

Genetische Programmierung  
Am Beispiel der Turingmaschine

Nico Krebs

2005-12-26



# Inhaltsverzeichnis

<b>1</b>	<b>Zusammenfassung</b>	<b>5</b>
<b>2</b>	<b>Biologische Grundlagen</b>	<b>7</b>
2.1	Aufbau und Funktionsweise der DNA . . . . .	7
2.2	Natürliche Kodierung der Erbinformationen . . . . .	8
2.3	Begriffe . . . . .	9
2.4	Abstraktion eines Algorithmus . . . . .	9
2.5	Einführungsbeispiel Rucksackproblem . . . . .	10
<b>3</b>	<b>Genetische Operatoren</b>	<b>15</b>
3.1	Selektion . . . . .	15
3.1.1	Wettkampf-Strategie . . . . .	15
3.1.2	Elitismus(n,m-Wettkampf) . . . . .	16
3.2	Mehr-Punkt-Kreuzung . . . . .	16
3.3	Mutation . . . . .	16
3.3.1	Bitweise Mutation . . . . .	17
3.4	Ersetzung . . . . .	17
3.5	Fitnessfunktion . . . . .	17
<b>4</b>	<b>Genetische Programmierung</b>	<b>19</b>
4.1	Grundlagen: Aufbau Turingmaschine . . . . .	19
4.2	Codierung im Gencode . . . . .	20
4.3	Anpassung der Operatoren . . . . .	22
4.3.1	Kreuzung . . . . .	22
4.3.2	Mutation . . . . .	23
4.3.3	Anpassung Fitnessfunktion . . . . .	23
<b>5</b>	<b>Optimierung</b>	<b>25</b>
5.1	Kodierungsprobleme . . . . .	25
5.2	Gray-Code . . . . .	25
5.3	Einführung verschiedener Spezies . . . . .	26
5.4	Kindergarten-Schema . . . . .	26
5.5	Seniorenunterkunfts-Schema . . . . .	26
<b>6</b>	<b>Grundlegender Aufbau des Programmcodes</b>	<b>29</b>
6.1	Eingabe der Trainingsdaten . . . . .	29
<b>7</b>	<b>Literaturverzeichnis</b>	<b>31</b>



# Kapitel 1

## Zusammenfassung

Dieses Dokument beschreibt die Herangehensweise an die Programmierung einer Turingmaschine mit Hilfe Genetischer Programmierung. Es ist Bestandteil des Projekts *Aicc-bot*, welches im Rahmen einer Belegarbeit bei Prof. Heino Iwe an der **Hochschule für Technik und Wirtschaft Dresden** im Fach **“Genetische Algorithmen”** entstanden ist.

Im zweiten und dritten Kapitel wird auf die grundlegende Funktionsweise der Algorithmen eingegangen. Es wird geklärt, welche natürlichen Gegebenheiten genutzt werden, um computerbasierte Algorithmen zu schaffen, die in der Lage sind, kombinatorische Probleme durch gezieltes Suchen, genauer: mit Mitteln der Evolution, effizient zu lösen.

Danach wird Kapitel 4 einige grundlegende Kodierungstechniken beschreiben. Kapitel 5 widmet sich der Vorgehensweise, GA so zu gestalten, dass sie in der Lage sind, Übergangsfunktionen(Programme) für einfache Turingmaschinen zu ermitteln. Dazu werden Trainingsdatensätze genutzt, die den Input und Output der TM darstellen. Die Verarbeitung übernimmt der GA, d.h. er übernimmt die Aufgabe, aus Input den entsprechenden Output zu erzeugen. Dazu ist es nötig, Genetische Operatoren anzupassen und zu optimieren.

Die wichtigste Aufgabe besteht darin, eine geeignete Fitnessfunktion zu finden, die im Beispiel teils gezielt und teils durch Ausprobieren entstanden ist. Da das hier erläuterte Problem nicht das einfachste ist, was ein Genetischer Algorithmus leisten kann, muss man gesondert auf die Optimierung der einzelnen Techniken eingehen. So wird der Aufbau von Gray-Code erläutert, das Populationsprinzip auf verschiedene Spezies erweitert und eine Elite-Population eingeführt.

Schließlich wird auf den Aufbau des JAVA-Programmes eingegangen, das zur Demonstration erstellt wurde. Die Erweiterbarkeit und der OpenSource-Charakter dieses Projektes wird erläutert. Wer genauere Informationen zur programmier-technischen Umsetzung benötigt, kann sich den vollständig kommentierten Sourcecode ansehen oder eine email an **nicokrebs-pc@web.de** senden.

### **Ergebnis:**

Das Projekt verlief erfolgreich, d.h. es wurde ein GA gefunden, der in der La-

ge ist, Turingprogramme für die Addition zweier unär kodierten Zahlen zu erzeugen. Nach durchschnittlich 600 Generationen wurden funktionsfähige Programme ausgegeben, die im weiteren Verlauf des GA nur noch gekürzt wurden. Bisher konnte aus Zeitgründen nur noch geprüft werden, ob der GA auch in der Lage ist, eine Multiplikation durchzuführen. Das Ergebnis hat mich überrascht, da schon nach durchschnittlich 450 Generationen ein Ergebnis vorlag. Diese Tests müssen aber noch besser verifiziert werden. Der geneigte Leser sei dazu aufgerufen, eigene Testläufe durchzuführen und sie auf der Projektseite [www.mensch-und-maschine.de](http://www.mensch-und-maschine.de) im Gästebuch zu veröffentlichen.

## Kapitel 2

# Biologische Grundlagen

### 2.1 Aufbau und Funktionsweise der DNA

DNS/DNA (Desoxyribonukleinsäure/-Acid[engl. Acid: Säure]) codiert Aminosäuren, die Bestandteile des Lebens. DNA besteht hauptsächlich aus vier Grundmolekülen, den Basen: Adenin, Guanin, Thymin und Cytosin(A,T,G,C) Jeweils zwei von ihnen können miteinander eine Verbindung eingehen: A und T, C und G. Wie wir hoffentlich alle noch aus dem Biologieunterricht wissen, (ansonsten:[Wiki1]-Kapitel 7), besteht DNA aus vielen dieser Brückenbausteine(Basenpaare), der Reihe nach angeordnet.

```
... ATC  GGA  ATC  CCT  ...
...  |||  |||  |||  |||  ...
... TAG  CCT  TAG  GGA  ...
```

Drei dieser “Stufen” bilden ein sog. Codon, also die kleinste Informationseinheit im Gencode. Eine Mutation eines Codons wird Allel genannt. Da man jede Stufe in jeder Richtung in den Genstrang einfügen kann(A-T,T-A;C-G,G-C), ergeben sich  $4^3 = 64$  mögliche Kombinationen, um eine Informationseinheit zu codieren. Da es aber nur 20 Aminosäuren gibt, müssen einige von ihnen mehrfach codiert sein! Biologen haben herausgefunden, welche Kombination welche Aminosäure codiert. Das Ergebnis: Einige sind einfach codiert, einige mehrfach, bis zu sechs mal, um genau zu sein. In der Informatik nennt man dies Form der Kodierung gemeinhin “Redundanz” und wird benutzt, um zum Beispiel häufige Tippfehler zu korrigieren, indem man verschiedene Schreibweisen eines Wortes eindeutig einer richtigen Schreibweise zuordnet. Sicherlich wird die Natur auch hier korrigieren “wollen” und häufig benutzte Aminosäuren bevorzugt auswählen bzw. häufig auftretende Fehler vermeiden.

