

noch zu 11.: Evolution

Klassifikation von Systemen mit künstlicher Evolution nach dem Grad der Offenheit der Evolution:

klassische evolutionäre Algorithmen (insbes. GA)	<ul style="list-style-type: none">• Fitnessfunktion vorgegeben• Gene mit fester Auswahl von Allelen• (meist) feste Genom-Länge
LindEvol	<ul style="list-style-type: none">• Fitnessfunktion (z.T.) intrinsisch• aber: Selbstreplikation wird durch das Simulationssystem erledigt
Tierra, Avida, Amoeba	<ul style="list-style-type: none">• Fitnessfunktion und Selbstreplikation intrinsisch

Tierra

(span.: "Erde")

Thomas S. Ray 1989/90

Biologe, Spezialgebiet Ökologie

1989 Assistant Prof. an der Universität Delaware

seine Def. von "Leben":

"I would consider a system to be living if it is self-replicating and capable of open-ended evolution." (Ray 1992, S. 372)

Idee: digitale Form von *offener* Darwin'scher Evolution anhand von Computerprogrammen (Motivation: Computerviren)

Ray erinnert sich, wie er diese Idee in einem Ökologieseminar vorstellte: "Ich wurde buchstäblich aus dem Raum hinausgelacht", sagt er. Auch Rays Kollegen, die kurz zuvor seine Anstellung befürwortet hatten, erklärten die Idee für verrückt.

Kontaktaufnahme mit Langton und Farmer (Los Alamos):

Idee, einen virtuellen Computer zu verwenden

biologisches Vorbild für Tierra:

nicht die Entstehung des Lebens auf der Erde, sondern die "kambrische Explosion": Entwicklung der ersten Mehrzeller im Kambrium, zugleich starkes Anwachsen der morphologischen Diversität

Grundlagen von Tierra:

- Genotyp = Phänotyp
- Befehle ähnlich Intel i860 Maschinensprache
- ähnlich Redcode (Core Wars)
- Ressourcen = CPU-Zeit und Speicherplatz
- aber: Redcode-Programme erwiesen sich als zu anfällig bei Mutationen ("brittleness" = Brüchigkeit der Selbstreplikationsfähigkeit)

Ursache: Redcode hat nur ca. 10 Befehle, aber die meisten Befehlen haben 1 oder 2 Operanden \Rightarrow wahre Größe des Befehlssatzes liegt bei 10^{11} !

- deshalb: Befehlssatz der (wahren) Größe 32 (5 Bits), *keine Operanden*
- template- (Schablonen-) basierte Adressierung (Idee aus der Biologie)
- zusätzlich Register und Stack für jeden "Organismus"

templates: Folgen von aufeinanderfolgenden Befehlen der beiden Typen `nop_0` und `nop_1` ("no operation" – reine Markerbefehle, meist 4 hintereinander)

Suchbefehl (`jmp`-Befehl, der durch Folge von `nop`-Befehlen gefolgt wird) sucht nach nächstem komplementären template (0 und 1 vertauscht) und lenkt den Befehlszeiger auf das Ende des gefundenen templates

