Darwin digital: Die Evolution von Komplexität*)

Klaus Manhart

München, Januar 2005

*) Populärwissenschaftliche, deutschsprachige Fassung von Lenski, Richard u.a.: The Evolutionary Origin of Complex Features, erschienen in: Nature 423, Mai 2003, S.139-143

Zusammenfassung

Die Entstehung komplizierter biologischer Organe ist unter Evolutionsforschern heftig umstritten. Reicht hier Darwins Theorie aus oder braucht es zusätzliche Annahmen? Eine aufsehenerregende Computersimulation beweist, dass kleine, scheinbar belanglose Änderungen über Tausende von Generationen zur Ausbildung hochkomplexer Eigenschaften führen können.

© Klaus Manhart

1. Einleitung

Darwin's Evolutionstheorie ist zweifellos eine der größten wissenschaftlichen Errungenschaften und bei Biologen und aufgeklärten Menschen auf breiter Basis anerkannt. Abgesehen von einigen religiösen Fanatikern bestreitet heute kein vernünftiger Mensch mehr, dass sich die lebendige Welt nach den Darwin'schen Prinzipien von Mutation und Selektion entwickelt hat.

So akzeptiert die Evolutionslehre in ihren Grundzügen ist, geht man ins Detail, sind einige zentrale Fragen offen. Eine dieser großen strittigen Fragestellungen ist, ob die natürliche Auslese der einzige Wirkfaktor ist, der die Entwicklung von Lebewesen beeinflusst. Zwar ist die natürliche Selektion als Hauptantriebskraft der Evolution grundsätzlich anerkannt. Das heißt aber nicht, dass sie die einzige ist. Es kann noch andere Faktoren geben, die die Evolution gravierend beeinflussen.

Schon Darwin bemerkte, dass „Organe von extremer Perfektion und Komplexität" eine Schwierigkeit für seine Theorie sein könnten. In der Tat erscheint es auf den ersten Blick unwahrscheinlich, dass Auge, Herz oder Leber allein durch natürliche Auslese geschaffen werden. Wer sehen kann, hat zwar offensichtlich einen enormen Selektionsvorteil gegenüber seinen blinden Konkurrenten. Aber wie soll ein Auge entstehen, wenn es zuvor noch keines gab?

Darwin selbst spekulierte, dass diese komplexen biologischen Merkmale durch kleine, schrittweise Übergänge über viele Zwischenstufen entstanden sein könnten. Allerdings muss dabei jede Zwischenstufe auf dem Weg zum Auge für sich genommen vorteilhaft gewesen sein, sonst wäre sie längst ausgestorben, bevor die nächste Mutation eintreten konnte. Sich solche vorteilhaften Zwischenschritte vorzustellen fällt schwer.

Der kürzlich verstorbene Paläontologe und Evolutionsforscher Stephen J. Gould und andere sind deshalb überzeugt, dass neben der natürlichen Selektion noch weitere Mechanismen für die Evolution verantwortlich sind. So behauptet Gould, dass sich komplexe Merkmale oder Arten nicht – wie Darwin vermutete - über viele Millionen Jahre durch akkumulierte genetische Veränderungen entwickeln, sondern kurze heftig Ausbrüche neue biologische Merkmale hervorbrachten. Ebenso spielen nach Auffassung von Gould schwerwiegende Katastrophen eine Rolle bei der Ausbildung langfristiger Evolutionsmuster wie das Aussterben der Dinosaurer vor 65 Millionen Jahren. Der Mensch, so Gould, hätte sich ohne diese Katastrophe gar nicht entwickelt.

Zwischen dem „pluralistische Lager" von Evolutionsforschern um Gould und den Anhängern der reinen Darwin-Lehre ist in den letzten Jahren ein Streit entbrannt, der fanatische Züge annahm. Gould beschimpfte seine Gegner als „Ultra-Darwinisten" und „Darwin'sche Fundamentalisten", die Darwins Ideen mit fast religiöser Ergebenheit vertreten. Die so beschimpften, vor allem Richard Dawkins und Daniel Dennett, bezeichneten Gould als Möchtegern-Revoluzzer, der Darwins Evolutionstheorie torpediert und seinen Bekanntheitsgrad missbraucht, um seine „verdrehten" Theorien zu verbreiten.

Amerikanische Forscher um Richard Lenski von der Michigan State University meinen nun, diesen Streitpunkt zu Gunsten Darwins und der „Fundamentalisten" beantworten zu können. Die Entstehung komplexer biologischer Phänomene lässt sich ausschließlich mittels

elementarster Vorgänge und natürlicher Selektion erklären. Zusätzliche Annahmen, wie sie Gould postuliert, sind nicht notwendig.

