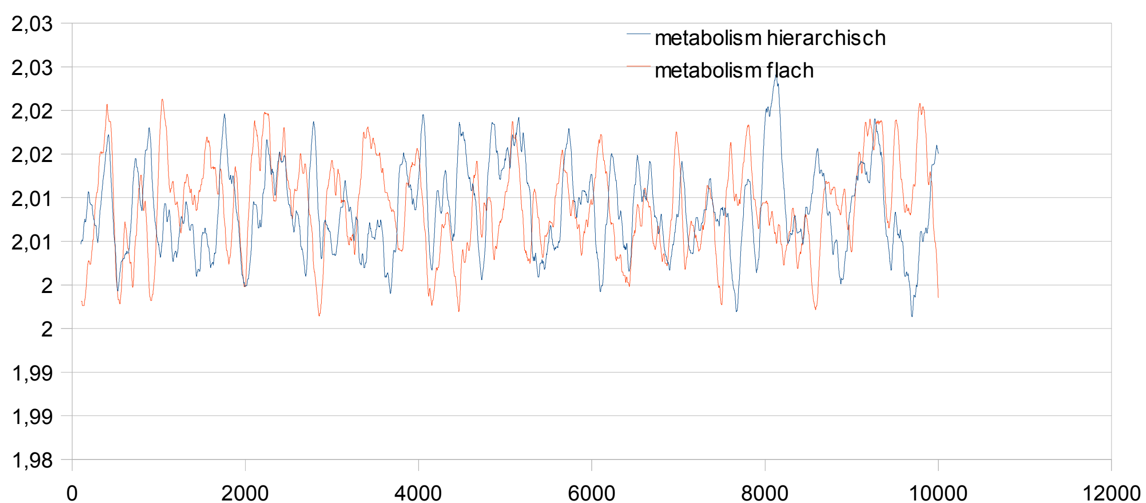

Abbildung 15: Durchschnittlicher Stoffwechsel

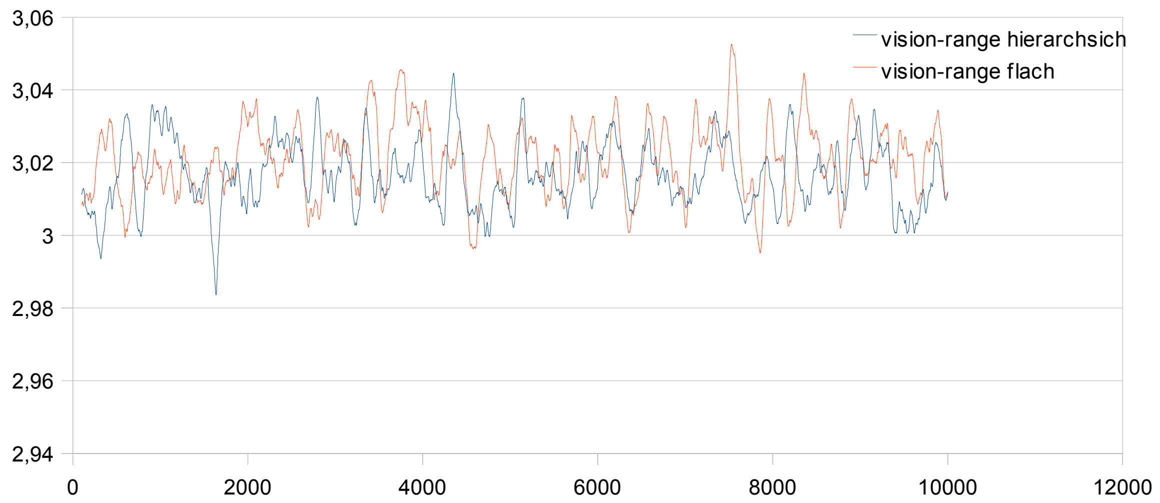

Abbildung 16: Durchschnittliche Sichtweite