Befehlssatz von Tierra:

```

void execute(int di, int ci)
{
    switch(di)
    {
        case 0x00: nop_0(ci);      break; /* no operation */
        case 0x01: nop_1(ci);      break; /* no operation */
        case 0x02: or1(ci);         break; /* flip low order bit of cx, cx ^= 1 */
        case 0x03: shl(ci);         break; /* shift left cx register, cx <<= 1 */
        case 0x04: zero(ci);        break; /* set cx register to zero, cx = 0 */
        case 0x05: if_cz(ci);       break; /* if cx==0 execute next instruction */
        case 0x06: sub_ab(ci);      break; /* subtract bx from ax, cx = ax - bx */
        case 0x07: sub_ac(ci);      break; /* subtract cx from ax, ax = ax - cx */
        case 0x08: inc_a(ci);       break; /* increment ax, ax = ax + 1 */
        case 0x09: inc_b(ci);       break; /* increment bx, bx = bx + 1 */
        case 0x0a: dec_c(ci);       break; /* decrement cx, cx = cx - 1 */
        case 0x0b: inc_c(ci);       break; /* increment cx, cx = cx + 1 */
        case 0x0c: push_ax(ci);     break; /* push ax on stack */
        case 0x0d: push_bx(ci);     break; /* push bx on stack */
        case 0x0e: push_cx(ci);     break; /* push cx on stack */
        case 0x0f: push_dx(ci);     break; /* push dx on stack */
        case 0x10: pop_ax(ci);      break; /* pop top of stack into ax */
        case 0x11: pop_bx(ci);      break; /* pop top of stack into bx */
        case 0x12: pop_cx(ci);      break; /* pop top of stack into cx */
        case 0x13: pop_dx(ci);      break; /* pop top of stack into dx */
        case 0x14: jmp(ci);         break; /* move ip to template */
        case 0x15: jmpb(ci);        break; /* move ip backward to template */
        case 0x16: call(ci);        break; /* call a procedure */
        case 0x17: ret(ci);         break; /* return from a procedure */
        case 0x18: mov_cd(ci);      break; /* move cx to dx, dx = cx */
        case 0x19: mov_ab(ci);      break; /* move ax to bx, bx = ax */
        case 0x1a: mov_iab(ci);     break; /* move instruction at address in bx
                                     to address in ax */
        case 0x1b: adr(ci);         break; /* address of nearest template to ax */
        case 0x1c: adrb(ci);        break; /* search backward for template */
        case 0x1d: adrf(ci);        break; /* search forward for template */
        case 0x1e: mal(ci);         break; /* allocate memory for daughter cell */
        case 0x1f: divide(ci);     break; /* cell division */
    }
    inst_exec_c++;
}

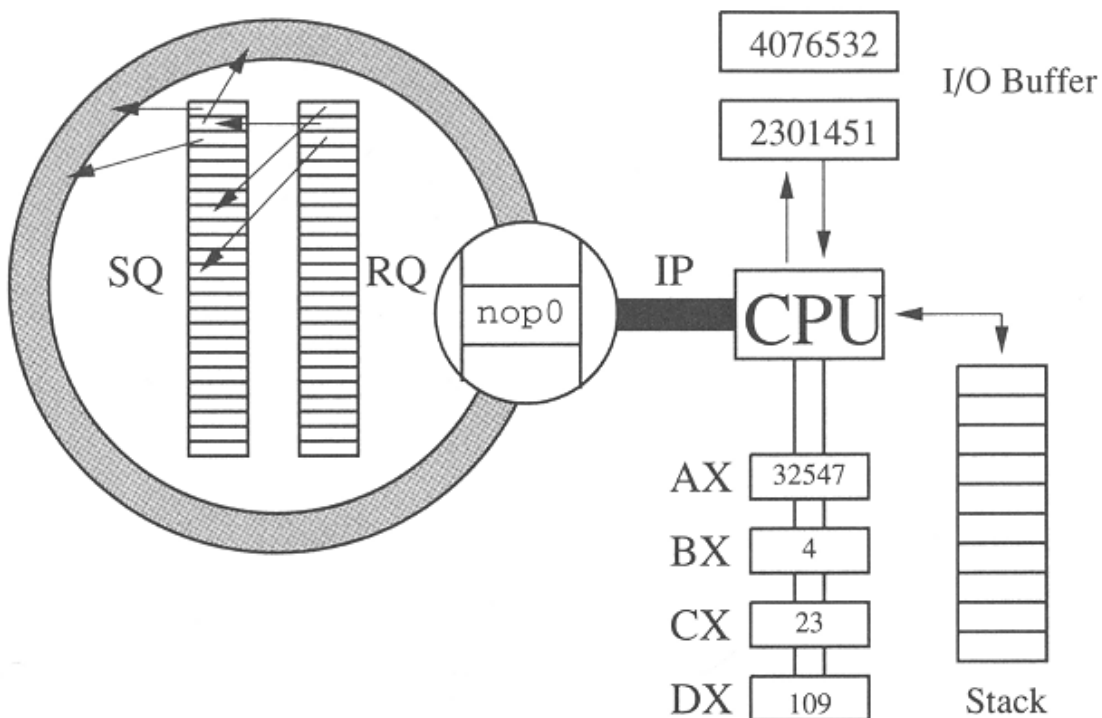
```