## 2.2 Natürliche Kodierung der Erbinformationen

Die Informationen über Aminosäuren werden in unseren Genen auf recht einfache Art und Weise kodiert. Jedes Basen-Triplett(drei eufeinanderfolgende Basenpaare) codiert eine Aminosäure:

...	ATC	GGA	ATC	CCT	...
...					...
...	TAG	CCT	TAG	GGA	...
...	Ile	Gly	Ile	Pro	...

(Ile: Isoleucin,Gly: Glycosin, Pro: Prolin)

Aminosäuren sind komplexe Kohlenstoffverbindungen, die, tausendfach zusammengesetzt, einzelne Zellen oder Flüssigkeiten bilden. Die Struktur eines Lebewesens ist sehr komplex, wobei sich der Grad der Komplexität zwischen Tieren/Menschen und Pflanzen um einige Potenzen unterscheidet. Nach genaueren Untersuchungen der Genome verschiedenster Spezies ist man zum vorläufigen Schluss gekommen, dass sich der Aufbau und das Zusammenspiel der Zellen und Organe von Pflanzen sowie Tieren allein aus der Reihenfolge der Basen-Triplets ergeben. Spricht man über Bakterien und einige Pflanzen, so mag das stimmen. Wendet man sich aber tierischen Genomen zu, so wird man sich schnell klar werden, dass die alleinige Anordnung der Gene, was gleichbedeutend wäre mit dem einfachen sequentiellen Ausführen einiger "Bauvorschriften", bei weitem nicht fähig ist, derartig komplexe Strukturen, wie z.B. die eines Gehirns hervorzu-bringen. Das belegt auch die Tatsache, dass die Reispflanze über weitaus mehr Gene verfügt, als der Mensch. und wenn wir sie genauer ansehen, werden wir feststellen, dass sie keine derartige Komplexität erreicht hat, wie der Mensch.

Forscher haben bei ihren Untersuchungen sog. "Müll/Junk-DNS" gefunden, die in allen tierischen Genomen vorkommt. Diese Form der DNS tritt im Wechsel mit gewöhnlicher, proteinkodierender DNS auf und wird Intron genannt, abgeleitet von der Tatsache, dass sie zwischen die kodierenden Teile, die Exons eingeschoben sind. Dieser angebliche Müll füllt 99% unseres Genoms aus. Hier wird deutlich, dass die Forschung bisher in die falsche Richtung verlief. Warum sollte die Natur so viel unnütze Erbinformationen mit sich herumtragen? Neueste Theorien deuten darauf hin, dass diese Informationen keineswegs unnütz sind, sondern eine entscheidende Rolle bei der Herausbildung unterschiedlicher Zellen spielt. Um den Gegenstand der Diskussion zu verdeutlichen folgt nun der schematische Aufbau unserer DNS:

Intron|Exon|Intron|Exon|Intron|Exon|Intron|Exon|Intron|Exon|...

Intronsequenzen werden vor der Herstellung eines Proteins ausgelesen und dafür genutzt, festzulegen, an welcher Stelle das Protein abgelegt werden soll. Man kann diesen Vorgang damit vergleichen, ein Haus zu bauen. Dafür braucht man zwei Dinge: eine Materialliste und einen Bauplan. Würde man diese Liste nach Art der Genomcodierung aufschreiben würde das wohl ungefähr so aussehen:

Material|Ort|Material|Ort|Material|Ort|...

Zement|Boden unter erste und zweite Ziegelposition|Stein|unterste Reihe, erste Position|Stein|unterste Reihe, zweite Position|Zement|unterste Reihe, zwischen erste und zweite Position|...

nach diesem Schema wird ersichtlich, dass viel weniger Information dafür gebraucht wird, welches Material genommen werden muss, als dafür, wohin das Material zu legen ist. Daraus lässt sich schnell erklären, warum der Anteil der Introns in unseren Genen so enorm ist. Wollen wir das Thema hier beenden und die weitere Forschung den Genetikern überlassen.

## 2.3 Begriffe

Man unterscheidet in der Genetik zwischen **Genotyp** und **Phänotyp** eines Individuums.

- Genotyp bezeichnet nichts anderes, als die Darstellung des Lebewesens in verschlüsselter Form, also den Genstrang, der in einer jeden Zelle enthalten ist.
- Phänotyp wird die äußere Form genannt, die entsteht, wenn man Moleküle, wie sie im Genstrang beschrieben sind, anordnet. Damit wird also das fertige “Produkt” Lebewesen bezeichnet.

In der Natur wird diese Übersetzung innerhalb der Zelle vorgenommen und in lange Molekülketten umgesetzt. Die Zelle kann also als molekularer Computer angesehen werden, dessen Eingabedaten die Gene sind und der als Output Proteine produziert. Wir wollen hier nicht klären, woher diese Mechanismen kommen und wozu sie der Natur nutzen sollen, sondern überlassen das lieber der Biologie und Philosophie.

Man kann dieses Modell in zweierlei Richtungen für eigene, informatische, Zwecke gebrauchen. Erstens ließen sich damit “Computer im Reagenzglas” ([StoStu]-Kapitel 7) bauen, die ihrerseits auch mit Molekülen als Input und Output arbeiten. Da wir jedoch nur beschränkt über die Chemie herrschen können, sollte für unser Ziel ein anderer Weg favorisiert werden: Man ahmt den Evolutionsprozess auf heutigen digitalen Computern nach und versucht damit komplexe Problem zu lösen.

## 2.4 Abstraktion eines Algorithmus

Um die Mechanismen der Evolution nutzen zu können, bedarf es natürlich eines Algorithmus. Da wir diesen ja frei abstrahieren können, werden wir ihn so einfach wie möglich gestalten.

1. erstelle einige zufällige Genketten
2. bestimme den Phänotyp der Genketten
3. ermittle, wie gut die Phänotypen ihre Aufgabe erfüllen
4. wähle die besten Individuen aus
5. Kreuze die Gene dieser Individuen und erstelle daraus Kindgene. Mutiere gegebenenfalls

6. Ersetze die schlechtesten Individuen durch die neuen Kindindividuen.
7. Falls Abbruchbedingung nicht erfüllt, Weiter bei Schritt 2, sonst Ende.

Im siebten Schritt muss eine Abbruchbedingung vorliegen. Die Natur hat soetwas natürlich nicht, aber sie ist ja auch kein Programm, das gewisse Probleme lösen und am Ende stehenbleiben soll. Im nachfolgenden Beispiel wird der Algorithmus besser veranschaulicht.

