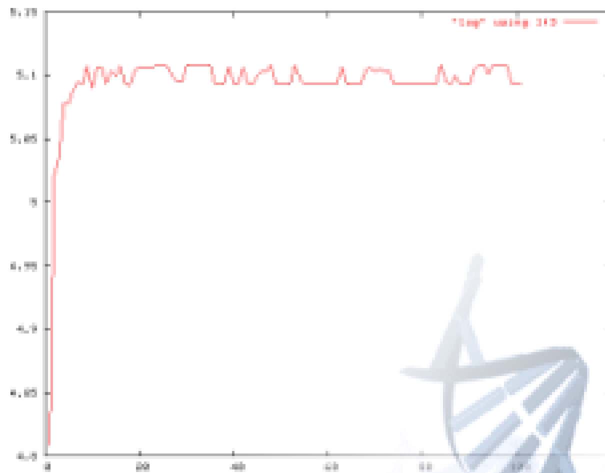
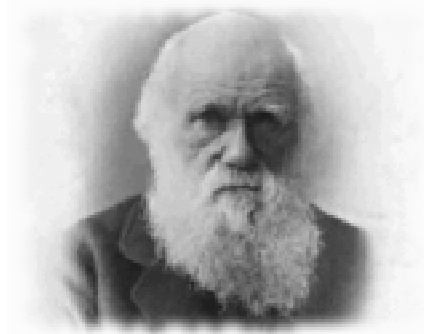
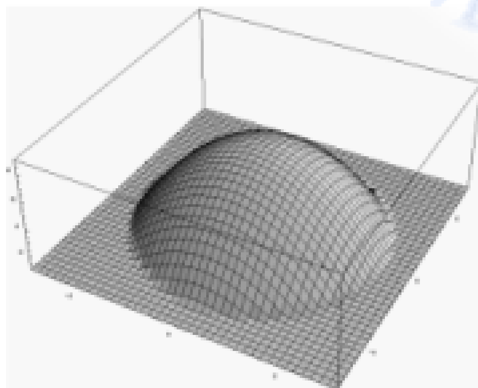
Evolutionäre Optimierung

Cédric Huwyler, Mathias Weyland

2. Fassung, 17. März 2004







Inhaltsverzeichnis

Inhaltsverzeichnis 2

1 Einleitung: Alles entwickelt sich 5

2 Implementierung einer Evolution 7

2.1 Biologische Grundlagen 7

2.1.1 Chromosomen, Gene und Allele 7

2.1.2 Geschlechtliche Vermehrung und Zellteilung 8

2.1.3 Selektion 9

2.2 Implementierung in einen Algorithmus 10

2.2.1 Ablaufschema eines Genetischen Algorithmus' 10

2.2.2 Codierung 11

2.2.3 Selektion 17

2.2.4 Geschlechtliche Vermehrung 21

2.2.5 Abbruchkriterium 24

2.3 Umsetzung des Algorithmus' in ein Programm 24

2.3.1 Struktur des Programms 25

2.3.2 Die Benutzerschnittstelle 26

3 Anwendungsbeispiel: Das Dreiecksproblem 28

3.1 Situation 28

3.1.1 Beschreibung 28

3.1.2 Berechnung des Umfangs 29

3.1.3 Berechnung der Dreiecksflächen 29
3.1.4 Aufstellen der Qualitätsfunktion 30
3.2 Auswertung 31
3.2.1 Gleichschenkliges Dreieck 31
3.2.2 Populationsgrösse variieren 32
3.2.3 Mutationsrate variieren 32
3.3 Fazit 36
3.3.1 Auflösung des Dreiecksproblemes 36
3.3.2 Vor- und Nachteile des Genetischen Algorithmus' 37

A Das Binärsystem 38

A.1 Darstellung von Binärzahlen 38

B Literaturverzeichnis 39

C Der Inhalt der CD 41

D Programmcode 42

D.1 ga.c 42
D.2 include/binary.h 45
D.3 include/debug.h 47
D.4 include/encoding.h 51
D.5 include/evaluation.h 54
D.6 include/genetic_operators.h 55
D.7 include/selection.h 57

E Analysetools 61

E.1 triangle.pl 61

E.2 wrap_gnuplot.pl 63

E.3 compute_triangle.c 65

F Die Autoren 66

G Danksagungen 66

1 Einleitung: Alles entwickelt sich

Alles begann damit, dass wir per Zufall im Internet auf ein Projekt stiessen, das ein für unseren Begriff sehr merkwürdiges, aber deshalb umso faszinierendes Ziel hatte: Es handelte sich dabei um ein Computerprogramm, das nur drei Dinge konnte:

- Sich fortpflanzen
- Ein anderes Programm *aufessen* (sprich löschen)
- Dem Benutzer berichten, was es tut

Die Entwickler dieses Projektes hatten mehrere dieser Computerprogramme in einer Testumgebung gegeneinander antreten lassen. Der Clou: Bei der Fortpflanzung wurde manchmal nicht eine identische Kopie erstellt, sondern es wurde eine kleine, zufällige Änderung in das Duplikat eingefügt. Durch diese Änderung entstanden nach und nach Programme, die entweder *besser* oder *schlechter* als das Original waren. Zum Beispiel besassen die besseren Nachkommen einen grösseren Aggressionstrieb, was sie im Kampf gegen andere effektiver machte.

Mit der Zeit verschwanden die schlechten Programme, oder anders gesagt, die guten Programme verdrängten die schlechten. Dieses Experiment hatte uns so sehr zugesagt, dass wir den Verlauf des Projektes beobachteten. Schon bald wurde eine erste Erkenntnis veröffentlicht, nämlich, dass die Programme von Generation zu Generation immer weniger und immer verstümmelter berichteten, was sie taten. Dies ist eigentlich verständlich, wenn man bedenkt, dass die Ausgabe von Informationen an den Benutzer für das Programm keinen Vorteil bringt, aber Zeit, welche für das Eliminieren von Artgenossen eingesetzt werden könnte, kostet. Dieses Ergebnis hat uns derart stark beeindruckt, dass wir uns entschlossen, unsere Maturaarbeit der evolutionären Simulation am Computer zu widmen.

Wir sind auch auf ein weiteres Projekt gestossen, in dem in einer Simulation Bakterien so gezüchtet werden, dass sie durch ihre Fortbewegungsart möglichst viel Nahrung finden. ([6])

Der Mythos dieser Computerprogramme, die sich selbst einem Evolutionsprozess unterwerfen und sich damit auf ein bestimmtes Ziel hin verbessern können, ist als *Genetische Programmierung* bekannt. Anfänglich beabsichtigten wir, unsere Maturaarbeit über dieses Thema zu schreiben. Uns wurde aber schnell klar, dass der Aufwand dafür ein sinnvolles Mass überschreiten würde. Darum beschlossen wir, uns mit der Grundlage der Genetischen Programmierung, den *Genetischen Algorithmen*, zu beschäftigen. Mit diesen ist es möglich, diverse mathematische, aber algebraisch unlösbare Probleme, die sich optimieren lassen, zu lösen.

So haben wir uns vorgenommen, eine Programmbibliothek mit Anwendungsbeispielen zu entwickeln, mit deren Hilfe solche Probleme gelöst werden können.

Wir haben unsere schriftliche Begleitarbeit so aufgebaut, dass dem Leser zunächst die wichtigen Vorgänge in der Natur erklärt werden, anschliessend eine Beschreibung der Implementierung dieser Vorgänge in einen Algorithmus folgt und schliesslich ein konkretes Anwendungsbeispiel behandelt wird. Im Anhang ist der gesamte Programmcode zu finden, auch sind dort einige Hilfsprogramme abgedruckt.

An verschiedenen Stellen haben wir Fussnoten eingebaut, die Themen vertiefen, die wir aus Platzgründen in unserer Arbeit nur anschneiden konnten.