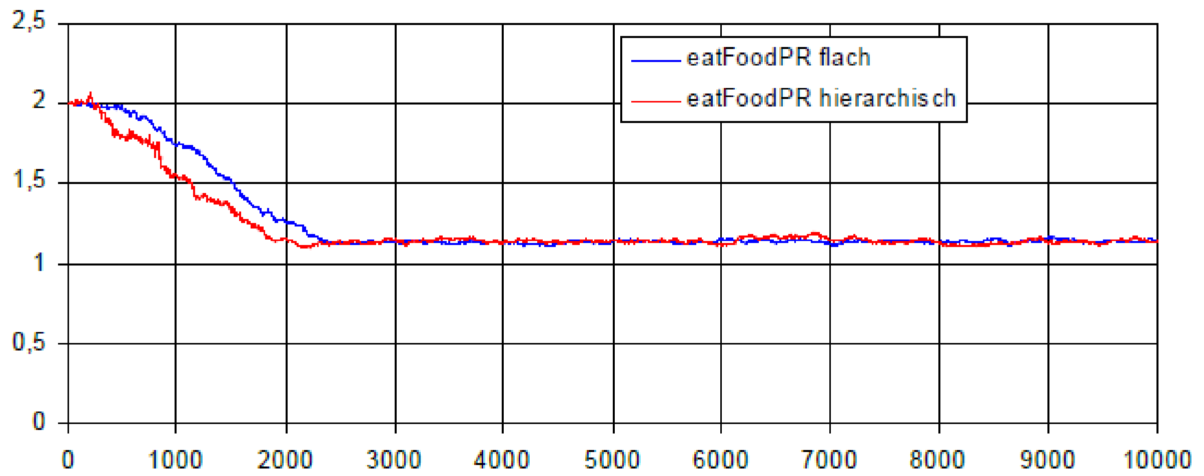

Abbildung 17: Ergebnis des Szenarios aus [Kö2000, S. 78]

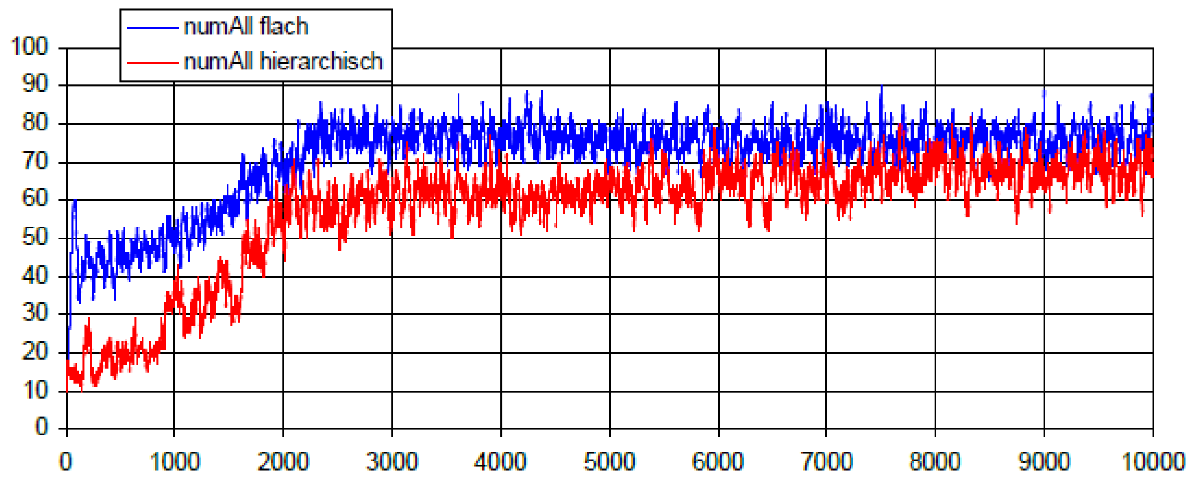

Abbildung 18: Ergebnis des Szenarios aus [Kö2000, S. 78]