Beachte: Zahlenkonstanten müssen durch Bit-Flipping und Shifts (`or1`, `shl`) konstruiert werden.

virtuelle CPU:

multiple instruction multiple data (MIMD) System, emuliert durch "slicer queue", die zyklisch jedem Individuum Rechenzeit zuordnet (vgl. Core Wars)

```
struct cpu { /* structure for registers of virtual cpu */
    int ax; /* address register */
    int bx; /* address register */
    int cx; /* numerical register */
    int dx; /* numerical register */
    char fl; /* flag */
    char sp; /* stack pointer */
    int st[10]; /* stack */
    int ip; /* instruction pointer */
};
```



(IP = Befehlszeiger, SQ = slicer queue, RQ = reaper queue)

Registerzellen und Stack sind die einzigen Operanden der Befehle

"Genbank-Programm" protokolliert Eigenschaften jedes Individuums

- ringförmiger RAM (vgl. Core Wars) mit (1. Version!) 60 000 Bytes (entspr. 60 000 Befehle) – Bezeichnung durch Ray: "The soup"
- im Gegensatz zu Core Wars sind die von einem Individuum belegten Speicherzellen *schreibgeschützt* (aber nicht lese- und execute-geschützt!)
- ein Individuum (= Programm) kann in 2. Speicherblock ("Tochter-Organismus") schreiben, wenn der **ma1**-Befehl (memory allocation) ausgeführt wurde
- nach **divide** wird dieser 2. Programmteil ein selbstständiges Individuum und wird in beide queues eingereiht
- um "Überbevölkerung" und deadlock des Systems zu vermeiden: *reaper queue* ("Schnitter-Schlange") – Individuen, die am Ende dieser Warteschlange stehen, werden "getötet", wenn der Speicher zu 80 % belegt ist (der "tote" Code wird nicht aus dem Speicher entfernt!), frisch "geborene" Individuen kommen ganz an den Anfang
- wenn ein Individuum bei Befehlsausführung eine Fehlerbedingung generiert, bewegt es sich um 1 Platz in der reaper queue weiter ("Bestrafung" schlechter Algorithmen)

3 Arten von Mutationen:

- cosmic ray mutations: "Hintergrundrauschen", das gelegentlich 1 zufällig gewähltes Bit aus dem RAM ändert (Rate: 1 Bit geflippt nach durchschnittl. 10 000 Befehlsausführungen)
- copy mutations: ungenaue Ausführung von copy-Befehlen (1 Bit fehlerhaft pro 1000-2500 kopierten Befehlen)
- Ausführungsmutationen: ungenaue Ausführung von anderen Befehlen, z.B. Inkrementierung eines Registerinhalts um 0 oder um 2 statt um 1 (niedrige Rate).

Selbstreplikation erfolgt nicht automatisch, sondern das Individuum muss in seiner Befehlsfolge selbst dafür Vorkehrung treffen

Start des ersten Tierra-Simulationslaufs:

1 von Hand programmierter "Urahne" mit 80 Befehlen, der zur Selbstreplikation fähig ist

Prinzip:

- Programm bestimmt Adressen seines Anfangs und seines Endes (über templates) und deren Differenz
- alloziert Speicherblock dieser Größe für Tochter-Organismus
- kopiert gesamtes Genom in Tochter-Speicherbereich (copy-Prozedur)
- führt divide-Befehl aus und macht weiter mit Allozieren von neuem Tochter-Speicherbereich (Endlosschleife)

Genom des "Urahnens":

001	nopl	021	nopl	041	nopl	061	inc_b
002	nopl	022	inc_a	042	nopl	062	jmp
003	nopl	023	sub_ab	043	nop0	063	nop0
004	nopl	024	nopl	044	nop0	064	nopl
005	zero	025	nopl	045	push_ax	065	nop0
006	orl	026	nop0	046	push_bx	066	nopl
007	shl	027	nopl	047	push_cx	067	if_cz
008	shl	028	mal	048	nopl	068	nopl
009	mov_cd	029	call	049	nop0	069	nop0
010	adrb	030	nop0	050	nopl	070	nopl
011	nop0	031	nop0	051	nop0	071	nopl
012	nop0	032	nopl	052	mov_iab	072	pop_cx
013	nop0	033	nopl	053	dec_c	073	pop_bx
014	nop0	034	divide	054	if_cz	074	pop_ax
015	sub_ac	035	jmp	055	jmp	075	ret
016	mov_ab	036	nop0	056	nop0	076	nopl
017	adrf	037	nop0	057	nopl	077	nopl
018	nop0	038	nopl	058	nop0	078	nopl
019	nop0	039	nop0	059	nop0	079	nop0
020	nop0	040	if_cz	060	inc_a	080	if_cz