Der schriftlichen Arbeit ist eine CD-Rom beigelegt, auf der sich der gesamte Programmcode und die Dokumentation befinden.

2 Implementierung einer Evolution

2.1 Biologische Grundlagen

Um evolutionäre Prozesse simulieren zu können, müssen wir zwingend die Grundlagen aus der Biologie, die uns als Vorbild dienen, verstehen. Unser Ziel ist es, ein Modell zu konstruieren, welches entscheidende Details dieses Vorbildes enthält. Wir möchten an dieser Stelle dem Leser jedoch nicht verschweigen, dass wir in den fogenden Abschnitten auch wichtige Einzelheiten ausgelassen haben, welche aber für das Verständnis unseres Modells nicht benötigt werden. Wer die genauen Unterschiede zwischen unserem Modell und den Vorgängen in der Natur nachvollziehen möchte, dem empfehlen wir [5].

2.1.1 Chromosomen, Gene und Allele

In jeder unserer Zellen ist der gesamte Bauplan unseres Körpers enthalten. Das Ziel der folgenden Abschnitte ist es, zu erklären, wie und in welcher Form diese enorme Zahl an Informationen gespeichert ist.

Chromosomen Alle in der Natur vorkommenden Lebewesen bestehen aus Zellen. In jeder dieser Zellen findet man unter anderem einen Zellkern, und innerhalb dieses Zellkernes befinden sich sogenannten Chromatinfäden. Auf diesen Fäden sind vier verschiedene Basen in unterschiedlicher Reihenfolge.

Es gibt im Zellkern jeweils zwei Chromatinfäden, die zusammenpassen. Wenn sich nun die Zelle teilt, dann werden diese beiden Chromatinfäden aufspiralisiert: es entsteht die bekannte Doppelhelix. Diese Doppelhelix spiralisiert sich ebenfalls auf und formt sich zu einem Chromosom:

Wir bezeichnen die Gesamtheit der Chromosomen in einem Individuum als Genom.

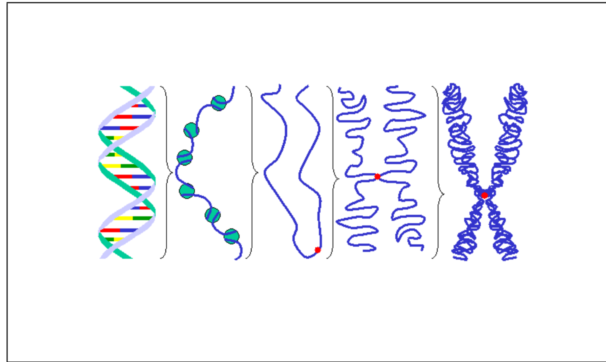


Abbildung 1: Aufspiralisierte Chromatinfäden, gesehen mit verschiedenen Vergrösserungen

Gene Die genaue Definition eines Genes ist ziemlich komplex und benötigt umfassendes Wissen, welches wir nicht voraussetzen[1]. Für uns ist folgendes entscheidend:

Wir haben weiter oben behauptet, jede menschliche Zelle enthielte den Bauplan des ganzen Menschen. Dieser Bauplan besteht aus vielen Genen, wobei jedes Gen eine Erbinformation trägt. Zum Beispiel gibt es bei den Erbsen ein Gen, welches die Erbsenfarbe bestimmt. Es kann aber durchaus vorkommen, dass ein Merkmal (beispielsweise die Augenfarbe eines Menschen) von mehr als einem Gen abhängt.

Allele Stellt man sich ein Gen als einen Schalter vor, der auf verschiedenen verschiedenen Positionen stehen kann, so ist ein Allel genau so eine Schalterposition.

Beim obigen Beispiel mit den Erbsen gäbe es zum Beispiel die Allele Gelb und Grün. Ein Allel kann aber durchaus mehr als nur zwei „Positionen“haben. (Ein Beispiel hierfür wären die Blutgruppen; dort gibt es Allele für A, B und 0.)

2.1.2 Geschlechtliche Vermehrung und Zellteilung

Es würde den Rahmen dieser Arbeit sprengen, wenn wir an dieser Stelle Meiose und Mitose erklären würden. Wir beschränken uns deshalb auf den Verweise auf eine gute Quelle wie das bereits erwähnte Buch ([5]).

[1] http://de.wikipedia.org/wiki/Gen

Rekombination Beim Verschmelzen von Spermium und Eizelle verschmilzt auch der Bauplan des Vaters (übertragen durch das Spermium) und der der Mutter (in der Eizelle enthalten). Bei diesem Vorgang wird das Erbgut **zufällig** neu kombiniert.

Die Rekombination ist eine Ursache für den Erfolg der Evolution, da auf diese Art und Weise sehr schnell neue Eigenschaften entstehen.

Mutation Die Mutation ist eine Veränderung der Informationen auf dem Chromatinfaden. Diese Veränderung geschieht zufällig und kann sich sowohl positiv als auch negativ auf das werdende Individuum auswirken.

Mutationen werden vorwiegend durch Strahlung hervorgerufen und sind deshalb ein natürlicher Vorgang, da wir auf der Erde Strahlung ausgesetzt sind.

Crossover Bei einer geschlechtlichen Zellteilung kann es vorkommen, dass zwei Chromatinfäden vertauscht und wieder eingebaut werden. Dieser Vorgang wird Crossover genannt.

2.1.3 Selektion

In der Mitte des 19. Jh. veröffentlichte *Charles Darwin*[2] seine Theorie der natürlichen Selektion.

Er hatte bemerkt, dass in der Natur zwei Faktoren von grosser Bedeutung sind:

- Begrenzte Ressourcen
- Konkurrenz

Darwin folgerte in seinen Thesen daraus: *„Es kann nur existieren, was dieser Konkurrenz gewachsen ist, da das Unterlegene die Grundlagen, die es braucht, nicht finden kann.“*[3]

[2] http://de.wikipedia.org/wiki/Charles_Darwin
[3] http://de.wikipedia.org/wiki/Darwinismus

Wir sehen in der Natur, dass sich die Lebewesen ständig mit immer neuen Tricks zu übertrumpfen versuchen. Zum Beispiel haben im Verlauf der Jahrhunderte die Tiere in kalten Regionen ihren Wärmehaushalt ständig optimiert, indem sie immer kleiner und runder wurden (siehe Bergmannsche Regel[4]).

Die Fitness (Grad der Überlebensfähigkeit) des Individuums äussert sich in der Anzahl seiner Nachkommen. Je einfacher es überlebt, desto mehr Nachkommen produziert es. Produziert es mehr Nachkommen, so ist sein Erbgut in der nächsten Generation weiter verbreitet als vorher: das Individuum wurde selektiert.

2.2 Implementierung in einen Algorithmus

Es stellt sich nun die Frage, ob sich diese Abläufe in ein relativ einfaches Kochrezept, einen Algorithmus, umsetzen lassen. Dazu müssen wir die verschiedenen Strukturen und Prozesse des Lebens in mathematische Werkzeuge umwandeln. Dieser Algorithmus bildet keineswegs die Natur ab, aber er benutzt einige ihrer Werkzeuge um sein Ziel zu erreichen. Wir werden uns im Folgenden den groben Ablauf eines Genetischen Algorithmus' anschauen und anschliessend genauer auf die einzelnen Teilprozesse eingehen.

2.2.1 Ablaufschema eines Genetischen Algorithmus'

Grob mit Pseudocode umrissen sieht ein Genetischer Algorithmus folgendermassen aus:

- Initialisiere Population aus bekannten Suchintervallen (Codierung).
- While(Abbruchkriterium nicht erfüllt)
  - Selektiere die fittesten Individuen aus der Population und wirf sie in einen Genpool (Selektion).
  - Paare die Individuen aus dem Genpool mit Hilfe von Mutation und Crossover (Reproduktion).
- Gib die Parameter des Individuums mit der besten Fitness zurück.