## 2.5 Einführungsbeispiel Rucksackproblem

Sie werden folgendes sicherlich kennen: der Urlaub steht bevor. Neben Kühlschrank, Fernseher, Hifi-Anlage und dem gesamten Kleiderschrank wollen sie auch noch einige Nahrungsmittel, Verbandszeug, Zeitungen, Seife, Töpfe, Waschmittel, Ersatzschuhe etc. in ihrem Rucksack mitnehmen? Das wird problematisch, da Sie als normaler Mensch selten mehr als 20 kg auf dem Rücken tragen wollen. Nun könnten sie den Kühlschrank zu Hause lassen und dafür den Verbandskoffer mitnehmen oder auch den Fernseher stehen lassen und dafür ein gutes Buch mitnehmen. Welche Kombination nun aber die gescheiteste ist, können sie aufgrund der vielen Gegenstände nicht sofort einschätzen.

Der sicherste Weg, die ideale Kombination herauszufinden ist, jedem Gegenstand einen persönlichen Wert zu geben, z.B. einen Wert zwischen 0 und 10 oder zwischen 0 und 100, wie sie wollen. Danach legen sie jedes Stück auf eine Waage und schreiben das Ganze in einer Tabelle auf. Die Letzte Spalte der Tabelle sagt uns, ob wir den Gegenstand mitnehmen(1), oder nicht(0). Unser Ziel ist es nun, die Kombination herauszufinden, die mit 20kg Gesamtmasse den höchsten Wert erreicht. Um eindeutig die beste Kombination an Gegenständen zu ermitteln, müsste man von allen möglichen Bepackungen diejenige finden, die am wertvollsten ist(Summe aller Wertigkeiten der mitzunehmenden Gegenstände), aber nur 20kg wiegt. Das macht bei nur 30 Gegenständen genau  $2^{30}$  verschiedene Kombinationen. Ein moderner Computer vermag evtl.  $2^9$  Rechenoperationen pro Sekunde auszuführen. Das macht bei  $2^{30}$  Rechenoperationen  $2^{22}$  Sekunden, was wiederum rund 48 Tagen entspricht. Schon bei 50 Gegenständen wächst die Rechenzeit auf gewaltige Siebzigtausend Jahre! Wir wollen hier einen wesentlich schnelleren Weg beschreiben. Tabelle zum Beispiel:

Gegenstand	Persönlicher Wert	Gewicht	Mitnehmen?
Taschenmesser	8	0,2 kg	Ja(1)
Kühlschrank	2	40kg	Nein(0)
Verbandskasten	10	0,6	1
...	...	...	...

Man kann diese Tabelle auch als Kette von Nullen und Einsen darstellen:  
 Gegenstand 1 2 3 4 ...  
 Mitnehmen? 1 0 1 1 ...

Sehen wir uns nur die unterste Zeile an, so könnte man diese als Gencode eines Binär-Lebewesens interpretieren, welches eine gepackten Rucksack beschreibt.

Genotyp: 1011101010001101110001100100

Phänotyp(Rucksack):

1. Taschenmesser: mitnehmen
  2. Kühlschrank: nicht mitnehmen
  3. Verbandskasten: mitnehmen
  4. ... :mitnehmen
- ...

Im **ersten** Schritt bedeutet das für unser Beispiel: wir erstellen einige zufällige Bitketten, z.B. acht Ketten der Länge 30, wenn 30 Gegenstände zur Auswahl stehen.

Diese werden im **zweiten** Schritt übersetzt, d.h. Die Bitkette wird Bit für Bit geprüft und im **dritten** Schritt wird die Wertigkeit und das Gewicht jedes Individuums(Rucksacks) ermittelt.. Findet das Programm an der aktuellen Position *Pos* eine 1, dann bedeutet das, der Gegenstand an *Pos* soll mitgenommen werden. Nun sucht man sich in der Tabelle an der angegebenen Stelle Gewicht und Wertigkeit des Gegenstandes heraus und speichert beide Werte. An *Pos+1* wird sich evtl. wieder eine Eins finden, also muss auch dieser Gegenstand mitgenommen werden. Sein Gewicht wird auf das bisherige Rucksackgewicht aufaddiert, seine Wertigkeit auf den bisher erreichten Wert. Wenn die Bitkette vollständig geprüft wurde, wird zur nächsten übergegangen und der Vorgang wiederholt. Am Ende haben alle Rucksäcke ein spezifisches Gewicht und einen Wert.

Im **dritten** Schritt wird für jedes Individuum ein Fitness-Wert errechnet. Dieser gibt an, wie gut, oder *fit* ein Individuum ist. In unserem Fall entspricht die Fitness eines Rucksacks genau seinem Wert, d.h. je höher der Wert eines Rucksacks ist, desto fitter ist er.

**Viertens** werden alle Individuen nach ihrer Fitness geordnet und die N besten für die Fortpflanzung ausgewählt. Uns genügen erst einmal 4 Individuen.

Im **fünften** Schritt wird die Kreuzung vollzogen. Einfaches Bsp:

Kette 1: 0101011110011001010100001100

Kette 2: 0111010110100011101011001001

Es wird zufällig ein Schnittpunkt ausgesucht.(z.b. die Mitte), dann wird der erste Teil der ersten Bitkette und der zweite Teil der zweiten zu Kind Nr. 1 zusammengebaut. Aus den anderen Hälften wird ein zweites Kind erstellt:

Kette 1: 010101111001100|101010000110011

Kette 2: 011101011010001|110101100100101

Kind 1: 010101111001100|110101100100101

Kind 2: 011101011010001|101010000110011

Mutation wird realisiert, indem man für jede Stelle im Gencode zufällig eine Zahl *A* zwischen 0 und 1 bestimmt. Sollte  $A \leq 0,5$  sein(was bei gleichverteilten Zufallszahlen in 50% der Versuche der Fall sein sollte), wird an der aktuellen Position ein Bitflip durchgeführt, d.h. Aus einer Eins wird eine Null und umgekehrt. Mit *A* kann man die Anzahl der Mutationen in einem Gen steuern. Wenn *A* klein ist( 0,03) dann wird selten mutiert, ist es groß, dann häufiger. Mutationen dienen dazu, neue Informationen in das Gen zu bringen.

Im **sechsten** Schritt werden die schlechtesten Individuen aus der Population entfernt und die Kindgene hinzugefügt. Normalerweise sollte die Zahl der aus der Population entfernten Individuen der Anzahl der Kind-Individuen entsprechen, um einen konstanten Fortlauf der Evolution zu gewährleisten.

Im **siebten** Schritt wird eine Abbruchbedingung abgefragt. In unserem Beispiel soll nach 500 Generationen Schluss sein und das bis dahin beste gefundene Individuum als Ergebnis präsentiert werden. Natürlich könnte man auch kompliziertere Abbruchbedingungen festlegen, wie z.B. dass erst aufgehört werden soll, wenn der beste Fitnesswert in der Population nach einer gewissen Zahl von

Durchläufen keine oder nur geringe Änderungen erfährt.