(kommentierte Fassung auf den nächsten 2 Seiten)

genotype: 80 aaa origin: 1-1-1990 00:00:00:00 ancestor
 parent genotype: human
 1st_daughter: flags: 0 inst: 839 mov_daught: 80
 2nd_daughter: flags: 0 inst: 813 mov_daught: 80

```

nop_1 ; 01 0 beginning template
nop_1 ; 01 1 beginning template
nop_1 ; 01 2 beginning template
nop_1 ; 01 3 beginning template
zero ; 04 4 put zero in cx
or1 ; 02 5 put 1 in first bit of cx
shl ; 03 6 shift left cx
shl ; 03 7 shift left cx, now cx = 4
; ax = bx =
; cx = template size dx =
mov_cd ; 18 8 move template size to dx
; ax = bx =
; cx = template size dx = template size
adrb ; 1c 9 get (backward) address of beginning template
nop_0 ; 00 10 compliment to beginning template
nop_0 ; 00 11 compliment to beginning template
nop_0 ; 00 12 compliment to beginning template
nop_0 ; 00 13 compliment to beginning template
; ax = start of mother + 4 bx =
; cx = template size dx = template size
sub_ac ; 07 14 subtract cx from ax
; ax = start of mother bx =
; cx = template size dx = template size
mov_ab ; 19 15 move start address to bx
; ax = start of mother bx = start of mother
; cx = template size dx = template size
adrf ; 1d 16 get (forward) address of end template
nop_0 ; 00 17 compliment to end template
nop_0 ; 00 18 compliment to end template
nop_0 ; 00 19 compliment to end template
nop_1 ; 01 20 compliment to end template
; ax = end of mother bx = start of mother
; cx = template size dx = template size
inc_a ; 08 21 to include dummy statement to separate creatures
sub_ab ; 06 22 subtract start address from end address to get size
; ax = end of mother bx = start of mother
; cx = size of mother dx = template size
nop_1 ; 01 23 reproduction loop template
nop_1 ; 01 24 reproduction loop template
nop_0 ; 00 25 reproduction loop template
nop_1 ; 01 26 reproduction loop template
mal ; 1e 27 allocate memory for daughter cell, address to ax
; ax = start of daughter bx = start of mother
; cx = size of mother dx = template size

```



```

call      ; 16 28 call template below (copy procedure)
nop_0    ; 00 29 copy procedure compliment
nop_0    ; 00 30 copy procedure compliment
nop_1    ; 01 31 copy procedure compliment
nop_1    ; 01 32 copy procedure compliment
divide   ; 1f 33 create independent daughter cell
jmp      ; 14 34 jump to template below (reproduction loop, above)
nop_0    ; 00 35 reproduction loop compliment
nop_0    ; 00 36 reproduction loop compliment
nop_1    ; 01 37 reproduction loop compliment
nop_0    ; 00 38 reproduction loop compliment
if_cz    ; 05 39 this is a dummy instruction to separate templates
          ;      begin copy procedure
nop_1    ; 01 40 copy procedure template
nop_1    ; 01 41 copy procedure template
nop_0    ; 00 42 copy procedure template
nop_0    ; 00 43 copy procedure template
push_ax  ; 0c 44 push ax onto stack
push_bx  ; 0d 45 push bx onto stack
push_cx  ; 0e 46 push cx onto stack
nop_1    ; 01 47 copy loop template
nop_0    ; 00 48 copy loop template
nop_1    ; 01 49 copy loop template
nop_0    ; 00 50 copy loop template
mov_iab  ; 1a 51 move contents of [bx] to [ax]
dec_c    ; 0a 52 decrement cx
if_cz    ; 05 53 if cx == 0 perform next instruction, otherwise skip it
jmp      ; 14 54 jump to template below (copy procedure exit)
nop_0    ; 00 55 copy procedure exit compliment
nop_1    ; 01 56 copy procedure exit compliment
nop_0    ; 00 57 copy procedure exit compliment
nop_0    ; 00 58 copy procedure exit compliment
inc_a    ; 08 59 increment ax
inc_b    ; 09 60 increment bx
jmp      ; 14 61 jump to template below (copy loop)
nop_0    ; 00 62 copy loop compliment
nop_1    ; 01 63 copy loop compliment
nop_0    ; 00 64 copy loop compliment
nop_1    ; 01 65 copy loop compliment
if_cz    ; 05 66 this is a dummy instruction, to separate templates
nop_1    ; 01 67 copy procedure exit template
nop_0    ; 00 68 copy procedure exit template
nop_1    ; 01 69 copy procedure exit template
nop_1    ; 01 70 copy procedure exit template
pop_cx   ; 12 71 pop cx off stack
pop_bx   ; 11 72 pop bx off stack
pop_ax   ; 10 73 pop ax off stack
ret      ; 17 74 return from copy procedure
nop_1    ; 01 75 end template
nop_1    ; 01 76 end template
nop_1    ; 01 77 end template
nop_0    ; 00 78 end template
if_cz    ; 05 79 dummy statement to separate creatures