[4] http://de.wikipedia.org/wiki/Bergmannsche_Regel

2.2.2 Codierung

Die Codierung ist das Fundament eines Genetischen Algorithmus'; sie bestimmt, wie die Erbinformationen gespeichert und abgerufen werden. Wir brauchen also eine mathematische Struktur, in die wir die Werte einzelner Gene einbeschreiben können.

Im Bereich der evolutionären Optimierung unterscheidet man grundsätzlich zwei Vorgehensweisen: die Evolutions-Strategie und der Genetische Algorithmus. Beide funktionieren nach dem obigen Ablaufschema und unterscheiden sich hauptsächlich in ihrer Codierung: *die Art und Weise des Speicherns und Abrufens von Genen*. Wir wollen im Folgenden kurz diese strukturellen Unterschiede erläutern:

In der **Evolutions-Strategie** werden die einzelnen Gene als reelle Zahlen in einzelnen Variablen gespeichert. Bei der Paarung (Kombination) zweier Individuen werden die neuen Gene der Nachkommen durch die Bildung des Mittelwerts der Gene der Eltern gebildet. Die Mutation der Zahlen erfolgt durch ihre Veränderung um einen bestimmten Wert, dessen Grösse im Bereich einer vorher definierten Normalverteilung liegt.

Im **Genetischen Algorithmus** werden die Zahlenwerte der einzelnen Parameter als Binärzahlen in sogenannten *Bitstrings* aneinandergereiht. Der Bitstring steht für ein Chromosom mit all seinen Genen, die nur durch im Algorithmus definierte Grenzen unterscheidbar sind; gegen aussen erscheint das Chromosom als eine einzige Binärzahl. Kombiniert (gepaart) wird durch Crossover, das verschiedene Segmente der zwei Elter-Bitstrings miteinander vertauscht. Die Mutation erfolgt durch die Invertierung eines Bits des Chromosoms.

Der hauptsächliche Unterschied zwischen den beiden Algorithmen liegt in der Codierung: reelle Zahlen oder Bitstrings. Obwohl die Umsetzung eines Genetischen Algorithmus' um einiges schwieriger als die einer Evolutions-Strategie ist, haben wir uns im Hinblick auf eine spätere Beschäftigung mit der *Genetischen Programmierung* (siehe Einleitung) für den **Genetischen Algorithmus** entschieden.

Die Codierung beinhaltet zwei Funktionen: *Encodierung* und *Decodierung* von Bitstrings.

Encodierung Die Encodierung digitalisiert die Parameter der Optimumssuche zu Chromosomen, mit denen der Genetische Algorithmus anschliessend arbeiten kann.

Um eine Population von Chromosomen erstellen zu können, müssen wir die Längen der einzelnen Gene in Bits berechnen und diese zur Gesamtlänge des Chromosoms aufsummieren. Zur Bestimmung der Länge eines Genes müssen wir dessen Ausmasse, also das Intervall in dem es sich bewegt, kennen. Dieses Intervall wird durch eine untere und eine obere Grenze repräsentiert. Die ungefähre Lage des Optimums, also seine obere und seine untere Grenze, ist oft bekannt. So weiss man zum Beispiel sicher, dass sich das Optimum einer quadratischen Funktion (Scheitel der Parabel) irgendwo zwischen ihren zwei Nullstellen verbirgt (falls sie welche hat). Unsere Optimumssuche müsste also im Intervall zwischen diesen beiden Nullstellen stattfinden. Falls man dieses 'Suchintervall' eines Genes nicht genau eingrenzen kann, ist es ratsam, das Intervall eher etwas grösser zu wählen, da sonst Optima verlorengehen könnten.

Die Tatsache, dass die Parameterwerte der Funktion, deren Optimum gesucht ist, als Binärzahlen abgespeichert werden, stellt uns vor zwei Probleme:

Wir können nur positive, ganze Zahlen im Binärformat abspeichern. Der Computer kann zwar mit vorzeichenbehafteten Fliesskommazahlen umgehen, benutzt dazu aber kleine Tricks, die für diese Arbeit nicht weiter relevant sind.

Bei vorzeichenbehafteten Zahlen enthielte das vorderste Bit den Wert des Vorzeichens, also '1' für '+' und '0' für '-'. Das kann bei den weiter unten beschriebenen Kombinationsmethoden schnell unerwünschte Folgen in Form einer starken Änderung einer grossen Zahl bewirken.
Um diesen Effekt zu verhindern, verschieben wir das Intervall, in dem das Optimum erwartet wird, so nach rechts oder links, dass es gerade bei 0 beginnt. Später, wenn wir die gespeicherte Zahl wieder herauslesen wollen, können wir dieses Intervall durch Addieren der unteren Intervallgrenze, einem sogenannten *Offset* (Verschiebungsfaktor), in seinen Ursprungszustand verschieben.

Wir machen alle Zahlen im Intervall ganzzahlig, indem wir das Intervall mit der richtigen

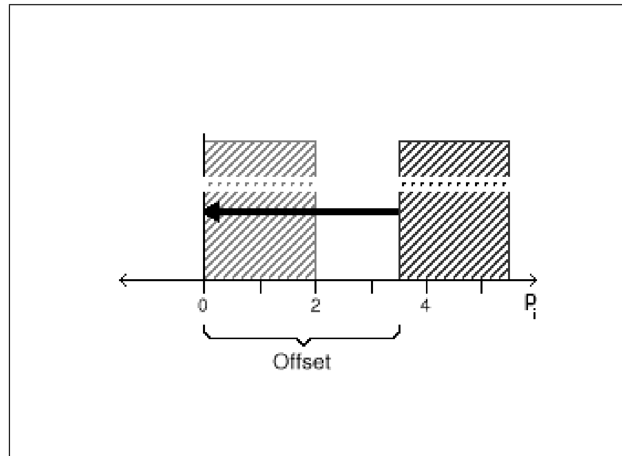


Abbildung 2: Jedes beliebige Intervall wird mit einem Offset so verschoben, dass es bei 0 beginnt.

Zehnerpotenz multiplizieren. Wünschen wir ein Resultat mit einer Genauigkeit von 3 Stellen, so multiplizieren wir es mit 10^3. Greifen wir irgendeine Zahl aus diesem Intervall heraus und möchten sie in ihre ursprüngliche Form zurückverwandeln, so dividieren wir sie einfach wieder durch 10^3 und erhalten eine Fliesskommazahl mit 3 Stellen hinter dem Komma.

Kennen wir einmal das Intervall, so können wir die Anzahl der für das Gen benötigten Bits berechnen:

Die Grösse unseres Intervalls lässt sich wie folgt ausdrücken:

intv	:=	Obere Grenze des positiven, ganzzahligen Intervalls. (Die untere Grenze ist 0)
a	:=	untere Grenze des Intervalls in dem das Optimum erwartet wird
b	:=	obere Grenze des Intervalls in dem das Optimum erwartet wird
prec	:=	Anzahl Stellen hinter dem Komma

$$intv = (b - a) \cdot 10^{prec}$$

Die Zahl mit der gesuchten Anzahl Stellen sollte gleich gross oder grösser als das Intervall

sein, damit dieses auf jeden Fall in die Binärzahl passt. Da im Binärcode (siehe Anhang) die n-te Stelle den Wert 2^n besitzt, passt eine Zahl, die kleiner als 2^n ist, sicher in diese Binärzahl. Wir runden n also auf den nächstgrösseren ganzzahligen Wert auf, falls n Dezimalstellen enthält.