Die These der Wissenschaftler beruht nicht auf der klassischen organischen Evolutionsforschung. In der Evolutionsforschung lassen sich solche Entwicklungen eines Organismus nur schwer beobachten, denn es handelt sich hierbei um einen sehr langsamen Prozess, der mitunter Jahrmillionen dauert. Statt dessen gelangten sie zu dieser Erkenntnis über Experimente mit digitalen Organismen, die in einer virtuellen Petrischale eines Hochleistungscomputers leben. Die Evolutionsforscher nutzten hierfür die Linux-Software Avida um die Entwicklung digitaler Organismen näher zu untersuchen.

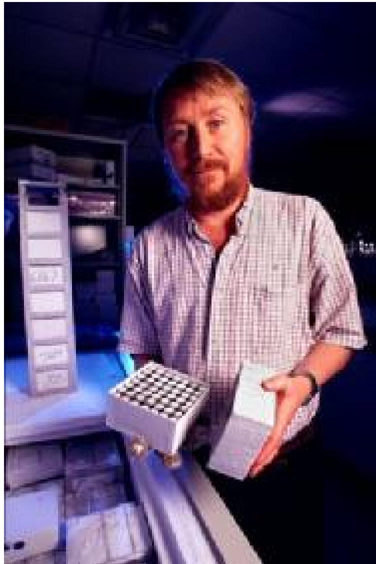

Abb. 1: Richard Lenski, Schöpfer der digitalen Organismen.

2. Digitale Organismen

Was sind digitale Organismen? Digitale Organismen kann man sich als kleine Computerprogramme vorstellen, die elementare Maschinenbefehle ausführen. In gewisser Weise ähneln sie Computerviren, die sich selbst kopieren und vermehren.

Ein digitaler Organismus in Avida ist eine virtuelle CPU, sie stellt den „Körper" oder die „Hardware" des Organismus dar (siehe Abb. 2). Die wesentlichen Teile eines Organismus sind sein **Genom** (linkes Band), zwei **Stacks** und drei **Register**. Die Ausführung des Genom-Programms erzeugt einen künstlichen „Stoffwechsel": Die numerischen Werte werden aus der Umgebung eingegeben, in den Stacks und Registern berechnet und als Ergebnisse an die Umgebung abgegeben werden.

Im Detail haben die einzelnen Komponenten eines Organismus folgende Funktion:

- Ein **Genom**, das aus einer Folge von Anweisungen besteht. Jede Anweisung ist mit einem Flag versehen um anzuzeigen, ob die Anweisung ausgeführt, kopiert oder mutiert wurde.
- Ein **Instruction Pointer** (lila Farbe), der anzeigt, welche Anweisung als nächstes ausgeführt wird.
- Drei **Register** AX, BX und CX (rosa), die vom Organismus verwendet werden können, um aktuell manipulierte Daten zu speichern. Sie werden meist von den Anweisungen verarbeitet und können 32-Bit-Strings speichern.

- Zwei **Stacks** (grün), die zum Speichern verwendet werden.
- Ein **Input**- und ein **Output-Buffer** (gelb), die der Organismus verwendet, um Information zu erhalten und die verarbeiteten Ergebnisse zurückzugeben.
- Ein **Read-Head, Write-Head und Flow-Head** (blau), die verwendet werden, um Positionen im Genom festzuhalten.

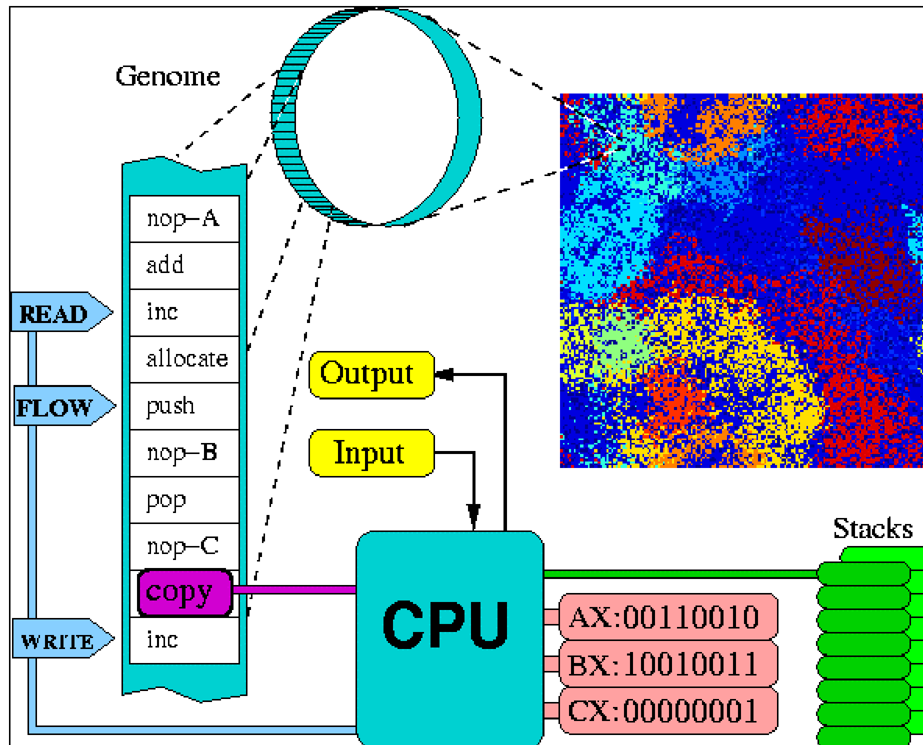

Abb. 2: Aufbau eines digitalen Organismus in Avida.