```

von Ray im Original-Versuch verwendete Hardware (1990):

Toshiba 5200/100 Laptop mit 80386-Prozessor und 80387-math. Coprozessor (20 MHz, entspr. 12 Millionen Tierra-Befehle pro Stunde)

839 Befehle für erste Replikation des Urahnen, 813 Befehle für weitere Replikationen

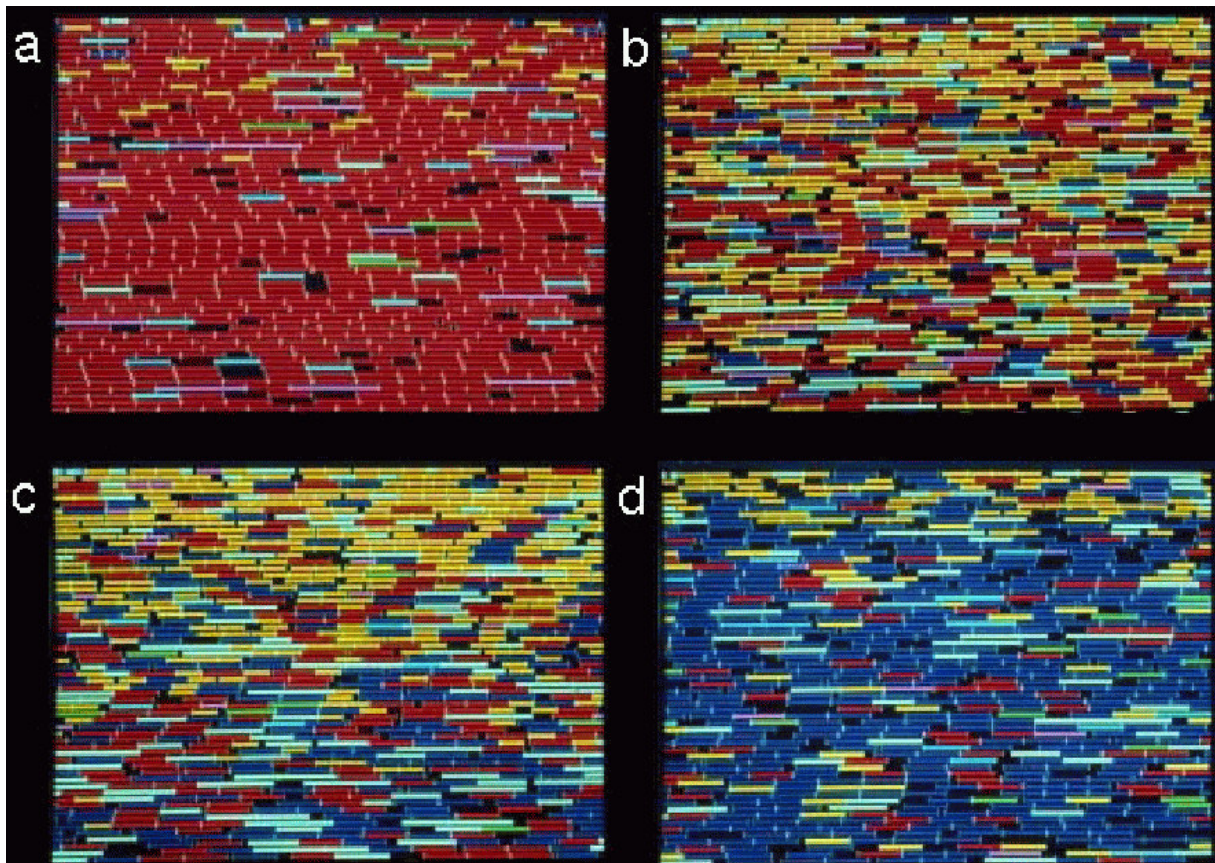
Was passierte beim ersten Evolutions-Lauf (nach 2 Monaten intensiver Programmierarbeit)?