Ein Beispieldokument, mit Openoffice-Calc und der integrierten Makro-Programmiersprache erstellt, liegt dem Aiccbot-Paket bei[OOoMak]7. Das Dokument bietet die Möglichkeit, Gegenstände und deren zugehörige Wertigkeit und Gewicht einzutragen. Ein GA wird nun damit beauftragt, geeignete Kombinationen zu finden. Dabei kann man mit der Anzahl der Durchläufe experimentieren. Ein Diagramm stellt die teilweise starken Fitnesssprünge dar und verdeutlicht so die relative Zufälligkeit aber auch die stetige Verbesserung der Ergebnisse.



## Kapitel 3

# Genetische Operatoren

Unter Genetischen Operatoren versteht man die Möglichkeiten, die zur Kreuzung und Mutation von Individuen zur Verfügung stehen. Im Beispiel des Rucksackproblems wurden schon einige genannt: Ein-Punkt-Kreuzung, Mutation und Fitnessfunktion. Da die kleinste Informationseinheit in dem Fall die Länge 1 Bit besitzt, sind Kreuzung, Mutation und Fitnessfunktion recht einfach zu gestalten.

### 3.1 Selektion

Werfen wir wieder zuerst einen Blick auf die Natur: Es ist zwar im Großen und Ganzen so, dass Darwins Lehre sich als richtig erwies, d.h. der Stärkere, oder auch "Fittere" macht das Rennen in Sachen Fortpflanzung. Man muss jedoch auch bemerken, dass durchaus auch weniger starke Lebewesen die Möglichkeit bekommen, ihr Erbgut weiter zu geben. Dieser Mechanismus macht durchaus Sinn, denn ohne ihn würde sich die Population schnell angleichen, nahezu identisches Erbgut könnte in allen Lebewesen einer Population gefunden werden. Wir kennen dieses Phänomen als "Darwin-Finken".

In der GP wollen wir ähnlich vorgehen und nicht nur die  $N$  besten Individuen zur Fortpflanzung heranziehen, sondern auch mit geringerer Wahrscheinlichkeit schlechte Individuen vermehren. Das sichert der Population die nötige Vielfalt und vermeidet die Konvergenz auf ein lokales Maximum, indem weite Teile des Suchraums erforscht werden.

#### 3.1.1 Wettkampf-Strategie

Bei der Wettkampf-Selektion werden in jedem Schleifendurchlauf  $N$  Individuen einer Population ausgewählt. Das Beste von diesen Individuen wird gespeichert und der Durchlauf wiederholt. Die Rekursion wird so oft durchgeführt, wie Individuen ausgewählt werden sollen. Jedes ausgewählte Individuum sollte aus dem Pool entfernt werden, um zu gewährleisten dass nicht zweimal das gleiche Individuum ausgewählt wird.

### 3.1.2 Elitismus(n,m-Wettkampf)

Aus der Population der Größe  $m$  sollen  $n$  Individuen ausgewählt werden. Ist die Zahl der zum Wettkampf herangezogenen Individuen  $n$  gleich der Größe der Population  $m$ , so werden nur die besten Individuen ausgewählt und man spricht von einer  $n,m$ -Wettkampf-Selektion. Diese ist gleichzusetzen mit der einfachsten Form der Auslese: Elitismus. Dabei wird die Population nach den Fitnesswerten (größter zuerst) der Individuen sortiert und die ersten  $N$  Individuen ausgewählt.

## 3.2 Mehr-Punkt-Kreuzung

Unter Multipoint-Crossover versteht man im einfachsten Fall (Ein-Punkt-Kreuzung) das "zerschneiden" zweier Gencodes, die danach zu zwei neuen Kind-Individuen zusammengefügt werden, wobei die beiden Hälften der Eltern vertauscht werden (siehe Rucksack).

Man kann diesen Fall erweitern und zwei oder mehr Bruchpunkte bestimmen. Zwei Bruchpunkte erzeugen somit drei Teile, die sich vertauschen lassen und zu theoretisch drei Kind-Individuen führen könnten usw.

Wir wollen aber immer nur zwei Kind-Individuen erzeugen und gehen daher folgendermaßen vor:

Der Gencode bestehe aus  $N$  Allelen, mit  $N=(1,2,3,\dots)$ . MP-Crossover kann auf maximal  $N$  Allele angewendet werden, d.h. es darf an höchstens  $N-1$  Stellen im Gencode geschnitten werden, um keinen Index-Fehler im Programm zu erzeugen. Ein Gencode bestehe aus 5 Allelen, welcher an vier Stellen gekreuzt werden soll. (Allele mit Buchstaben gekennzeichnet)

Elter 1: **A** | **B** | **C** | **D** | **E**

Elter 2: **F** | **G** | **H** | **I** | **J**

Es entstehen pro Elternteil (Elter) 5 Teile, die nun neu zusammengefügt werden müssen. Um flexibel in der Anzahl der Kreuzungspunkte zu bleiben, habe ich mich dafür entschieden, jedes ungerade Allel des ersten Elters und jedes gerade des zweiten Elters in das erste Kind-Gen zu kopieren. Für das zweite Kind-Gen wird der Ablauf umgekehrt.

Kind 1: **A** | **G** | **C** | **I** | **E**

Kind 2: **F** | **B** | **H** | **D** | **J**

## 3.3 Mutation

Unter Mutation versteht man das zufällige Verändern einzelner Genteile um dadurch neue Informationen in das Genom schleusen zu können.

### 3.3.1 Bitweise Mutation

Es werden zufällig  $n$  Positionen im Gencode ermittelt. An diesen Stellen wird ein Bitflip durchgeführt. (Siehe Rucksack)

## 3.4 Ersetzung

Die Ersetzung der Individuen einer Population wird ebenfalls mittels Wettkampfschema durchgeführt. Es werden wiederum  $n$  Individuen ausgewählt. Diesmal wird jedoch das Schlechteste ausgewählt und aus dem Pool entfernt, bevor die Schleife wiederholt durchlaufen wird. Wenn ebenfalls  $N$  Rekursionen durchgeführt werden, bleibt die Größe der Population konstant und somit kontrollierbar. Die Anwendung hat gezeigt, dass es von Vorteil ist, einige der schlechtesten Individuen nicht sofort aus der Population zu löschen.

## 3.5 Fitnessfunktion

Eine Fitnessfunktion ist eine Gleichung, die festlegt, wie "gut" ein Individuum ist. In der Natur wird dieser Wert ermittelt, indem das Individuum heranwächst und sich in seiner Umwelt behauptet. Im Computer müssen wir ähnlich vorgehen: wir werden den Gencode in seine phänotypische Form umwandeln und daraus seine Fitness ableiten. Beim Rucksackproblem auf Seite 10 wird das gelöst, indem die Wertigkeiten aller mitzunehmenden Gegenstände summiert werden. Um aber ein großes Spektrum der mit GA lösbaren Probleme abzudecken folgt nun eine allgemeine Herangehensweise für Fitnessfunktionen:

Grundsätzlich sollte ein hoher positiver Fitnesswert auf eine gute Fitness schließen lassen. Jeder Einflussfaktor kann einzeln gewichtet werden. Damit erreicht man, dass es in der Fitnessfunktion wichtige und eher unwichtige (und welche dazwischen) Einflüsse gibt. Soll ein zahlenmäßig großer Faktor für einen relativ kleinen Fitnesswert sorgen, so sollte dieser als Kehrwert in die Funktion eingehen.