Wie in der Natur steht im Zentrum des digitalen Organismus das Genom, also die Menge der Erbinformationen. Während in der Natur die Erbinformationen in der DNA abgelegt sind, setzt sich das Genom der digitalen Organismen aus einem sich selbst replizierenden Computercode zusammen. Die Kette von Genen besteht aus einer Folge von Programminstruktionen, einfachen Maschinenbefehle, mit denen die Organismen simple Berechnungen durchführen kann. Normalerweise werden die Befehle hintereinander ausgeführt, Sprunganweisungen können die sequentielle Abarbeitung jedoch unterbrechen und den Pointer an eine andere Position springen lassen. Jeder Befehl in der Sequenz gehört zu einer von insgesamt 26 möglichen Anweisungen. Ist die Befehlskette abgearbeitet, beginnt das Spiel wieder von vorne, denn jeder Genomsatz ist ringförmig angelegt. Die technischen Details zu digitalen Organismen sind im Kasten „Digitale Organismen in Avida“ näher beschrieben.

In der virtuellen Petrischale beginnen die Organismen am Punkt Null. Zu Beginn sind alle Organismen identisch, jedes einzelne von ihnen konkurriert aber von Anfang an um bestimmte Ressourcen. Wie natürliche Organismen um Nahrung, Brutplätze oder Geschlechtspartner kämpfen, wetteifern die digitalen Gegenstücke im Computer um Rechenzeit. Je erfolgreicher ein Organismus, je mehr Rechenzeit er bekommt, umso komplexere Aufgaben kann er ausführen – und umso erfolgreicher kann er Nachkommen zeugen.

Jeder Organismus enthält in seinem Genom ein Copy-Kommando, mit dem er sich selbst kopieren kann. Dieser Befehl kopiert das Erbgut, sprich: den Programmcode, des Organismus Stück für Stück an eine andere Stelle in der Nachbarschaft des „Elternteils“. Die Kopie beginnt dann selbst wieder zu „leben“ und ihre Programm auszuführen.

Analog zu Naturvorgängen treten bei diesem Kopiervorgang mit einer geringen Wahrscheinlichkeit Fehler auf, Mutationen, die das Erbmaterial verändern. Dabei wird ein zu kopierendes Gen durch ein anderes ersetzt, weggelassen oder um ein weiteres verlängert - was sich entweder positiv oder negativ auf die Reproduktionsfähigkeit des Organismus auswirkt. Der Organismus wird dadurch entweder robuster oder so geschädigt, dass er ausselektiert wird. Welcher Genotyp sich erfolgreich durchsetzt und vermehrt hängt wie in der Natur von der Art der Mutation und der Umgebung ab. Die meisten Mutationen in Avida sind neutral oder schädlich, andere hingegen replizieren sich schneller oder können Operationen ausführen, die andere nicht im Programm haben. Dieser kleine Teil der Mutationen steigert die Fitness.

3. Kampf um Nahrung

Wie in der Natur verbraucht auch jeder künstliche Organismus Energie. Und wie in der Natur findet auch bei Avida ein Wettstreit um die „Nahrung“ zur Aufrechterhaltung der Lebensfunktionen statt. In der Simulation erfolgt die Energiezuteilung in Form von diskreten Einheiten, den „Single-Instruction-Processing“-Units, kurz SIPs. Mit jedem SIP kann genau eine Instruktion ausgeführt werden. Führt nun ein Organismus Befehle seines Genomprogramms aus kann er seine Fitness steigern, mehr Energie erhalten und sich besser und schneller kopieren.

Organismen können in Avida auf zweierlei Arten Energie erhalten. Erstens erhält jedes Individuum SIPs im Verhältnis zu seiner Genom-Länge. Je größer sein Genom, umso mehr Energie bekommt es. Zweitens und viel wichtiger: Zusätzliche SIPs erhält ein Organismus, indem er mehr oder weniger komplexe Logikverknüpfungen auf 32-Bit Strings ausführt. Wenn ein Organismus zwei vorgegebene Bitketten einliest und eine weitere Bitkette ausgibt, die genau dort eine Eins hat, wo beide eingelesenen Ketten eine Eins hatten, dann hat er die logische Verknüpfung UND richtig ausgeführt. Als Belohnung für diese Arbeit erhält er vier SIPs.

Solche Logikoperationen sind der zentrale Punkt der Simulation. Sie stellen in der Simulation die Problemlösefähigkeit eines Organismus dar und sind ein Maßstab für seine Komplexität. Ein Organismus, der keine oder nur primitive Operationen ausführen kann ist von seiner biologischen Struktur her einfacher aufgebaut als einer, der komplexe Operationen ausführen kann.