Um die Anzahl Binärstellen n zu bestimmen, lösen wir folgende Gleichung:

$$\begin{aligned} intv &= \lceil 2^n \rceil \\ \ln intv &= \lceil n \cdot \ln 2 \rceil \\ n &= \left\lceil \frac{\ln(intv)}{\ln 2} \right\rceil \end{aligned}$$

Auf diese Art und Weise können wir für jedes Gen die benötigten Bits berechnen und zur Länge des gesamten Chromosoms aufaddieren:

Länge eines Chromosoms:

l	:=	Länge des Chromosoms in Bits
a_i	:=	untere Grenze des Intervalls des Parameters i
b_i	:=	obere Grenze des Intervalls des Parameters i
q	:=	Anzahl Gene

$$l = \sum_{i=1}^{q} \frac{\ln\left((b_i - a_i) \cdot 10^{prec_i}\right)}{\ln 2}$$

Ein Byte sind acht Bits; wir rechnen also um und runden auf, da wir mit dem Computer nur ganze Bytes verwalten können.

Anzahl der benötigten Bytes

$$bytes = \left\lceil \frac{l}{8} \right\rceil$$

Zum besseren Verständnis erklären wir diesen Vorgang an einem kleinem Beispiel:

Das Optimum der Funktion $f(x, y) = -x^2 - y^2 + 1$ wird gesucht; die erwünschte Genauigkeit beträgt 2 Stellen hinter dem Komma.

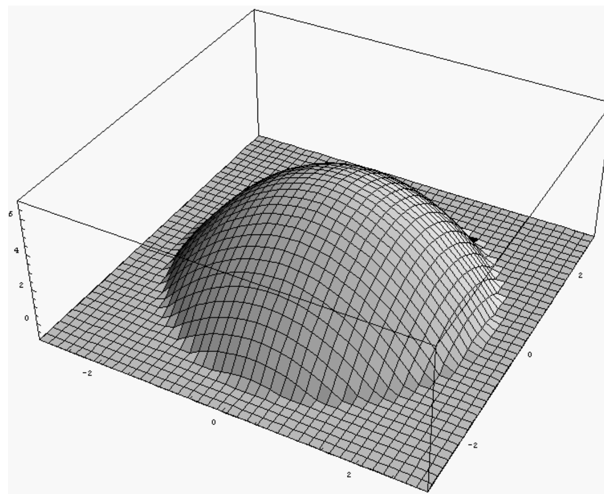


Abbildung 3: Beispielfunktion: $f(x, y) = -x^2 - y^2 + 1$

Die ganzzahligen Nullstellen dieser Funktion liegen, wie aus ihrem Graph ersichtlich, bei $x = \{-1, 1\}$ und $y = \{-1, 1\}$. Wir können also annehmen, dass sich das gesuchte Optimum irgendwo im Intervall $x = [-1; 1]$ und $y = [-1; 1]$ liegt.

Die oberen Grenze der positiven, ganzzahligen Intervallen von x und y betragen also

$$intv_x = (1 - (-1)) \cdot 10^2 = 200$$

und

$$intv_y = (1 - (-1)) \cdot 10^2 = 200$$

Demzufolge ist die Anzahl n der benötigten Binärstellen für x und y

$n_x = \lceil \frac{\ln 200}{\ln 2} \rceil = 8$ Bits und $n_y = \lceil \frac{\ln 200}{\ln 2} \rceil = 8$ Bits

Daraus berechnen wir die Länge des Chromosoms:

$l = n_x + n_y = 8 + 8 = 16$ Bits

Ein Chromosom benötigt also $\lceil \frac{16}{8} \rceil = 2$ Bytes.

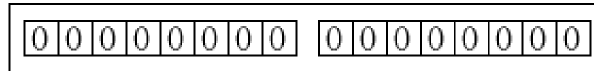


Abbildung 4: Ein Chromosom bestehend aus einem 2 Byte langen Bitstring.

Decodierung Um die einmal gespeicherten und veränderten Gene später wieder aus dem Chromosom herauszulesen (Decodierung), müssen wir die Binärzahlen (Siehe Anhang) wieder in Dezimalzahlen umwandeln können. Dies gestaltet sich sehr viel einfacher:

Berechnung des Dezimalwertes des Binärzahl

g_i	:=	Dezimalwert des i-ten Gens
u	:=	Startposition des Gens im Chromosom
v	:=	Endposition des Gens im Chromosom

$$g = \sum_{i=u}^{v} bit_i \cdot 2^{bit_i}$$

Um unsere Veränderungen bezüglich Intervallgrösse und -position (siehe oben) wieder rückgängig zu machen, dividieren wir die reelle Zahl g_i durch die Zehnerpotenz, die wir in der Encodierung mit dem Intervall multipliziert haben und addieren das Offset, also die Verschiebung von 0.

Rückverwandlung in das Originalintervall:

$prec_i$	:=	Genauigkeit des in Gen i gespeicherten Parameters
a	:=	untere Grenze des in Gen i gespeicherten Parameters
p_i	:=	angepasster Dezimalwert des in Gen i gespeicherten Parameters

$p_i = \frac{g_i}{10^{prec_i}} + a_i$

Benutzen wir auch hier das obige Beispiel, um diesen Vorgang zu erläutern:

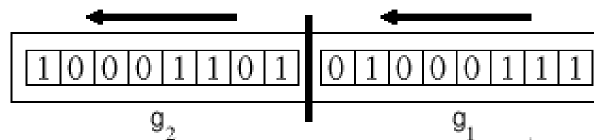


Abbildung 5: Im Chromosomen werden die einzelnen, durch Grenzen abgesteckten Gene von rechts nach links eingelesen. Diese Grenze kann natürlich auch irgendwo in einem Byte liegen.

Wir haben der Population das oben abgebildete Chromosom entnommen. Wie lauten die zwei in ihm gespeicherten Parameter ?

Zuerst lesen wir die zwei gespeicherten Gene aus:

$g_1 = 2^0 + 2^1 + 2^2 + 2^6 = 71$ und $g_2 = 2^0 + 2^2 + 2^3 + 2^7 = 205$

Anschliessend verwandeln wir diese Genwerte in die gesuchten Parameterwerte:

$p_1 = \frac{g_1}{prec_1} + a_1 = \frac{71}{10^2} - 1 = -0.29$ und $p_2 = \frac{g_2}{prec_2} + a_2 = \frac{205}{10^2} - 1 = 0.41$

Ausgehend von der Codierung können wir uns nun den auf dieser aufbauenden Prozessen widmen: *Selektion* und *geschlechtliche Vermehrung*

2.2.3 Selektion

Die Selektion sorgt dafür, dass sich in einer Population das stärkste Individuum durchsetzt, weil es am meisten Nachkommen produzieren kann. Das Mass für die Stärke ist die

Fitness, die mit dem Grad der Anpassung eines Individuums an seine Umwelt zunimmt.

Wir brauchen also Funktionen, die die Fitness der Chromosomen berechnen und die Anzahl ihrer Nachkommen bestimmen können.

Die Fitnessfunktion Die Fitnessfunktion setzt sich aus der sogenannten *Ziel-* oder *Qualitätsfunktion* zusammen. Die Qualitätsfunktion berechnet die Qualität (Nähe zum Optimum) eines Chromosoms; sie stellt quasi das Umfeld des Individuums dar und gibt die Bedingungen vor, die eine Verbreitung des eigenen Erbgutes erlauben. Auf der Erde besteht diese zum Beispiel aus den klimatischen Verhältnissen, dem Futterangebot oder den natürlichen Feinden.

Je besser ein Individuum diese Bedingungen erfüllt, desto höher sind seine Überlebenschancen, desto grösser ist die Anzahl seiner Nachkommen und desto grösser ist sein Durchsetzungsvermögen.

Die Fitnessfunktion muss also eine Funktion sein, die einen kleinen Wert für eine schlechte Qualität und einen grossen Wert für eine gute Qualität zurückgibt. Die Qualitätsfunktion muss also mit Hilfe einer *Anpassungsfunktion* so in die Fitnessfunktion eingebunden werden, dass aus einer grossen Qualität eine grosse und aus einer kleinen Qualität eine kleine Fitness resultiert.