Allgemein gilt:

$$\begin{aligned}
 &F(a_1, \dots, a_n) \dots \text{Fitnessfunktion} \\
 &g_i \dots \text{Wichtung der Faktoren} \\
 &a_i \dots \text{Faktoren der Fitnessfunktion} \\
 & \quad i = (1, 2, 3, 4, \dots, n) \\
 &F(a_1, \dots, a_n, g_1, \dots, g_n) = \sum_{i=1}^n (g_i * a_i)
 \end{aligned}$$

Die Gewichte sollten so gewählt werden, dass der Fitnesswert den größtmöglichen Zahlenbereich nicht überschreitet. So ist man bei einer Größenordnung von 32 Bit zwar nicht gerade begrenzt, doch wenn sich jemand entscheiden sollte, die Funktion folgendermaßen zu erweitern, stößt man rasch auf Grenzen der 32-Bit-Technik:

$$\begin{aligned} & b_i \dots \text{Exponenten für jeden Faktor} \\ & i = (1, 2, 3, 4, \dots, n) \\ F(a_1, \dots, a_n, g_1, \dots, g_n) &= \sum_{i=1}^n (g_i * a_i^{b_i}) \end{aligned}$$

Diese Form der Fitnessfunktion hat einen entscheidenden Vorteil: mittels des Exponenten  $b$  lässt sich die Streuung der Fitnesswerte weitaus stärker beeinflussen. Die Entropie nimmt drastisch zu, was in einigen Fällen erwünscht ist, beispielsweise, wenn statistische Faktoren die Fitness beeinflussen. Anstelle des Kehrwertes kann ein Wert auch negativ in die Gleichung eingehen, wenn er dafür sorgen soll, die Fitness zu verringern, wenn er groß ist und zu erhöhen, wenn er klein ist.

# Kapitel 4

## Genetische Programmierung

### 4.1 Grundlagen: Aufbau Turingmaschine

Man stelle sich einen Kassettrecorder vor. Er verfügt über ein Band und einen Schreib-/Lesekopf, welcher mit dem Band arbeiten kann.

Das Band T(Tape) ist theoretisch unendlich lang und enthält Zeichen eines Alphabets. Es wird mit 0 beginnend indiziert und nach links mit negativen und nach rechts mit positiven Indizes markiert. Somit befindet sich der Anfang in der Mitte des Bandes.

Arbeitsweise:

StartPosition: 0, StartZustand: 0, HaltZustand: H

1. Lies Zeichen an Position 0
2. Lies aktuellen Zustand aus
3. Suche aus Regelwerk eine passende Regel
4. Wende Regel an, d.h. Schreibe ein Zeichen  $S$ , wechsele in Zustand  $Z$  und bewege Band eine Position nach links oder rechts.
5. wenn  $Z$  ungleich Haltzustand, beginne wieder bei 1., sonst Halte an (Verarbeitung abgeschlossen)

Hier ein Beispiel:

Das Band enthält folgende Zeichen an den Positionen(0-9) :

\_\_\_\_\_111+11111=\_\_\_\_\_

...10123456789....(Positionsnummern)

Die Unterstriche stehen für das Leere Zeichen (Leerzeichen). Ich habe mich für den Unterstrich als Symbol für das Leere Zeichen entschieden, weil er am deutlichsten ist und selten gebraucht wird. Beispielsweise kann man für die Addition folgendes Regelwerk erstellen:

Eingabezeichen	Akt. Zustand	Ausgabezeichen	Folgezustand	Bewegungsrichtung
' 1 '	0	1	0	R
' + '	0	1	1	R
' = '	1	_	2	L
' 1 '	1	1	1	R
' 1 '	2	_	H	L

In Kurzform: (weitere Möglichkeit:)

1,0 : 1,0,L
+,0 : 1,1,R
=,1 : _,2,L
1,1 : 1,1,R
1,2 : _,H,L
1,0 : 1,0,R
+,0 : 1,0,R
=,0 : _,1,L
1,1 : _,H,L

Beide Regelwerke erzeugen aus einem Ausdruck der Form:  $111+11111=$  einen Ausdruck der Form:  $11111111\_ \_$   
 Man kann das als unäre Addition verstehen, d.h. 111 steht für drei, 11111 für fünf. In dezimaler Schreibweise also "3+5=".  
 Das Ergebnis kann wieder durch Zählen der Einsen ermittelt werden: acht Einsen=8, d.h. 3+5=8.

Damit ergeben sich fünf Alphabete:  $\Sigma_I$  Eingabe-Alphabet

$\Sigma_O$  Ausgabe-Alphabet

$\Sigma_{ZA}$  Alphabet aller lesbaren Zustände

$\Sigma_{ZF}$  Alphabet aller möglichen Folgezustände ( $\Sigma_{ZF} = \Sigma_{ZA} + \Sigma_{Ze}$ );  $Ze = \text{Endzustand}$

$\Sigma_M$  Alphabet aller möglichen Bewegungsrichtungen

Nun kann man sich einen Vektor zurecht definieren, der eine vollständige Turing-Regel darstellt:

$$\vec{T}_R = \{S_I, S_{ZA}, S_O, S_{ZF}, S_M\}$$

$$\text{mit } S_I \in \Sigma_I, S_{ZA} \in \Sigma_{ZA}, S_O \in \Sigma_O, S_{ZF} \in \Sigma_{ZF}, S_M \in \Sigma_M$$

Ein weiterer Vektor kann definiert werden, der die Anzahl der Zeichen jedes Alphabets enthält.

$$\vec{T}_A = \{|\Sigma_I|, |\Sigma_{ZA}|, |\Sigma_O|, |\Sigma_{ZF}|, |\Sigma_M|\}$$

Ein vollständiges Turingregelwerk kann wiederum als Vektor verstanden werden:

$$\vec{T}_A = \{|\Sigma_I|, |\Sigma_{ZA}|, |\Sigma_O|, |\Sigma_{ZF}|, |\Sigma_M|\}$$

## 4.2 Codierung im Gencode

Turing-Regeln sind blockweise angeordnet, wobei jeder Block einem Codon entspricht und somit eine Informationseinheit kodiert. Hier ein Beispiel einer Bitkette:

(Addition)

benötigte Alphabete:

$$\Sigma_{\text{Eingabe}} = \{1, +, =\}$$

$\Sigma_{Akt.Zustand} = \{1, 2, 3, 4\}$   
 $\Sigma_{Ausgabe} = \{1, \_ \}$   
 $\Sigma_{Folgezustand} = \{1, 2, 3, 4, H\}$   
 $\Sigma_{Bewegungsrichtung} = \{R, L\}$

Regelformat:

[<Input> ',' <akt. Zust> ':' <Output> ',' <Folgezust.> ',' <Bew.-richtung>]						
...	...	Regel k	Regel k+1	...	...	(Formal)
...	...	011 00 101 00 1	101 10 010 01 0	...	...	(Genotyp)
...	...	'1' '3' ' _ ' 'H' 'R'	'+' '1' '1' '2' 'L'	...	...	(Phänotyp)
...	...	1 , 3 : _ , H , R	+ , 1 : 1 , 2 , L	...	...	(Turing-Programm)

Um den Genotyp in den Phänotyp zu konvertieren, wird der Gencode codonweise, jedes Codon wiederum Blockweise für jedes Zeichen bearbeitet. (Als Block wird hier jeweils ein Untercodon eines Codons bezeichnet)

Ist das getan, kann der Binärcode jedes Blocks in Dezimalcode umgewandelt und als Indexnummer genutzt werden. Ist diese Zahl größer, als die Anzahl der für die aktuelle Information möglichen Zeichen, so wird der Modulo gebildet und ein neuer Index innerhalb des Alphabets ausgewählt. Wie sie sehen, ergibt sich hierbei eine doppel-Codierung, welche man vorteilhaft für seine Zwecke nutzen, die aber auch störend wirken kann.

Wir wollen Gene weiterhin in Form von Bitketten darstellen, d.h. jedes Zeichen eines Alphabetes muss mittels einer Kombination aus Nullen und Einsen dargestellt werden. Dabei ergeben sich zwei Probleme:

1. Wieviele Bits sind notwendig, damit alle Zeichen eines Alphabets codiert werden können?

2. Davon ausgehend, dass man mit einer Bitkette der Länge  $c$  genau  $2^c$  Kombinationen erhält, kann man auch genauso viele Zeichen damit codieren. Was passiert, wenn die Anzahl der Zeichen nicht genau einer ganzzahligen Potenz von 2 entspricht?

Sind alle Alphabete ermittelt und als Variable vorliegend, so kann man mittels einer Schleife die Anzahl der zur Codierung nötigen Bits bestimmen:

```

k...Anzahl der Zeichen
n=0;
while(k >= 2^n AND k < 2^(n+1)) do:(n=n+1)

```

Wenn die Schleife abbricht, wird  $(n+1)$  zurückgegeben.

Man könnte jetzt der Einfachheit halber jedem Zeichen eine fortlaufende, ganze Zahl  $j$  zuordnen und diese dann in binären Code umsetzen. Damit erhält man die gewünschte Menge an Codierungs- Kombinationen. Sollte  $j$  dabei größer werden, als die Anzahl der Zeichen  $k$ , wird einfach der Modulo von  $k$  und  $j$  gebildet (Index =  $k \bmod j$ ). Dabei kann man sicher sein, dass höchstens  $k-1$  Zeichen doppelt codiert sind, weil  $k$  immer zwischen zwei Zweierpotenzen  $a$  und  $b$  liegt, und der Modulo laut Definition nie ein Ergebnis größer als  $b$  liefern kann. Sind z.B. 5 Zeichen zu codieren, so braucht man mindestens drei Bits, was aber 8 Kombinationen ermöglicht. Leider gibt es keine halb-, oder drittel-Bits, die uns genau 5 Kombinationen ermöglichen würden, also muss man sich ein wenig

Hilfe bei der Natur holen.

Wenn mehr Kombinationen existieren, als Zeichen, ordnen wir einigen Zeichen einfach mehrere Bitkombinationen zu:

Bsp: 5 Zeichen = 3 Bit = 8 Kombinationen, binär aufsteigend geordnet(000=0,001=1,...,110=6,111=7)

Zeichen	Kombination
a	000;101
b	001;110
c	010;111
d	011
e	100

Im Falle der GP soll einer gegebenen Bitkette ein ganzes Regelwerk zugeordnet werden. Daher muss sein Gencode blockweise interpretiert werden, was einen schematischen Aufbau erfordert. Aufgrund der o.g. Definitionen der Turing-Alphabete, kann man exakt bestimmen, wieviele Bits benötigt werden, um ein Zeichen zu codieren.

Hat man die für jedes Alphabet benötigte Bitzahl ermittelt, kann man errechnen, aus wievielen Bits eine Regel(ein Codon) bestehen muss, indem man alle Elemente des Vektors  $\vec{T}_A$  aufsummiert. Daraus wiederum ergibt sich die Gesamtlänge der Gencodes, wenn die Zahl der Regeln bekannt ist. Die Startpopulation muss demnach so initialisiert werden, dass genau  $n \cdot (\text{Codonlänge})$  Bits generiert werden. Eine Schleife, die bei jedem Durchlauf zufällig eine Eins oder Null an den Gencode anhängt, erfüllt diesen Zweck sehr gut.

Man könnte die Initialisierung der Population noch verfeinern, indem man keine feste Codon-Anzahl benutzt, sondern diese für jedes Individuum zufällig (immer als ganzes Vielfaches der Codonlänge) bestimmt, aber wir begnügen uns mit einer festen Start-Größe.

### 4.3 Anpassung der Operatoren

Da es sich bei der Genetischen Programmierung um größere Informationsblöcke handelt werden wir einzelne Informationen in einem Codon codieren. Dadurch können o.g. Operatoren fast vollständig übernommen werden. Es werden neue Operatoren für die Mutation eingeführt und der Aufbau der Fitnessfunktion verdeutlicht.

#### 4.3.1 Kreuzung

Da die meisten Anwendungen Genetischer Algorithmen, wie auch die der Genetischen Programmierung nicht so einfacher Art sind wie im Beispiel des Rucksackproblems, werden wir den Begriff des Codons einführen. Ein Codon ist eine Mehrstellige, informationscodierte Einheit im Gencode, d.h. es werden mehrere Bits zusammengefasst, um eine Information zu codieren. Dieses Prinzip ist hier von der Natur direkt übernommen und kann auf Seite 8 am Anfang des Kapitels "Natürliche Kodierung der Erbinformationen" nachgelesen werden.

Angewandt auf die Kreuzung zweier Individuen, die ein komplettes Turing-Regelwerk darstellen, darf ein Genom nur an ganz bestimmten Stellen geteilt werden, nämlich an den Grenzen eines ganzen Codons. Das ist nötig, um zu

gewährleisten, dass nur syntaktisch korrekte Programme entstehen, da jede einzelne Regel eines Regelwerkes der Turingmaschine eine Informationseinheit darstellt. Das Genom bestehe aus  $N$  Codons. Dadurch dürfen maximal  $N-1$  Bruchpunkte existieren, an denen gekreuzt wird. Da es zusätzliche Operatoren gibt, die den Gencode Codonweise kürzen, muss vor jedem Kreuzungsprozess die Codonanzahl des Genoms ermittelt oder wie im Programm in einer eigenen Individuum-Klasse gespeichert werden.

### 4.3.2 Mutation

Die einfache, bitweise Mutation des Gencodes ist für Zwecke der GP nicht ausreichend. Da Programme bekanntermaßen unterschiedliche Längen haben, müssen Mutationsoperatoren eingeführt werden, die Teile in den Code einfügen oder aus ihm heraustrennen.