Prinzipiell ist jeder Organismus fähig, solche Logikoperationen auszubilden, weil sich unter den 26 Anweisungen auch die logische Verknüpfung NAND befindet. Die Funktion NAND („not and“) ist wie folgt definiert:

not (A and B).

A und B sind hierbei die Inputs der Funktion. Nach der üblichen Semantik für logische Operatoren liefert der NAND-Operator 0 („false“), wenn beide Inputs 1 („true“) sind; er liefert 1, wenn einer oder beide der Inputs 0 sind.

Mehr Logikoperatoren befinden sich nicht unter den Befehlen. Wie kann ein Organismus dann komplexere Logikfunktionen ausbilden? Dies ist deshalb möglich, weil jede denkbare logische Verknüpfung zweier Inputs sich aus Anwendungen von NAND zusammensetzt. Manche sind schwieriger als andere, weil sie mehr Anwendungen von NAND erfordern. Die schwierigste von ihnen heißt EQU (=EQUAL) und wird mit dem Höchstgewinn von 32 SIPs belohnt.

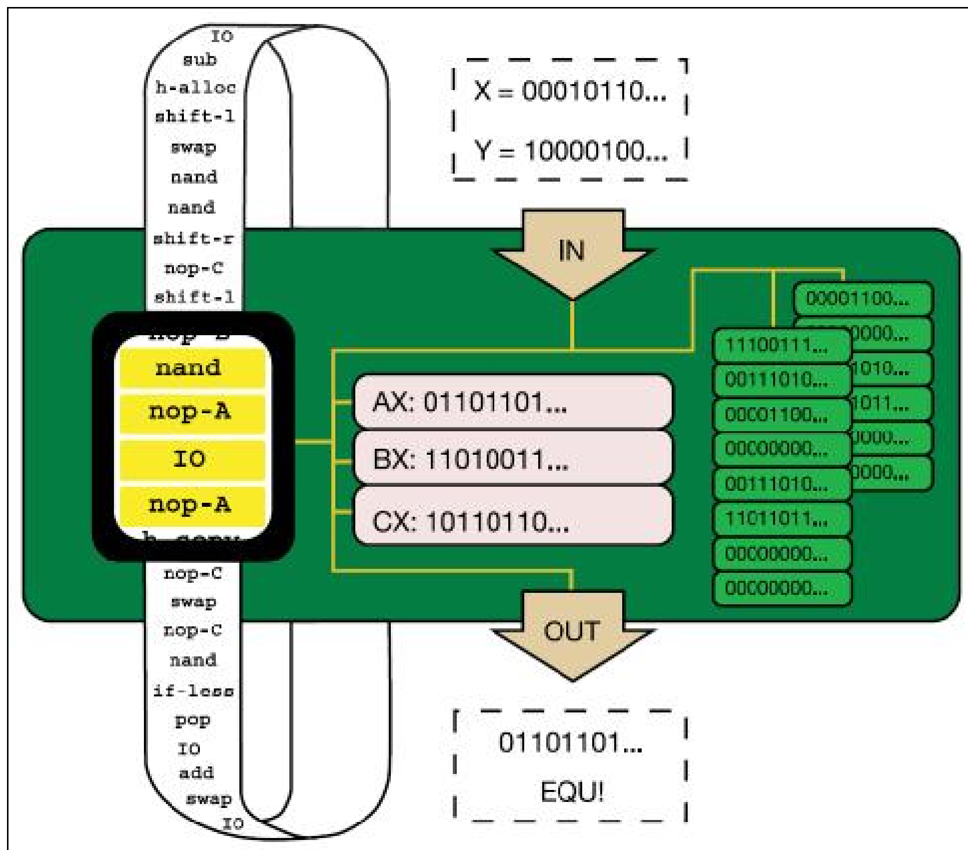

Abb. 3: Beispiel für die Ausführung eines Genom-Programms mit zwei Eingabe-Strings.

Das Beispiel in Abb. 3 veranschaulicht das eben Gesagte praktisch. Die Ausführung der hervorgehobenen NAND-Anweisung führt dazu, dass die Bit-Strings in den BX- und CX-Registern gemäß der Operatordefinition kombiniert werden. Tabelle 1 zeigt das Ergebnis der Operation bei den vorliegenden Inputs. In der 1. Zeile steht der Inhalt des BX-Registers, in der 2. der Inhalt von CX, die 3. Zeile zeigt die logische UND-Verknüpfung der Zellen. Die letzte Zeile ist das Ergebnis der logischen NAND-Kombination der BX-und CX-Register. Dieses Ergebnis wird in das AX-Register geschrieben (siehe Abb. 3).