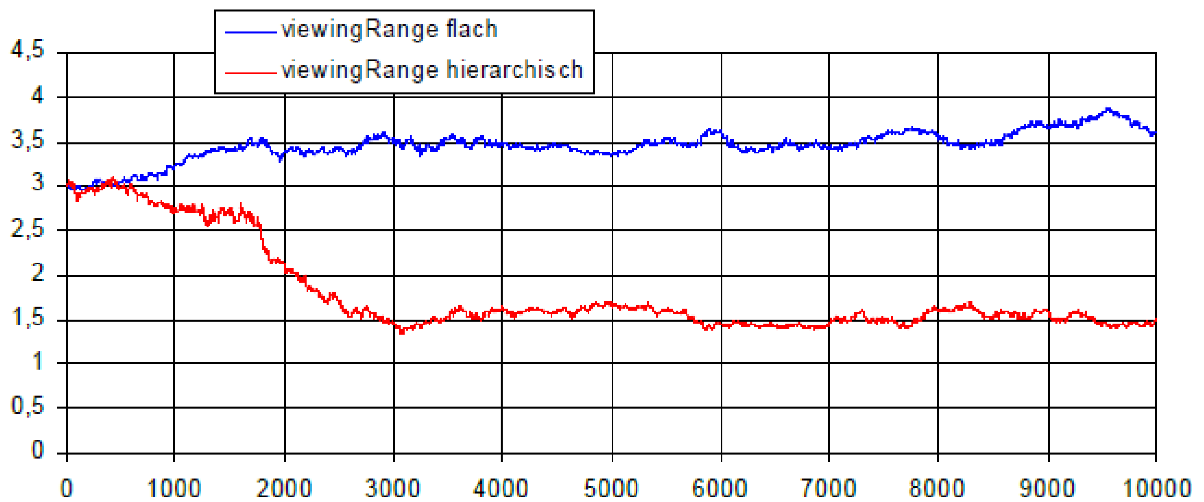

Abbildung 19: Ergebnis des Szenarios aus [Kö2000, S. 80]

5 Fazit

Nach einigen Monaten Entwicklungszeit ist es mir gelungen, eine sehr gute Replikation der Arbeit von Andreas König zu erstellen. Wie in Kapitel 4 zu sehen ist, gleichen sich unsere Ergebnisse bei 2 von 3 Experimenten stark. Ich bin überzeugt davon, dass man mit genügend Zeit und Verständnis auch noch das dritte Experiment zur Übereinstimmung führen kann. Der Weg bis zu den Ergebnissen der Experimente war jedoch lang und anstrengend. Da Java eine vollwertige Programmiersprache ist und man hier alle Freiheiten hat (Klassen, Funktionen, Prozeduren, Vererbung etc.), ist es sehr schwierig für einen Außenstehenden, sich in das vorhandene Programm einzuarbeiten. Da Andreas König ein begabter Programmierer ist, hat er sich die Vorteile von Java zu eigen gemacht und sein Programm z.B. über mehrere Klassen verteilt. Dies ist zwar aus programmiertechnischer Sicht sehr gut, aber für einen Dritten, der sich einarbeiten muss, sehr schwer nachzuvollziehen. NetLogo dagegen ist einfacher zu verstehen, schon aufgrund der Tatsache, dass man nur eine Datei hat, in welcher der komplette Code ist. Ich bin der Meinung, dass man den Code in NetLogo schneller und einfacher verstehen kann und dass diese Entwicklungsumgebung weniger Einarbeitungszeit in Anspruch nimmt als eine traditionelle Programmiersprache wie z.B. Java.

Ein weitere Schwierigkeit war, dass Andreas König in seiner Diplomarbeit nur auf die mathematischen Formeln, die er zur Berechnung genutzt hat, eingegangen ist, nicht aber auf seinen Code. In meiner Arbeit war es mir deshalb wichtig, den Code und besonders die wichtigen Prozeduren zu erklären, so dass ein Dritter in einer angemessenen Zeit dazu in der Lage ist, das Programm zu interpretieren.

Die Arbeit mit dieser Simulation war für mich sehr interessant und hat mir aufgezeigt, wie schnell ein sozialwissenschaftliches Modell doch komplex werden kann. Dies merkt man auch, wenn man die NetLogo Simulation mit einer hohen Anzahl an Turtles laufen lässt. Dann dauert die Berechnung eines einzigen Ticks schon mehrere Minuten.

Abschließend möchte ich mich hier klar für NetLogo aussprechen. Diese Entwicklungsumgebung stellt für mich eine sehr gute Lösung zur Simulierung von sozialwissenschaftlichen Modellen dar. NetLogo bietet einen schnellen Einstieg und auch viele grafische Elemente (z.B. das Anzeigen von Links zwischen Turtles inklusive deren Richtung) ohne großen Aufwand. In Java hingegen müsste man, nachdem man die Logik programmiert hat, noch aufwendig eine grafische Oberfläche entwickeln. Diese Arbeit nimmt einem NetLogo komplett ab. Des weiteren bietet NetLogo eine große Sammlung an Beispielszenarien, in denen beinahe alle Funktionen abgedeckt werden, so dass man schnell versteht, welche Funktion man am besten für welches Szenario verwendet.