Bei Minimierungsproblemen kann die Anpassungsfunktion zum Beispiel das Reziproke der Qualitätsfunktion zurückgeben; somit wird die Fitness grösser wenn der Rückgabewert der Qualitätsfunktion kleiner wird.

Die Fitness eines Individuums wird wie folgt berechnet:

QF()	:=	Qualitätsfunktion der Parameter des Optimierungsproblems.
AF()	:=	Anpassungsfunktion
p_i	:=	einzelner Parameter des Optimierungsproblems.
P_i	:=	Menge der Parameter des Optimierungsproblems.

$$P_i = \{p_1, p_2, p_3, ..., p_n\}$$
$$fitness_i = AF(QF(P_i))$$

Der Selektionsalgorithmus

Der Selektionsalgorithmus ordnet einem Individuum einer Population die Anzahl seiner Nachkommen zu.

Dafür gibt es verschiedene bekannte Methoden; wir haben uns die *Roulette Wheel Selection* ausgewählt und ein wenig modifiziert.

Die Roulette Wheel Selection belegt für jedes Individuum einen Sektor eines Roulette-Rades. Die Grösse dieses Sektors ist proportional zur Fitness des jeweiligen Individuums. Die Kreislinie wird anschliessend, analog zum Hineinwerfen der Kugel in das Roulette-Rad und Beobachten, wo sie liegen bleibt, mit zufälligen Winkeln im Intervall [0°;360°[bombardiert. Der jeweilige Besitzer des getroffenen Teils der Kreislinie darf sein Erbgut in einen für die spätere Paarung vorgesehenen, sogenannten Genpool geben. Dieser Prozess wird so lange wiederholt, bis die Anzahl ausgewählter Chromosomen die Grösse der Population erreicht.

Das Vorteilhafte an dieser Methode ist die Möglichkeit, dass sich auch ein mit eher schlechten Genen ausgestattetes Individuum fortpflanzen kann (wenn auch nur mit einer

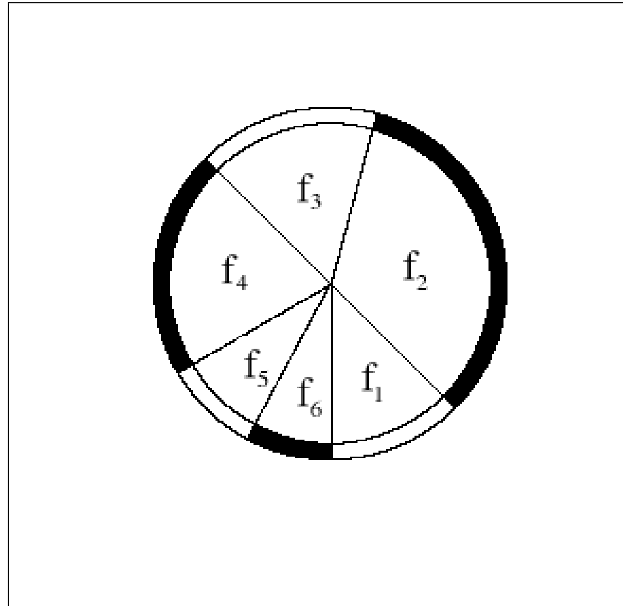


Abbildung 6: In der Roulette Wheel Selection bekommt jedes Individuum einen zu seiner Fitness passenden Sektor des Roulette-Rades zugeordnet.

geringen Wahrscheinlichkeit), was für eine gewisse genetische Vielfalt sorgt und eine Monokultur, die zu einer schnellen Konvergenz in ein lokales Optimum führt, verhindert.

Nun haben wir uns gedacht, dass die Idee eines Roulette-Rades zwar schön vorstellbar, aber eigentlich viel zu ineffizient und kompliziert ist. So nehmen wir für den Kreis das Intervall [0;1[und für die Kreissektoren gewisse Bereiche in diesem Intervall.

Für die Kennzeichnung der einzelnen Bereiche berechnen wir die *oberen* Intervallgrenzen der jeweiligen Individualfitness. Dies tun wir mit einer Summenformel.

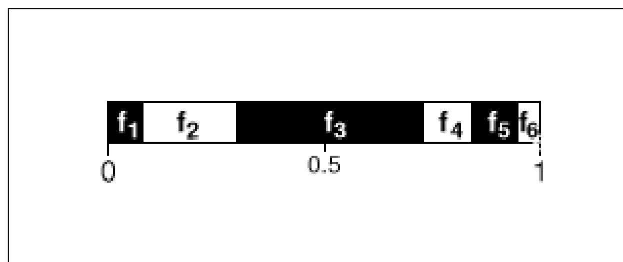


Abbildung 7: Um sinnlos komplizierte Berechnungen am Kreis zu vermeiden, haben wir das Roulette-Rad auf das Intervall [0;1[umgemünzt.

Für die Kennzeichnung der einzelnen Bereiche berechnen wir die *oberen* Intervallgrenzen der jeweiligen Individualfitness. Dies tun wir mit einer Summenformel.

k	:=	Individuum
f_k	:=	obere Grenze des Intervallsektors des Individuums k
f_{tot}	:=	Summe aller Fitnesswerte der Population
n	:=	Populationsgrösse

$f_k = \sum_{i=1}^{k} \frac{fitness_i}{f_{tot}}$ wobei $f_{tot} = \sum_{i=1}^{n} \frac{fitness_i}{n}$

Wir relativieren die Fitnesswerte an das Intervall [0;1[, indem wir sie jeweils durch die totale Fitness dividieren.

Es werden nun solange Zufallswerte im Intervall [0;1[erzeugt, mittels der vorher berechneten oberen Grenzen den Individuen zugeordnet und deren Chromosomen in den Genpool gelegt, bis dessen Grösse die Grösse der Population erreicht hat. In diesem Genpool werden die fitten Chromosomen häufiger vorkommen als die weniger fitten: das Ziel der Selektion ist erreicht.

2.2.4 Geschlechtliche Vermehrung

Die geschlechtliche Fortpflanzung ist der verändernde Faktor im Genetischen Algorithmus. Die Chromosomen werden durch sie gepaart, nach bestimmten Regeln in ihrem Betrag geändert und führen so zu neuen, vielleicht überlebensfähigeren Individuen. Sie setzt sich aus **Mutation** und **Crossover** zusammen.

Paarung Um eine geschlechtliche Vermehrung herbeizuführen, müssen wir die Individuen der Population zufällig untereinander paaren. Da wir die im Genpool liegenden Chromosomen bereits zufällig ausgewählt haben, können wir diesen direkt in Paare aufteilen. Danach können wir alle diese Paare mittels Mutation und Crossover kombinieren.

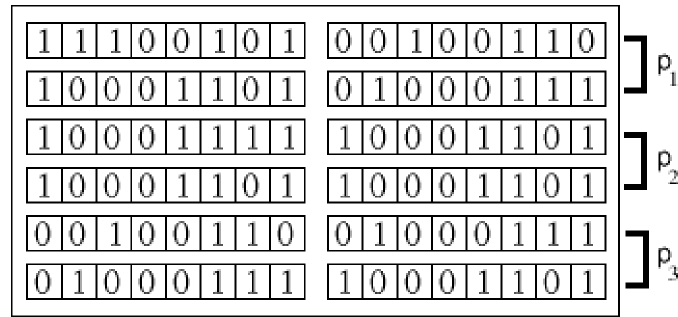


Abbildung 8: Die Population wird in Zweierpaare aufgeteilt.

Mutation Der Sinn der Mutation ist eine sprunghafte Veränderung des Chromosoms, die sich positiv (näher am Optimum) oder negativ (weiter vom Optimum entfernt) auswirken kann.