Tabelle 1: **Das Ergebnis der NAND-Operation**

Logische Verknüpfung								
1: BX	1	1	0	1	0	0	1	1
2: CX	1	0	1	1	0	1	1	0
3: BX and CX	1	0	0	1	0	0	1	0
4: not (BX and CX) (=NAND)	0	1	1	0	1	1	0	1

Der Organismus erhält nun eine extra Menge Energie, da dieser Output im AX-Register perfekt mit dem Wert der vorgegebenen EQU-Funktion übereinstimmt. Im Beispiel sieht dies konkret wie folgt aus: Der Organismus in Abb. 3 erhält als Eingabe die Strings X (00010110..) und Y (10000100...) und liefert als Output den String im AX-Register (01101101...), wie er von der eben berechneten NAND-Instruktion erzeugt wurde. Der Organismus erhält nun die Belohnung von 32 SIPs (siehe Tabelle 3) für die Funktion EQU und zwar deshalb, weil der Output-String eine 1 an jeder Stelle hat, an der X und Y gleich sind (beide 0 oder beide 1) und eine 0, wenn X und Y verschieden sind.

Tabelle 2: **Das Ergebnis der EQU-Operation**

Logische Verknüpfung								
1:X	0	0	0	1	0	1	1	0
2:Y	1	0	0	0	0	1	0	0
3:X and Y	0	0	0	0	0	1	0	0
4:not X and not Y	0	1	1	0	1	0	0	1
5:(X and Y) or (not X and not Y) (=EQU)	0	1	1	0	1	1	0	1

Von allen Logikfunktionen (Tabelle 3) nimmt die EQU-Funktion in der Simulation eine zentrale Stelle ein. EQU bringt die höchste Belohnung und damit die meiste Energiezufuhr in dem Experiment, weil ihre Ausführung komplexer ist als jede andere der acht Logik-Funktionen. Für sie sind mindestens 5 NAND-Anwendungen nötig (siehe Avida-Beispielprogramm). Die Belohnungen errechnen sich aus 2 hoch n, wobei n die Minimalzahl von NAND-Operationen ist, die zur Ausführung der Funktion nötig ist. Tabelle 3 gibt einen Überblick über alle verwendeten Funktionen und deren Belohnung.

Tabelle 3: **Verwendete Logik-Funktionen und ihre Belohnungen**

Funktionsname	*Logik-Operation*	*Belohnung*
NOT	not A; not B	2
NAND	not (A and B)	2
AND	A and B	4
OR_N	(A or not B); (not A or B)	4
OR	A or B	8
AND_N	(A and not B); (not A and B)	8
NOR	not A and not B	16
XOR	(A and not B) or (not A and B)	16
EQU	(A and B) or (not A and not B)	32

Zusammengefasst passiert bei Ablauf der Simulation also folgendes. Die Urahnen, die ersten Organismen, können sich replizieren, aber können noch keine Logikfunktionen ausführen. Ein einzelner Kopierfehler (Mutation) kann noch nicht einmal die einfachste Funktion durchführen. Statt dessen müssen mehrere Mutationen in gerader Linie erfolgen – und zwar so, dass sie koordiniert ausgeführt werden – um selbst die einfachsten Funktionen exekutieren zu können. Ein Organismus, der eine oder mehrere der Logikfunktionen ausführen kann erhält zusätzliche Energie. Der Nutzen wächst exponentiell mit der Schwierigkeit jeder Funktion. Je komplexer die Funktion, umso mehr Energie erhält der Organismus und umso schneller kann er sich replizieren.

Soweit die Ausgangsbasis. Die alles entscheidende Frage, die sich nun stellt, ist: Schaffen es die digitalen Organismen, bis zur komplexesten aller Logikfunktionen, der EQU-Funktion, vorzudringen. Übersetzt in die Sprache der Evolutionstheorie heißt das: Können einfache Organismen komplexe Problemlösungsmechanismen entwickeln, die allein über Mutation und Selektion entstehen?

4. Darwins Sieg

Lenski und seine Mitarbeiter untersuchten 50 unterschiedliche Populationen mit je 3600 digitalen Organismen. Jede der 3.600 Organismen war beim Start eine identische Kopie eines Genom-Ahnen, bestehend aus 50 Zeilen Genom-Code. 15 der Code-Zeilen waren dabei nötig für die Selbstreplikation, Code für die Ausführung von Logikoperationen war nicht enthalten.

Nach knapp 16.000 Generationen zeigten 23 der 50 Populationen - also fast jede zweite - die Fähigkeit, die komplexen EQU-Operationen ausführen zu können und zwei Inputs auf ihre Äquivalenz zu prüfen. Der Genom-Code dieser Individuen – also die Anzahl der Programmbefehle - variierte zwischen 49 und 356 Anweisungen. Es gab also sehr viele verschiedene - kurze und längere - Wege zu dieser Fähigkeit. Der letztendlich dominante Typ der Organismen enthielt 83 Instruktionen und die Fähigkeit, alle neun Logikfunktionen ausführen zu können. War die EQU-Fähigkeit einmal vorhanden, ging sie nicht wieder verloren - auch wenn sich die Organismen weiter entwickelten und zusätzliche Fähigkeiten ausbildeten.