#### Kürzen des Gencodes

Es wird zufällig ein Codon ausgewählt, welches aus dem Code entfernt wird. Der Operator kürzt das Programm und sorgt somit dafür, dass evtl. sinnfreie Informationen aus dem Programm verschwinden, bzw. dass der Gencode darauf vorbereitet wird, ein kürzeres Programm entwickeln zu müssen. Die entsprechenden Änderungen der einzelnen Regeln sollen dann erfolgen, indem Mutationsoperatoren ausgeführt werden.

#### Hinzufügen eines Codons

Per Zufallsgenerator wird eine Position  $P$  im Gencode ermittelt. Danach wird eine Bitkette zufällig erzeugt und an der ermittelten Stelle  $P$  eingefügt. So gelangt neue Information in den Genpool. Der Operator ändert die Programmlänge und bringt neue Informationen in das Chromosom.

#### Invertieren eines Codons

Das Invertierungssystem wurde auch in der natürlichen Mutation nachgewiesen und sollte daher Einzug in die GP halten. Es wird zufällig ein Codon aus dem Gencode ausgewählt und herausgetrennt. Alle Bits werden in umgekehrter Reihenfolge wieder an die gleiche Codon-Position geschrieben.

### 4.3.3 Anpassung Fitnessfunktion

Um eine wirksame Fitnessfunktion zu finden, die ein Turingprogramm sachgemäß einstuft, bedarf es einiger Überlegung und ein wenig Probierarbeit.

Es muss überprüft werden, wie gut das produzierte Ergebnis an das gesuchte herankommt. Dazu braucht man eine statistische Funktion, die das Maß der Übereinstimmung zweier Strings misst.

Weiterhin muss bei Ausführung des Turingprogramms mitgezählt werden, wieviele Rechenschritte benötigt wurden, um das gesamte Programm auszuführen. Dabei kann auch gleich ermittelt werden, wieviele verschiedene Programmzeilen(Regeln) zur Ausführung benötigt wurden. Natürlich muss man auch über

das Wissen verfügen, wieviele Regeln das gesamte Regelwerk nun eigentlich umfasst.

auch ist es notwendig, sog. Nullbedingungen einzuführen. Diese dienen dazu, keinesfalls lauffähige Programme mit einer Fitness von Null zu belegen. Dazu gehört die Syntaktische Korrektheit der Regeln, d.h. die Alphabete der geforderten und der produzierten Ausgabe werden verglichen. Sollten nicht alle gewünschten Zeichen in der produzierten Ausgabe vorkommen, wird die Fitness auf Null gesetzt. Weiterhin dürfen keine Zeichen der Eingabe auf dem Band verbleiben, sofern sie nicht auch im Alphabet der Ausgabe vorkommen. Sollte sich keine einzige Regel anwenden lassen, soll auch dies entsprechende negative Auswirkungen auf die Fitness haben.

Natürlich ist es wichtig, Programme zu honorieren, die ein Ergebnis liefern, dass völlig mit dem gewünschten übereinstimmt. der Faktor 5000 hat sich hierbei als ausreichend deutlich erwiesen.

Hilfreich ist es auch, das völlige Fehlen falscher Regeln mit einer Verdopplung der Fitness zu honorieren.

Mit diesen Informationen und einigen Gewichtungsfaktoren(durch Probieren ermittelt) kann man eine Fitnessfunktion gestalten, die nach durchschnittlich 600 Generationen(Addition) bzw. 450(Multiplikation) ein funktionierendes Programm ausgibt.

$$\begin{aligned}
 &F(a_1, \dots, a_n) \dots \text{Fitnessfunktion} \\
 &g_i \dots \text{Wichtung der Faktoren} \\
 &a_i \dots \text{Faktoren der Fitnessfunktion} \\
 & i = (1, 2, 3, 4, \dots, n) \\
 &F(a_1, \dots, a_n, g_1, \dots, g_n) = \sum_{i=1}^n (g_i * a_i)
 \end{aligned}$$

$a_1$ ... Übereinstimmung des Zeichensatzes auf dem Band mit gewolltem Output-Zeichensatz

$a_2$ ... Zahl der benötigten Rechenschritte  $a_2 = a_2^{-1}$

$a_3$ ... Anzahl der Regeln  $a_3 = -a_3$

Nullbedingungen:(unerlaubte Programme etc.)

$n_1$ ... Zeichen, die vorkommen sollen, aber nicht produziert werden

$n_2$ ... mindestens ein unerlaubtes Zeichen aus Input auf Band verblieben

$n_3$ ... keine gültige Regel enthalten

Gewichte der Faktoren :

$$g_1 = 100$$

$$g_1 = 100$$

$$g_1 = 1$$

Algorithmus zur Fitness-Ermittlung eines Individuums: { Für jeden Trainingsdatensatz : WENN ( $n_1$  ODER  $n_2$  ODER  $n_3$  erfüllt) DANN : Fitness = 0

SONST : Fitness = Fitness +  $\sum_{i=1}^3 (g_i * a_i)$

Fitness =  $\frac{\text{Fitness}}{\text{AnzahlTrainingsdatensätze}}$  WENN ( $a_1 = 1$ ) DANN : Fitness = Fitness\*

5000 WENN (Anzahl der benutzten Zeilen =  $a_3$ ) DANN : Lösung gefunden! (GA - Abbruch)

# Kapitel 5

## Optimierung

### 5.1 Kodierungsprobleme

Die Interpretation der Bits als Dualzahl kann im Falle der GP einige Gefahren bergen. Da der Gencode eines Individuums nur mit geringer Wahrscheinlichkeit mutiert wird, werden sich demnach auch nur wenige Bits durch Mutation verändern. Wie man aber deutlich sehen kann, muss man für die Änderung der Kombination von Zeichen 'd'(011) zu Zeichen 'e'(100) drei Bits verändern, obwohl man in der Tabelle nur um einen Platz nach unten rückt. Das macht es fast unmöglich, dass durch Mutation eine kleine Änderung im Phänotyp, also im resultierenden Turing-Programm, hervorgerufen wird. Die Anzahl der sich ändernden Bits von einer Bitkombination auf eine andere nennt man "Hamming-Klippe". Im Beispiel von d nach e beträgt sie drei bits. Je länger die Bitketten werden, desto mehr fällt dieser Effekt ins Gewicht und beeinflusst die Lauffreudigkeit des GA negativ.

### 5.2 Gray-Code

Messgeräte, die mit binärer Codierung arbeiten, haben ebenfalls ein großes Problem damit, Änderungen von mehr als einem Bit auf einem Messstreifen zu registrieren. Daher wurde einst der Gray-Code entwickelt. (Frank Gray, Patent: 1947)

Er besitzt die Eigenschaft, dass er zwar auch eine binäre Darstellung natürlicher Zahlen ist, jedoch ändert sich beim Sprung von einem Wert auf den nächsten grundsätzlich nur ein Bit.