Ray: "Ich dachte: o.k., ich schaffe es, den Simulator zum Arbeiten zu bringen, aber es wird mich Jahre kosten, um eine Evolution in dem System in Gang zu setzen. Wie sich herausstellte, musste ich aber keine weitere Kreatur herstellen..."

- bald entstehen etwas kürzere Mutanten, die den Urahnen der Länge 80 verdrängen
- plötzlich trat ein Organismus der Länge 45 auf, der sich rasch vermehrte! (extrem kurz für ein selbstrepl. Programm in dieser Sprache...) – Ursache: *Parasitismus*
- neuer Organismus verwendet Copy-Schleife seiner größeren Nachbarn für die eigene Fortpflanzung
- oszillierende Populationsgrößen, da Parasit auf Wirtsorganismen angewiesen (Lotka-Volterra-Dynamik)
- später: Auftreten von Immunität bei den Wirtsorganismen ("evolutionärer Rüstungswettlauf")
- Umgehen von Immunität durch Parasiten

- Hyper-Parasiten, die die Parasiten ausnutzen
- Hyper-Parasiten bringen die Parasiten zum Aussterben und entwickeln danach eine Gemeinschaft von "sozialen" Organismen, die sich gegenseitig (in Aggregationen) bei der Vermehrung helfen (und 24% kürzer als der Urahn sind)
- "Betrüger" nutzen die sozialen Organismen aus

Visualisierung des Speicherinhalts von Tierra (erst bei späteren Versionen realisiert):



rot: Urahn (80 Befehle)
gelb: Parasiten
dunkelblau: immune Organismen

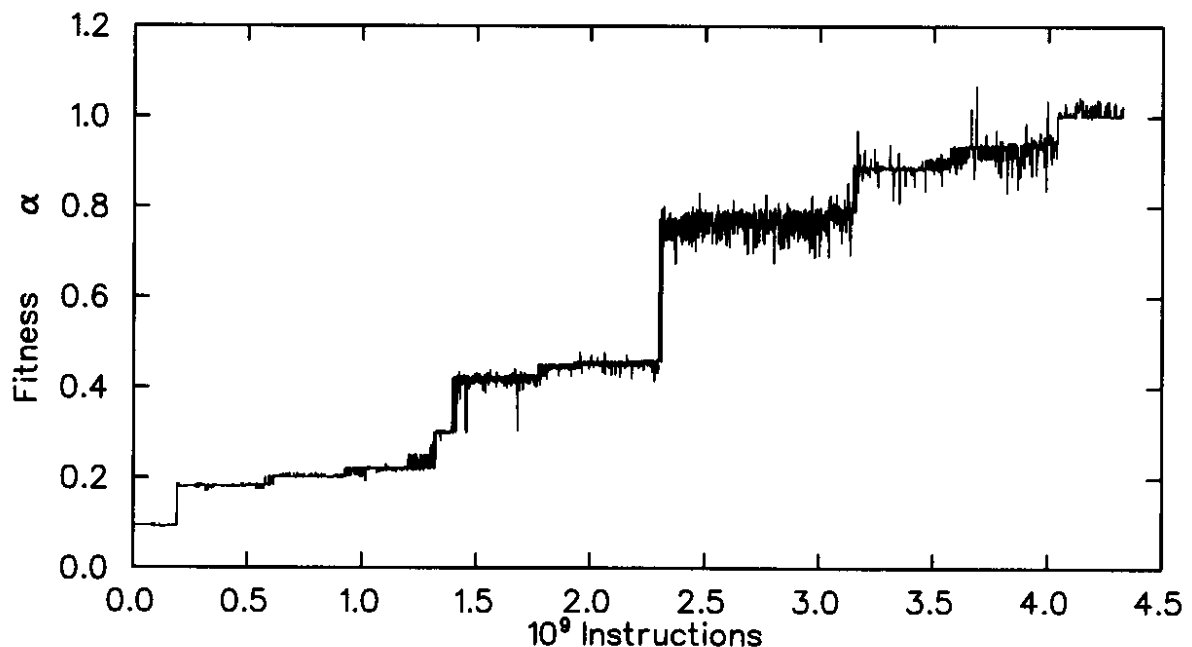
weitere Beobachtung (später):