Der Mutationsvorgang ist sehr simpel: Die Bits im Chromosom werden mit einer bestimmten Wahrscheinlichkeit (in unserem Programm zum Beispiel 20%) invertiert, was zu einer Veränderung des Gens, das dieses Bit beinhaltet, führt. Diese Mutation kann je nach Stelle einen verschieden grossen Einfluss auf die Veränderung des Gen-Wertes ausüben. Der Wert des Gens verändert sich um 2^n, wobei n für die Stelle in der Binärzahl steht. Findet die Mutation also weiter links im Binärcode des Genes statt, wird das Gen viel stärker verändert, als wenn die Mutation weiter rechts stattfindet.

Schauen wir uns doch ein Beispiel an:

Ein Gen eines Chromosoms hat den Wert $220_{10} = 11011100_2$. Das dritte und das siebte Bit werden 'Opfer' einer Mutation: der neue Wert ist nun $40_{10} = 01011000_2$. Der Wert des Gens ist von 220 auf 40 gesunken, also um 180 kleiner geworden.

Dieser Effekt kann sowohl als behindernd, als auch als hilfreich interpretiert werden: Behindernd ist er insofern, indem ein gutes Gen auf einen Schlag viel zu stark verändert werden kann und so infolge mangelnder Fitness schlagartig aus der Population verschwinden kann.

Hilfreich ist er hingegen, wenn es um die Aufrechterhaltung der genetischen Vielfalt geht. Ist die genetische Vielfalt klein, so konvergieren die Gene viel zu schnell gegen ein

willkürlich gewähltes Optimum; neue, bessere Gene entstehen kaum. Ist sie jedoch gross, so hat der Genetische Algorithmus immer wieder Chancen, aus einem lokalen Optimum herauszukommen.

Dieser Effekt kann, muss aber nicht, unterbunden werden. Man kann entweder den Gray-Code statt dem Binärcode benutzen, oder die Mutationswahrscheinlichkeit abhängig von der Bit-Position im Gen machen. Wir verwenden jedoch keine dieser Methoden, wollen deshalb auch nicht weiter darauf eingehen und verweisen auf die Seiten 12-15 in [3].

Crossover Das Crossover sorgt, wie in der Genetik, für einen *Informationsaustausch* zwischen zwei gepaarten Individuen. Durch diesen Informationsaustausch besteht die Chance, neue Individuen mit den kombinierten positiven Eigenschaften der Eltern zu erhalten. Auf den Genetischen Algorithmus umgemünzt, muss das Crossover beliebige oder bestimmte, man spricht dann von uniformem Crossover, Bit-Sequenzen zweier Chromosomen vertauschen und kreiert so aus zwei Elter-Chromosomen zwei Nachkommens-Chromosomen mit verändertem Erbgut.

Hier weichen wir übrigens bereits stark von der Natur ab: Wir benutzen das Crossover, das in der Natur nur zufällig auftritt, als *Rekombinationsmethode* zur Erstellung der von den Eltern verschiedenen Nachkommen. Da wir nur ein Chromosom pro Individuum verwenden, können wir natürlich keine Rekombination der verschiedenen väterlichen und mütterlichen Chromosomen benutzen.

Schauen wir uns doch ein Zahlenbeispiel an:

Wir vertauschen die Bit-Sequenz der beiden Elter-Chromosomen mit den Werten

$$1110010100100110_2 \quad \text{und} \quad 1000110101000111_2$$

beginnend bei Bit 6 und endend bei Bit 11.

Wir erhalten Nachkommens-Chromosomen mit den Werten 1110010101000110_2 und 1000110100100111_2

Es ist schwer zu erklären, warum ein Genetischer Algorithmus mittels Crossover zur Konvergenz führt und würde den Rahmen dieser Maturaarbeit sprengen. Der interessierte Leser sei jedoch auf Kapitel 4 in [4] verwiesen.

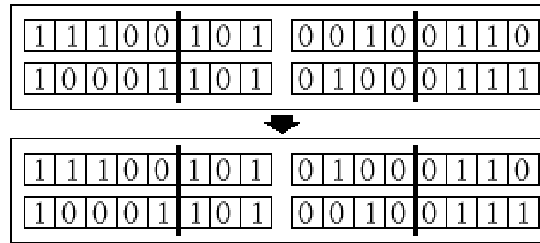


Abbildung 9: Crossover - Die Vertauschung der Bits zweier Chromosomen zwischen zwei Grenzen.

Wir besitzen jetzt also *Mutation* für Sprünge im Suchintervall und *Crossover* für eine Konvergenz zum Optimum als genetische Operatoren, die wir auf Chromosomen anwenden können.

2.2.5 Abbruchkriterium

Das Abbruchkriterium gibt an, wann das Resultat ausreichend ist und der Genetische Algorithmus beendet werden kann. Dafür gibt es folgende Fälle:

1. Eine gewisse Anzahl von Generationen g_{max} wurde überschritten.
2. Die beste Individualfitness hat sich über eine definierte Generationsspanne durchsetzen können (Konvergenz).
3. Die beste Individualfitness übersteigt einen definierten Wert.

Es ist nicht sehr einfach, diese Kriterien so zu bestimmen, dass sie optimal zum gestellten Problem sind. Es ist daher empfehlenswert, den Algorithmus einige Male mit verschiedenen Abbruchkriterien durchzuprobieren.

2.3 Umsetzung des Algorithmus' in ein Programm

Die Umsetzung des Genetischen Algorithmus' in ein Programm ist in den Kommentaren im beiliegenden Programmcode ausführlich erklärt. Wir möchten aber dennoch kurz auf

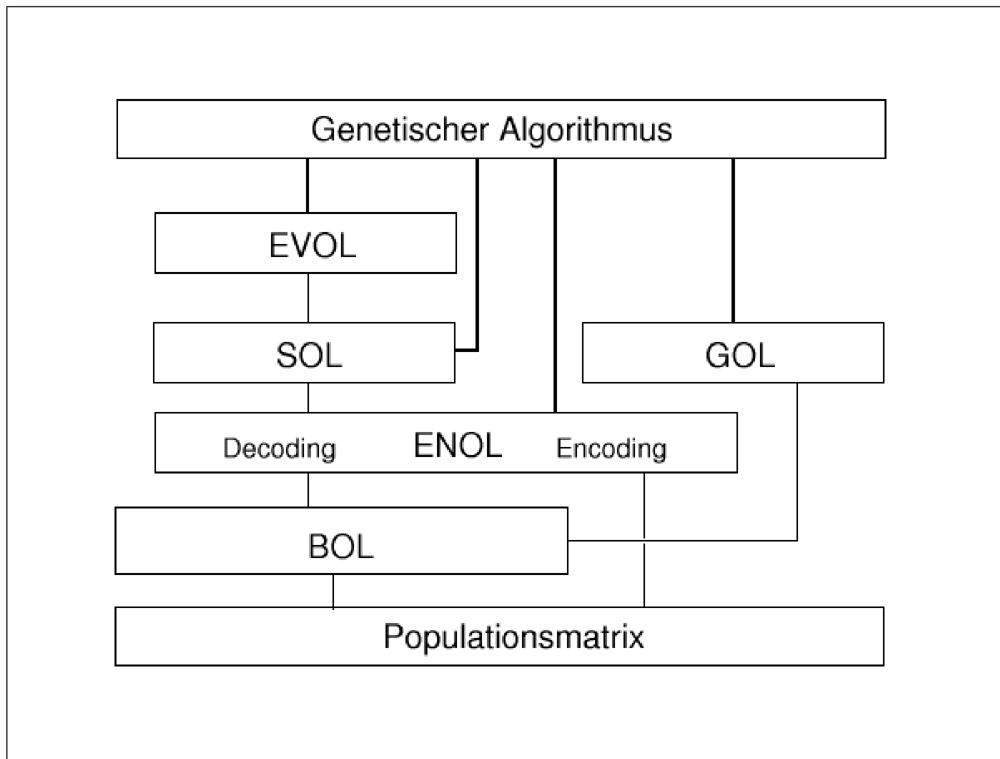


Abbildung 10: Die verschiedenen Layer des Programmcodes

zwei Punkte eingehen:

2.3.1 Struktur des Programms

Wir haben das Programm in fünf sogenannte *Layer* (Ebenen) aufgeteilt, die miteinander kommunizieren können:

Das Binary Operation Layer (BOL) Hier finden alle Operationen auf binärer Ebene statt. Das Binary Operation Layer kann Bits in der Population lesen, schreiben, zufällig initialisieren und vertauschen.