5.1 Ausblick

Die Auswahl der Aktionen eines Turtles in diesem Modell entsprechen der Bedürfnispyramide von Andreas König und es werden rein mathematische bzw. ökonomische Aspekte berücksichtigt. Diese Pyramide eignet sich für diese Simulation sehr gut, ist aber für andere eher weniger geeignet, bzw. müsste erweitert/verändert werden. An der Universität Kassel haben sich einige Leute vom Center for Environmental Systems Research eine Architektur überlegt, mit der es möglich sein soll, die Lücke zwischen ABM[5] toolkits (z.B. NetLogo), die keine eingebauten psychologischen Funktionen bieten, und vollwertigen kognitiven Architekturen, die eine Reihe bedeutender Funktionen haben, zu schließen [EEKHK09]. Ihr Projekt nennt sich LARA (Lightweight Architecture for boundedly Rational citizen Agents). Bei dieser Architektur wird die Wahrnehmung eines Agenten umgewandelt in subjektive Attribute, die in einem Gedächtnis gespeichert werden. Die Wahrnehmung beinhaltet dabei biophysische, sozioökonomische und soziale Umweltparameter. Die Agenten sollen dann Entscheidungen treffen, die der Situation angepasst sind (z.B. Gewohnheitsentscheidungen oder Blitzentscheidungen, die auf heuristischen Verfahren beruhen). Interessant wäre es zu sehen, wie

[5] Agent Based Modelling

sich die Agenten aus unserer Simulation verhalten, wenn diese Architektur zu Grunde liegt.

Das Verhalten der Agenten in meiner Simulation, besonders bei aktivierter Hierarchie, spiegelt auch das Verhalten von gewissen Geschäftsszenarien aus der Wirtschaft wider. So könnte man Koordinatoren als Consultants ansehen und die untergeordneten Turtles als deren Kunden. Gilt eine hohe Probationtime (Vertragslaufzeit), so benötigt ein Consultant eher wenige Kunden. Kommt es aber nur zu kurzen Vertragslaufzeiten zwischen den Geschäftspartnern, so muss jeder Consultant versuchen, mehrere Kunden zu akquirieren. Dadurch entsteht ein Kampf am Markt um den Kunden. Mithilfe einer solchen Simulation könnten Firmen eventuell ihre Strategien optimieren. Daher bin ich der Meinung, dass Simulationen in Zukunft auch häufiger in Unternehmen verwendet werden sollten, um geplante Entscheidungen zu untersuchen. So könnte z.B. das Verhalten des Marktes auf spezielle Preisveränderungen etc. simuliert werden. Die Möglichkeiten und das Potential sind hier riesig.

Literatur

[Kö2000] König, A.: Eine Multi-Agenten-Simulationsumgebung zur Untersuchung von Hierarchiebildung und Nachhaltigkeit, Diplomarbeit, Universität Koblenz-Landau 2000.

[EEKHK09] Elbers, M., Ernst A., Krebs F., Holzhauser S., Klemm D: LARA: A Lightweight Architecture for boundedly Rational citizen Agents, 6th European Social Simulation Association Conference, Guildford, UK, 2009.

[Kg08] Kirchgässner, G.: Homo oeconomicus : das ökonomische Modell individuellen Verhaltens und seine Anwendung in den Wirtschafts- und Sozialwissenschaften. 3., erg. Und erw. Auflage, Tübingen, 2008.

[EA96] Epstein J.M., Axtell R.: Growing artificial societies: social science from the bottom up, First Edition, 1996

[UOL11] University of Leicester, Interdisciplinary Science, The Centre for Research., URL:http://www2.le.ac.uk/departments/interdisciplinary-science/research/the-sugarscape , Abruf: 23.05.2011

[Wi09] Iain Weaver: NetLogo User Community Models: Sugarscape http://ccl.northwestern.edu/netlogo/models/community/Sugarscape Abruf: 24.05.2011

[Wp11] Lotka-Volterra-Regeln, URL: http://de.wikipedia.org/wiki/Lotka-Volterra-Regeln , Abruf: 26.05.2011

[Hg68] Hardin, G.: The Tragedy of the Commons, Science VOL. 162, Seiten 1243-1248, 1968

[Ln11] NetLogo, a multi-agent programmable modelin environment, URL: http://ccl.northwestern.edu/netlogo/ , Abruf: 02.07.2011

[Oe99] Ostrom E.: Die Verfassung der Allmende – jenseits von Staat und Markt, Mohr Siebeck, 1999

[GT05] Gilbert N.G., Troitzsch K.G.: Simulation for the social scientist, 2. Auflage, Open University Press, 2005