Im Prinzip konnten schon 16 Mutationen und drei Anweisungen, die im Originalcode der ersten Organismen präsent waren, so kombiniert werden um Individuen zu produzieren, die die komplexe EQU-Operation ausführen konnten. Die aktuellen Pfade waren jedoch viel länger und sehr variabel und zeigen, wie weitschweifig und unverhersagbar die Evolution ist.

Die EQU-Operation erschien erstmals irgendwo zwischen Schritt 51 und 721 im evolutionären Baum und die Organismen benutzten zwischen 17 und 43 Codezeilen für die Ausführung.. Die effizienteste der sich entwickelnden EQU-Funktionen basierte auf gerade mal 17 Codezeilen – zwei weniger, als der hocheffiziente Code, den die Forscher von Hand ermittelten (siehe Anhang „Avida Beispielprogramm“).

Um herauszufinden welche der Programm-Anweisungen notwendig sind für die Ausbildung der EQU-Funktion wurde eine weitere Simulationsvariante durchgeführt. Dabei zeichnete ein Tool auf, welche Codezeilen eines Organismus gebraucht werden, um eine bestimmte Funktion auszuführen. Dieses Werkzeug offenbarte, wie sich die Leistungsfähigkeit des ganzen Organismus durch Entfernen jeder Anweisung änderte.

Die Simulation bewies, dass die komplexeste EQU-Funktion sich nur entwickelte, wenn zuvor einfachere Funktionen hergeleitet wurden. Einige simple Funktionen standen schon wenige Mutationen nach Simulationsstart zur Verfügung, diese dienten als Basis, auf die komplexere Funktionen aufbauten. In einem Fall ging nach Abänderung jeder einzelnen von 60 Codezeilen in 35 Fällen die EQU-Fähigkeit verloren.

Letztlich werden also über viele Generationen erworbene, einfachere Fähigkeiten gebraucht, um komplexere Operationen durchführen zu können. Diese Ergebnisse sprechen dafür, dass komplexe Merkmale auf einfacheren Merkmalen aufbauen und durch Veränderung bestehender Strukturen und Funktionen entstehen. Die künstlichen Organismen stützen also ganz klar Darwins Vermutung, dass sich komplexe Merkmale infolge einer schrittweisen Anhäufung kleinerer Veränderungen bilden.

Instruction	RepL	NOT	NAND	AND	OR N	OR	AND_N	NOR	XOR	EQU
1 r *h-alloc*										
2 m *dec*										
3 z *set-flow*										
4 a *nop-A*										
5 v *mov-head*										
6 c *nop-C*										
7 g *push*										
8 m *dec*										
9 c *nop-C*										
10 i *swap*										
11 q *IO*										
12 q *IO*										
13 p *nand*										
14 t *h-copy*										
15 q *IO*										
16 p *nand*										
17 q *IO*										
18 c *nop-C*										
19 p *nand*										
20 c *nop-C*										
21 t *h-copy*										
22 l *inc*										
23 e *if-less*										

Abb. 4: Welche Befehle sind notwendig für die Ausführung von EQU? In der linken Spalte findet man einen Auszug der Genom-Sequenz für einen Beispielorganismus, oben in der Zeile die Logikfunktionen. Grün bedeutet, der Organismus kann die Funktion ausführen, rot, er kann sie nicht ausführen. Die Farben in den Zellen geben Aufschluss darüber was passiert, wenn der jeweilige Befehl eliminiert wird. Rot bedeutet, dass die Funktion zerstört wird, grün produziert die Funktion und weiß hat keinen Effekt. Würde die 2. Zeile „dec" also weggelassen, könnte der Organismus die Funktionen OR, AND-N, NOR und EQU nicht ausführen.

Was aber ist mit den Vorteilen bei den Zwischenstufen, die bei den Evolutionsschritten zu komplexen Organen wie dem Auge ja ihrerseits immer gegeben sein müssten? Bilden sich solche Fähigkeiten nicht aus, wenn die Zwischenstufen keinen Vorteil bringen? Die Forscher ließen die Simulation abermals ablaufen, gaben aber diesmal für eine oder zwei der einfachen Logikfunktionen keine Belohnung. Das Ergebnis: Egal welche das waren, die Fähigkeit, die komplexe EQU-Funktion ausführen zu können entwickelte sich kaum seltener. Erst als alle kleinen Belohnungen gestrichen wurden, brachte es kein Organismus mehr zu Höchstleistungen. Vielmehr wurden sie im Laufe der Zeit kürzer: Es gab eben nichts, wofür es sich gelohnt hätte, ein längeres Genom mit dem Nachteil verzögerter Fortpflanzung zu kultivieren. Auf einem kurzen Genom aber findet beim besten Willen keine EQU-Fähigkeit Platz. Nur wenn die Umwelt die Entwicklung langer Genome begünstigt, entwickeln sich komplexe Eigenschaften.