Man geht dazu von dualer Codierung aus und übersetzt diese in Graycode, indem man immer zwei Bit mittels XOR vergleicht und das Ergebnis des Vergleichs als neues Gray-Bit setzt. Das erste Bit wird immer als erstes Gray-Zeichen übernommen. Im Nachhinein wird die Bitkette noch invertiert, um die Gray-typische Darstellung herzustellen.

Bsp: (Gray to Bin):

$$\begin{array}{rcl}
 & & g_i \rightarrow b_j \\
 0. & 1 & \rightarrow \quad (b_0 = b_0 = 1) \\
 1. & 0 & \rightarrow (b_1 = (0 \text{ XOR } 1) = 1) \\
 2. & 1 & \rightarrow (b_2 = (1 \text{ XOR } 1) = 0) \\
 3. & 1 & \rightarrow (b_3 = (1 \text{ XOR } 0) = 1) \\
 4. & 0 & \rightarrow (b_4 = (0 \text{ XOR } 1) = 1) \\
 & & \text{invert (Gray)} \Rightarrow \text{Bin'ar : 11011}
 \end{array}$$

Bei der Umwandlung von Gray- nach Binär-Code wird nicht innerhalb der Gray-Bitkette, sondern immer das aktuelle Gray-Bit mit dem zuletzt erzeugten Binär-Bit mittels XOR verglichen. Vorher muss der Gray-Code invertiert werden, um die umgekehrte Anordnung der Bits einfach zu realisieren.

### 5.3 Einführung verschiedener Spezies

In der Natur kann man ein grundsätzliches Vorgehen beobachten: jede Tier- und Pflanzenart hat ihren Zweck. Betrachtet man die Überlebensstrategien von Ameisen, die sich Läuse "halten" um sich von deren zuckerhaltigen Ausscheidungen zu ernähren, dann wird klar, dass jede Spezies eine eigene Aufgabe hat und diese als hochspezialisierter Profi erfüllt.

Um dieses System auf GP zu übertragen, ist es sinnvoll, eine art "Welt" zu programmieren, die unterschiedlichsten Spezies und deren Untergattungen Platz bietet. Dazu werden für jede Spezies u Unterpulationen erstellt. Individuen zweier Unterpulationen können sich nicht direkt durchmischen, d.h. ihre Entwicklung läuft getrennt ab. Weiterhin bedarf es einer Elite-Population, die die besten Individuen aller Unterpulationen enthält. Führt man den GA auch auf dieser Elite-Population aus, sollte sich dort die beste, globale Lösung finden. Alle Versuche mit dem GP-System haben die Wirksamkeit dieses Systems belegt.

### 5.4 Kindergarten-Schema

Um die Konvergenzgeschwindigkeit des GA weiter zu erhöhen, wird ein sog. Kindergarten-Pool eingeführt. In jeder Generation werden k Individuen in den KiGa-Pool kopiert. Diese werden p Generationen lang darin aufbewahrt, ohne dass sie am Selektionsprozess teilnehmen dürfen. In jeder Generation wird versucht, die gespeicherten "Kinder" durch Mutation zu verbessern. Sollte sich eine Verbesserung einstellen, wird ein schlechtes Individuum in der "richtigen" Population durch das mutierte Kind ersetzt. Es befinden sich demnach stets  $p \cdot k$  Individuen im Kindergarten.

### 5.5 Seniorenunterkunfts-Schema

Wie im richtigen Leben sollte es Älteren, aber fitten Individuen erlaubt sein, etwas länger am Selektionsprozess teilnehmen zu dürfen. Dazu wird ein Pool,

ähnlich dem Kindergarten erstellt, in den in jeder Generation die  $m$  besten Individuen eingefügt werden. Diese werden über  $n$  Generationen aufbewahrt und dürfen in jedem Evolutions-Schritt am Selektionsprozess teilnehmen. Somit befinden sich stets  $m \cdot n$  Individuen im Pool. Die Praxis zeigt auch hier, dass ein solcher Pool die Konvergenzgeschwindigkeit erhöht, jedoch erwies sich die Implementierung von Kindergarten und Seniorenunterkunft als fehlerträchtig und wurde deshalb in der Endversion nicht eingebunden.



## Kapitel 6

# Grundlegender Aufbau des Programmcodes

Das Projekt unterliegt der GNU General Public License. Die Lizenz liegt im Verzeichnis "Programmcode" in der Datei "License.txt"

Diese Dokumentation unterliegt der GNU Free Documentation License, zu finden im Verzeichnis "Dokumentation" in der Datei "gfdl"

Das gesamte Projekt wurde in Pakete(Packages) aufgeteilt um die Übersichtlichkeit zu wahren.

Packages:

java: Nutzung JAVA-Eigener Funktionen, wie java.io.\*,ArrayList, Math...

GenAlg: Klassen des Genetischen Algorithmus, wie Kreuzung, Fitnessbewertung, Population, Individuum, Steuerungsklasse(GAWorldManager)

Turing: Komplette Turingmaschine

ToolPack: Mathematische Werkzeuge (Gray2Bin,...), String-Funktionen, Sonstiges(Array-Größenänderung)

### 6.1 Eingabe der Trainingsdaten

Die Datei "config.aic" enthält alle wichtigen Einstellungen zum Programm. Hier finden sich auch Eingabemöglichkeiten für Trainingsdatensätze. Da man ja gewillt ist, die Turingprogramme gleich bei ihrer Entwicklung zu verifizieren, wird nicht nur ein TrainingsInput/Output angegeben, sondern mehrere. Die Anzahl hängt einfach davon ab, wieviele Ihnen zur Verfügung stehen. Bei einfachen Rechenaufgaben sollte es kein Problem darstellen, mehrere Datensätze einzugeben. Da eine Turingmaschine aber auch in der Lage sein soll, komplexe Konvertierungen von Zeichenketten(Chiffrierung/Dechiffrierung) vorzunehmen, wird in einer späteren Version die Möglichkeit bestehen, reguläre Ausdrücke als Trainingsdaten anzugeben. AiccBot wird dann jede einzelne mögliche Zeichenkette überprüfen und bewerten. Die Fitness eines Programmes ergibt sich dadurch

aus dem Durchschnitt aller erzielten Ergebnisse, was die Richtigkeit eines Programms beweisen würde.

# Kapitel 7

## Literaturverzeichnis

[OOoMak] *OpenOffice.org-Makro zum Rucksack-Problem(GAKnapSack.swx)*

[Wiki1] *Wikipedia - Aufbau der DNA*  
<http://de.wikipedia.org/wiki/Desoxyribonukleins%C3%A4ure>

[StoStu] *Stoschek und Sturm - Molecular Computing*  
<http://lat.inf.tu-dresden.de/research/papers/2001/StoschekSturm+-IFE-01.pdf>

[HabilKokai] *Gabriella Kokai - Erfolge und Probleme evolutionärer Algorithmen, induktiver logischer Programmierung und ihrer Kombination*  
[http://fau20b.informatik.uni-erlangen.de/Forschung/Publikationen/download/Habil\\_Kokai.pdf?language=de](http://fau20b.informatik.uni-erlangen.de/Forschung/Publikationen/download/Habil_Kokai.pdf?language=de)