Das Encoding Operation Layer (ENOL) Das Encoding Operation Layer ist eine Implementation der Codierung (Abschnitt 2.2.2). Es kann aus den bekannten Suchintervall- und Genauigkeitsdaten die Grösse der Population bestimmen und im Arbeitsspeicher des

Computers einen Speicherbereich für diese reservieren.

Das Selection Operation Layer (SOL) Das SOL beinhaltet die Funktionen zur Berechnung der Fitness eines Individuums und zur Berechnung des Fitness-Intervalls der ganzen Population.

Das Genetic Operation Layer (GOL) Dieses Layer enthält die Funktionen zur geschlechtlichen Vermehrung von zwei Individuen: Mutation und Crossover.

Das Evaluation Operation Layer (EVOL) Das Evaluation Operation Layer prüft die Population auf die Erfüllung eines der in 2.2.5 beschriebenen Abbruchkriterien und bricht den Algorithmus allenfalls ab.

2.3.2 Die Benutzerschnittstelle

Bei der Erstellung des Programms wurde viel Wert auf ein dynamisches Verhalten gelegt; der Benutzer kann das Programm mit wenigen Parametern bedienen.

Parameter des Problems

- benutzerdefinierte Qualitätsfunktion
- Festlegung der Suchintervalle und der Genauigkeiten

Parameter des Genetischen Algorithmus'

- Populationsgrösse
- Mutationsrate
- Abbruchkriterien
  - Maximale Anzahl von Generationen

- Anzahl Generationen mit gleicher, bester Fitness
- Fitnessschwelle

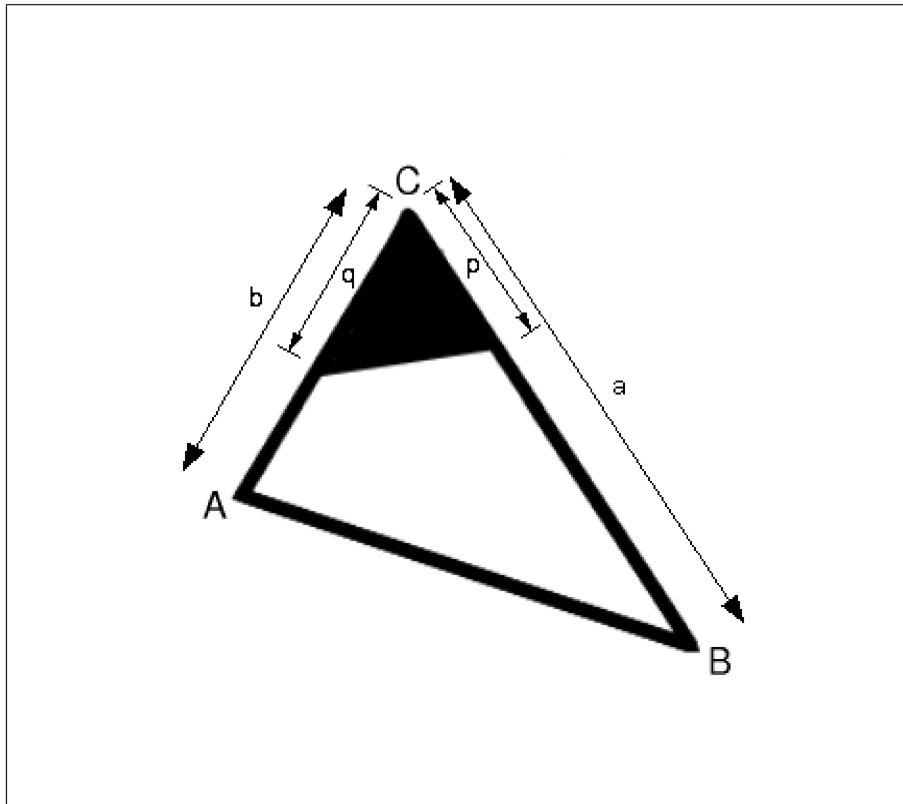


Abbildung 11: Das Dreiecksproblem

3 Anwendungsbeispiel: Das Dreiecksproblem

Jetzt, wo wir unseren Algorithmus in ein Programm umgesetzt haben, brauchen wir auch ein Optimierungsproblem, um ihn auf seine Funktion und Effizienz zu testen.

3.1 Situation

3.1.1 Beschreibung

Eine Sehne soll so in ein Dreieck gelegt werden, dass sie Umfang und Fläche halbiert. Dass dies bei einem gleichschenkligen Dreieck funktioniert, ist klar, da diese Sehne dann genau die Höhenlinie der Grundseite ist. Wie sieht es aber bei einem beliebigen Dreieck aus, dessen Seiten a, b und c so gewählt sind, dass weder ein gleichschenkliges noch sonst ein besonderes Dreieck entsteht ? Wir benutzen den Genetischen Algorithmus, um

zu zeigen, ob diese umfang- und flächenhalbierende Sehne bei einem beliebigen Dreieck überhaupt existiert oder ob wir uns mit einer kleinen Abweichung begnügen müssen.

Wir stellen nun Formeln für die Berechnung von Umfang und Fläche des gesamten Dreiecks und des durch die Sehne begrenzten, oberen Dreiecks auf (siehe Abbildung 11), um anschliessend die Qualitätsfunktion zur Beurteilung der Sehne erstellen zu können.

3.1.2 Berechnung des Umfangs

Der Umfang des gesamtes Dreiecks beträgt natürlich

$$U = a + b + c$$

Der Anteil des Teildreiecks am gesamten Umfang besteht aus der Summe der Streckenlängen von p und q:

$$s_{pq} = p + q$$

3.1.3 Berechnung der Dreiecksflächen

Die Gesamtfläche des Dreiecks beträgt

$$A = \frac{g \cdot h}{2} = \frac{a \cdot h_a}{2} = \frac{a \cdot b \cdot \sin\gamma}{2}$$

und die durch die Sehne begrenzte, obere Teilfläche

$$A_{pq} = \frac{g \cdot h}{2} = \frac{p \cdot h_p}{2} = \frac{p \cdot q \cdot \sin\gamma}{2}$$

Den Zwischenwinkel γ berechnen wir mit dem Cosinussatz.

Cosinussatz: $c^2 = a^2 + b^2 - 2a \cdot b \cdot \cos\gamma \Rightarrow \gamma = \arccos\frac{a^2+b^2-c^2}{2 \cdot a \cdot b}$

3.1.4 Aufstellen der Qualitätsfunktion

A_{pq} sollte möglichst nahe bei $\frac{A}{2}$ liegen, s_{pq} möglichst nahe bei $\frac{U}{2}$.