Der Blick auf die Evolution aus der Vogelperspektive räumt auch mit einer alten Vorstellung auf: Biologen nahmen bislang an, dass die Evolution von neuen Mutationen ein stetig bergauf führender Anstieg ist. Vergleichbar einem Bergsteiger, der immer aufwärts steigt oder sich allenfalls seitlich bewegt, so die Vorstellung, hängen die Gewinner von den fittesten Organismen früherer Generationen ab. Die Simulation zeigt, dass dies nicht unbedingt notwendig ist.

Die Forscher fanden, dass unter den Vorfahren der späteren Gewinner einige Mutationen hatten, die kurzfristig schädlich waren, also die bislang aufgebauten Fähigkeiten wieder zerstörten. Zwei Schritt zurück zu gehen wäre manchmal besser gewesen. Einige dieser schädlichen Mutationen arbeiteten aber gut mit anderen Mutationen zusammen, die später auftraten. Während die meisten schädlichen Mutationen ausgemerzt wurden, wurden einige weitergegeben und wandelten ein kurzfristiges Handicap in einen langfristigen Vorteil um. Ein kurzfristiger Rückschritt kann also langfristig durchaus von Vorteil sein.

In einem typischen Simulationslauf tauchte die EQU-Funktion beispielsweise im Schritt 111 auf. Dabei stellten sich 45 Mutationen langfristig als gut, also Fitness-steigernd, 48 als neutral und 18 als schlecht heraus. 15 der 18 schlechten Mutationen verringerten die Fitness des Nachwuchs im Vergleich zu den Eltern leicht. Zwei der schlechten Mutationen halbierten sogar die Fitness um 50 Prozent. Eine dieser sehr schlechten Mutationen produzierte jedoch einen Nachkommen, der einen Schritt später eine Mutation erzeugte, die die EQU-Funktion ausführen konnte.

5. Fazit

Überträgt man die Evolution der EQU-Funktion auf das Eingangsbeispiel vom Auge bedeutet dies: EQU ist ebenso wie das Sehen eine komplexe Fähigkeit. Sie erfordert eine große Zahl an Mutationen, deren keine für sich allein vorteilhaft ist. So wie sehr viele Computer-Populationen unabhängig voneinander die komplexe EQU-Funktion „erfunden“ haben, können die verschiedensten Tierpopulationen unabhängig voneinander das komplexe Auge „erfunden“ haben. Sehen können ist so ungeheuer vorteilhaft, dass die blinde Evolution früher oder später einfach drauf kommen musste. So wie es viele evolutionäre Wege zur EQU-Ausführung gibt, gibt es viele entwicklungsbiologische Wege zum Sehen. Jeder von ihnen ist vielleicht eine höchst unwahrscheinliche Anhäufung von Zufällen. Aber dass die Natur von den vielen Wegen einen findet, ist nicht mehr ganz so unwahrscheinlich.

Abb. 5: Im Digital Live Lab des California Institus of Technology erfährt man alles über Avida.

Wer selbst einmal Evolution spielen will, kann dies gerne tun. Eine Möglichkeit ist zum Beispiel, geschlechtliche Vermehrung einzuführen, die in Lenski's Modell noch nicht berücksichtigt ist. Im Internet findet sich eine Kopie des Programms, mit dem solche und andere Experimente versucht werden können. Das Programm und die Konfigurations-Dateien für das Experiment können unter http://myxo.css.msu.edu/papers/nature2003 kostenlos bezogen werden. Dort finden sich auch weitere Informationen über Avida sowie zusätzliche Daten aus dem Experiment.

Literatur

Lenski, Richard u.a.: The Evolutionary Origin of Complex Features, in: Nature 423, Mai 2003, S.139-143, http://myxo.css.msu.edu/papers/nature2003/Nature03_Complex.pdf

Dawkins, Richard: Das egoistische Gen. Berlin, Springer-Verlag, Heidelberg 1978

Dawkins, Richard: Der blinde Uhrmacher. München, Rowohlt, 1987

Dennett, Daniel: Darwins gefährliches Erbe. Hamburg, Hofmann & Campe, 1997

Morris, Richard: Darwins Erbe. Hamburg, Europa-Verlag, 2001

Klaus Manhart
www.klaus-manhart.de
mail@klaus-manhart.de

ANHANG: Technischer Teil

Þ Avida – Das Grundkonzept

Avida (http://dllab.caltech.edu/avida/), geschrieben von Christoph Adami vom California Institute of Technology in Pasadena, ist ein selbstadaptierendes genetisches System, das vorrangig für den Einsatz auf dem Gebiet des Künstlichen Lebens konzipiert wurde. Es basiert auf ähnlichen Konzepten wie das bekanntere Tierra-Programm (http://www.isd.atr.co.jp/~ray/tierra/). Bei Simulationssystemen dieses Typs kämpfen assemblerähnliche Programme um Speicherplatz oder Rechenzeit in ihrem Host-Computer. Die oft nur aus wenigen Anweisungen bestehenden Programme können sich an freie Stellen im Hauptspeicher kopieren, neue Anweisungen hinzufügen oder alte löschen.