Auftreten einer neuen Variante der Bestimmung der eigenen Programmlänge: kein template am Ende, dafür in der Mitte; Differenz zwischen Anfang und Mitte wird verdoppelt!

Makro-Evolution:

- Perioden, wo relativ wenig passiert, werden unterbrochen von Phasen starken evolutionären Wandels ("punctuated equilibrium", auch von Biologen konstatiert)

Typischer Verlauf der Fitness des besten Individuums in einem Tierra-Lauf (aus Adami 1999):



- nach vielen Millionen Befehlsschritten werden die "kurzen" Organismen (Länge um 80 Befehle) meist abgelöst durch Organismen mit Länge 400-800.
- bei vereinzelt Läufen kam es auch zum Aussterben der gesamten Population

Fazit:

Auftreten von:

- Koevolution
- Parasitismus, Symbiose
- Immunität (wie bei Bakterien)
- typischer ökologischer Phänomene
- Fitness-Landschaft ändert sich dynamisch durch die vorhandenen Organismen (biotische Umwelt)
- insbes. Einfluss des Parasitismus auf die Fitness
- Anwachsen der Diversität

Ray: "It is worth noting that the results presented here are based on evolution of the first creature that I designed, written in the first instruction set that I designed. (...)
It would appear then that it is rather easy to create life."

Begrenzungen / Nachteile:

- alle Organismen können mit allen interagieren, globaler "reaper" ⇒ fehlende räumliche Struktur
- wegen Schreibschutz der Individuen keine "Räuber", die Speicherplatz okkupieren, möglich
- keine Morphogenese
- Genotyp = Phänotyp

Sourcecode von Tierra verfügbar unter:

<http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/alife/systems/tierra/0.html>

Avida

C. Titus Brown, Charles Ofria, Dennis Adler, Travis Collier,
Christoph Adami

California Institute of Technology, 1994-1997

Homepage: <http://dllab.caltech.edu/avida>

Unterschiede zu Tierra:

- 2-dimensionales Gitter, in jeder Zelle kann (potenziell) ein Individuum lokalisiert sein, mit seinem gesamten Genom
- Interaktionen zwischen den Individuen nur lokal (Moore-Nachbarschaft im Gitter)
- lokale reaper-queues für jede Zelle

Beispiel für einen Avida-Zustand (Großbuchstaben stehen für Genotypen, die mindestens dreimal vorliegen, also vermutlich selbstreplikationsfähig sind, Punkte für andere Mutanten; A war der "Urahn"):

```
. A . .
  A A A . . G G
K A A A A A G A G . G
K K K A . A A A G . G G G G
  K K D A . G A . G . G E G
. A A D A . . G G . E G . G
. A . D A A G C C C . E C C J J
  L A . A . . A A . B G C C E E J
A A A A A A A A D B G E C C E E E .
. L A A . A . . B B E E E . E .
  A . F . A . . A A B E E . . .
  A A + . A . A C C E E C C C . .
  A A A F . C C C C C E E C E E . .
. . C . . C C C C E E E . E
  A C C C C C C C . E . E E
. . C C C C C C C . C E E
  C C C . C . . C C
    C .
```