Der Hauptzweck dieser Simulationssysteme ist es, die evolutionäre Anpassung zu beobachten oder allgemeine Eigenschaften von lebenden Systemen (wie z. B. Selbstorganisation) sowie andere Ergebnisse, die zur theoretischen oder evolutionären Biologie und zu dynamischen Systemen gehören.

Avida erinnert stark an Assembler, unterscheidet sich davon aber durch die zusätzliche Verwendung von nop-Anweisungen („No Operation“-Kommandos). Diese haben bei der Ausführung keine direkte Auswirkung auf die virtuelle CPU, aber verändern oft die Auswirkung von Befehlen, die ihnen vorangehen. Drei solcher nop-Befehle sind in Avida enthalten: nop-A, nop-B und nop-C.

Die anderen Avida-Befehle können in drei Gruppen geteilt werden. Zur ersten Gruppe gehören jene Befehle, die von nops nicht beeinflusst werden. Die meisten von ihnen sind „biologischen“ Anweisungen wie copy, die direkt in den Replikationsprozess eingreifen.

Zur zweiten Gruppe gehören jene Anweisungen, bei denen nop den Head oder Register (siehe Abb. 1) ändert, abhängig von den vorherigen Befehlen. Ein inc-Kommando beispielsweise, dem der Befehl nop-A folgen würde, würde den Inhalt des AX-Registers um 1 erhöhen. Ein inc-Kommando gefolgt von nop-B würde hingegen BX inkrementieren.

In Avida werden Befehlsdefinitionen, bei denen Komponenten wie Register und Head gemäß eines nop-Kommandos ersetzt werden können, mit beidseitigen Fragezeichen markiert, etwa: ?BX?. Wenn die Komponente zwischen den Fragezeichen ein Register ist, dann repräsentiert ein folgendes nop-A das AX-Register, nop-B ist BX und nop-C ist CX. Wenn die Komponenten ein Head einschließlich des Instruction Pointers ist, dann repräsentiert nop-A den Instruction Pointer, nop-B den Read-Head und nop-C den Write-Head.

Zur dritten Gruppe von Avida-Befehlen gehören schließlich jene, die eine ganze Serie von nop-Anweisungen nutzen. Da diese hier nicht wichtig sind, wollen wir sie übergehen.

Die folgende Tabelle zeigt die Definition einiger Avida-Befehle, die zum Verständnis des kürzesten, handgeschriebenen Programms (siehe Extra-Kasten) nötig sind.

Auszug aus Avida-Befehlen

Befehl	*Definition*
nop-A, nop-B, and nop-C	No-operation-Anweisungen; diese modifizieren andere Anweisungen
pop	Entferne eine Zahl aus dem aktuellen Stack und lege sie in ?BX? ab
push	Kopiere den Wert von ?BX? auf den gegenwärtigen Stack
swap	Tausche den Inhalt von ?BX? mit seinem Komplement
inc	Inkrementiere ?BX?
dec	Dekrementiere ?BX?
nand	Führe bitweise NAND-Operation auf BX und CX aus; lege das Ergebnis in ?BX? ab
IO	Gib den Wert von ?BX? aus und ersetze ihn mit einem neuen Input

Þ Avida Beispielprogramm

Das folgende handgeschriebene Programm scheint das kürzeste zu sein, das zur Ausführung der EQU-Funktion führt. Es hängt nicht vom Anfangsinhalt der Stacks und Register ab, deshalb sind die Anfangsinhalte durch Fragezeichen dargestellt. Ob dies tatsächlich das kürzeste Programm ist, ist allerdings nicht bewiesen worden, es handelt sich nur um eine Vermutung der Forscher. Auch enthält dieses Programm keinen Code für Selbstreplikation, sondern führt ausschließlich Logikoperationen aus.

Kürzestes, handgeschriebenes EQU-Programm

Nr.	*Befehl*	*AX*	*BX*	*CX*	*Stack*	*Output*
1	**IO**	?	X	?	?	?
2	**IO**	?	X	Y	?	?
3	**nop-C**					
4	**push**	?	X	Y	X, ?	
5	**nand**	?	X nand Y	Y	X, ?	
6	**swap**	?	Y	X nand Y	X, ?	
7	**nand**	?	X or not Y	X nand Y	X, ?	
8	**swap**	X or not Y	?	X nand Y	X, ?	
9	**nop-A**					
10	**pop**	X or not Y	X	X nand Y	?	
11	**nand**	X or not Y	Y or not X	X nand Y	?	
12	**swap**	X nand Y	Y or not X	X or not Y	?	
13	**nop-C**					
14	**nand**	X nand Y	X xor Y	X or not Y	?	
15	**push**	X nand Y	X xor Y	X or not Y	X xor Y, ?	
16	**pop**	X nand Y	X xor Y	X xor Y	?	
17	**nop-C**					
18	**nand**	X nand Y	X equ Y	X xor Y	?	
19	**IO**	X nand Y	Z	X xor Y	?	X equ Y