Wir berechnen also die Abweichungen der beiden Umfänge und der beiden Flächen:

$\Delta U = \left|s_{pq} - \frac{U}{2}\right| = \left|p + q - \frac{a+b+c}{2}\right|$
$\Delta A = \left|A_{pq} - \frac{A}{2}\right| = \left|\frac{p \cdot q \cdot \sin\gamma}{2} - \frac{a \cdot b \cdot \sin\gamma}{4}\right| = \left|\frac{\sin\gamma}{2} \cdot (p \cdot q - \frac{a \cdot b}{2})\right|$

Eine von vielen möglichen Qualitätsfunktionen ist:

$QF(p,q) = \frac{1}{\Delta A^2 + \Delta U^2}$

Wir nehmen das Reziproke der Summe der quadrierten Residuen (Abweichungen) als Qualitätsfunktion: Durch die quadrierten Fehler fällt ein kleiner Fehler stärker ins Gewicht und wir müssen uns auch nicht mehr um negative Vorzeichen kümmern. Das Reziproke benützen wir, damit sich die Fitnessfunktion umgekehrt proportional zur Fehlerfunktion verhält, das heisst, dass ein kleiner Fehler zu einer grossen Fitness führt.

Als zweites bestimmen wir die Suchintervalle für die zu optimierenden Variablen p und q; p darf natürlich nur auf a und q nur auf b liegen:

$$p \in [0; a]$$

$$q \in [0; b]$$

Als Genauigkeit wählen wir 3 Stellen hinter dem Komma.

Wir können nun unser Programm mit der obigen Qualitätsfunktion und den Suchintervallen von p und q initialisieren und die Resultate auswerten.

3.2 Auswertung

An dieser Stelle möchten wir einige Tests präsentieren, um dem Leser zu zeigen, dass, und vor allen Dingen auf welche Art und Weise, unsere Implementierung funktioniert.

Wir möchten auch auf die beiden Analyse-Tools, die wir zur Verdeutlichung des Funktionsprinzips geschrieben haben, verweisen. Es handelt sich dabei um *triangle.pl* und *wrap_gnuplot.pl*.

triangle.pl ist ein Perl-Script, mit dessen Hilfe man von jeder Generation ein Bild mit dem Dreieck, der besten und der schlechtesten Gerade zeichnen kann. Diese Bilder lassen sich dann mit Hilfe von *mencoder* ([9]) zu einem Filmchen verarbeiten. triangle.pl basiert auf Funktionen der GD-Library für Perl. wrap_gnuplot.pl verarbeitet den Output des Hauptprogrammes so, dass er von einem Plot-Programm ([10] leicht verarbeitet werden kann. Wir haben von wrap_gnuplot.pl Gebrauch gemacht, um die Diagramme in den folgenden Abschnitten zu generieren. Beide Hilfsprogramme wurden nur unter Linux getestet.

Falls nicht anders erwähnt rechnen wir in den folgenden Tests mit einem Dreieck mit den Seiten a=2, b=4, c=5.

3.2.1 Gleichschenkliges Dreieck

Als erstes möchten wir zeigen, dass unser Programm wirklich funktioniert. Wir setzen deshalb ein gleichschenkliges Dreieck mit den Seiten a=3, b=4, c=4 in ga.c ein:

```
double sin_gamma=0.927025, A=2.78107, U=5.5; // equilateral triangle
```

Bei einem Testdurchlauf mit einer Population von 20'000 (#define POPULATION 20000) und einer Mutationsrate von 20% (#define MUTATION_RATE 0.2) haben wir nach 10 Generationen folgende Werte erhalten:

(10) Best fitness: 4e+10 ⇒ p=1.5 q=4 [...]

Die Strecke p nimmt also die Hälfte der Seite a und die Strecke q die ganze Seite b ein. Und genau das ist die optimale Lösung.

3.2.2 Populationsgrösse variieren

Wir haben 100 Zyklen einmal mit einer Populationsgrösse von 50 und einmal mit 10000 Individuen durchgerechnet. In den Diagrammen (Abbildungen 12 und 13) ist klar ersichtlich, dass der Fitness-Verlauf bei einer kleinen Population sehr volatil ist und die Fitness fast immer zwischen 0 und 2 liegt, bei einer grossen Population hingegen recht schnell auf das Optimum einschwenkt, aber immer noch recht volatil ist. Man darf hier aber nicht vergessen, dass der Zeitaufwand für die Berechnung mit der Populationsgrösse steigt. 100 Zyklen mit einer Population von 50'000 Individuen zu berechnen dauert auf einem Athlon 2000XP über 15 Minuten!

Wir führen den schlechten Verlauf bei kleinen Polulationen auf die Tatsache zurück, dass nicht genügend Erbgut vorhanden ist, und sich die Erbfaktoren nur ungenügend durchmischen. Bei einer Populationsgrösse von 2 ist dieses Phänomen noch viel stärker ausgeprägt, wir gehen deshalb davon aus, dass das geschieht, was in der Natur als Inzucht[5] bezeichnet wird. Dass der Verlauf immer wieder vom Optimum abweicht, hängt damit zusammen, dass ab und zu während einer Generation eine grosse Zahl von Individuen negativ mutiert wird.

3.2.3 Mutationsrate variieren

Dieser Störeffekt sollte sich eigentlich durch Reduktion der Mutationsrate vermindern lassen.

In der Tat haben wir bei einer Mutationsrate von 1% viel weniger Störungen (Abbildung 14).

[5] `http://de.wikipedia.org/wiki/Inzucht`

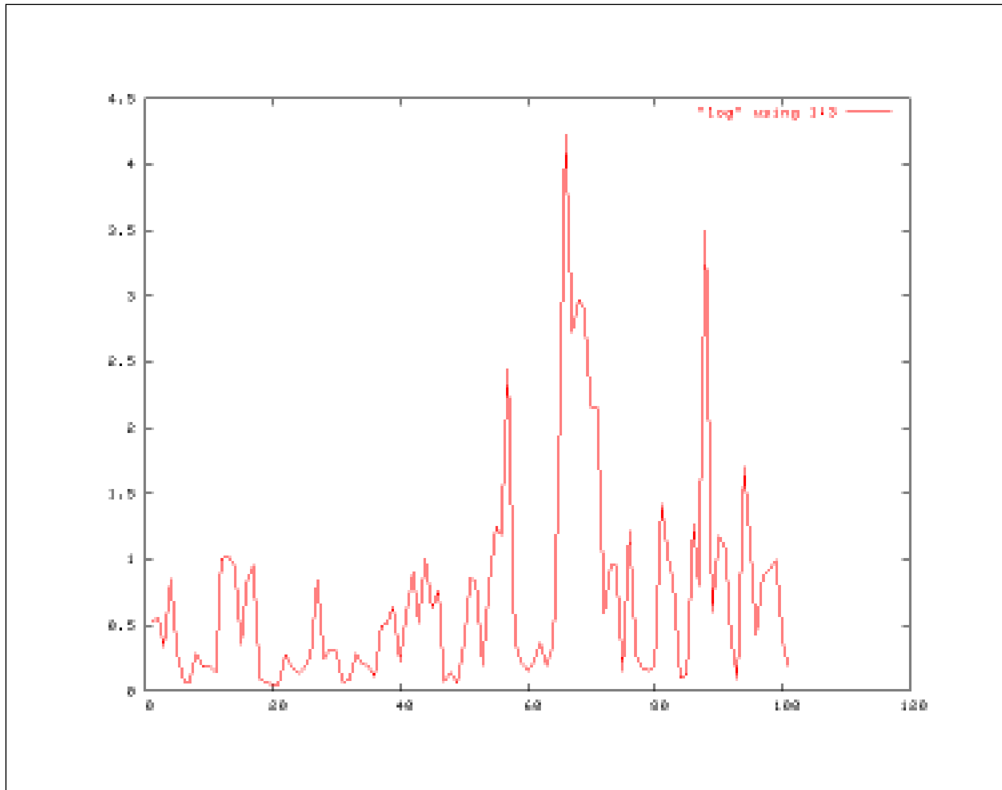


Abbildung 12: Fitnessverlauf einer Population von 50 Individuen (Mutationsrate 20%)

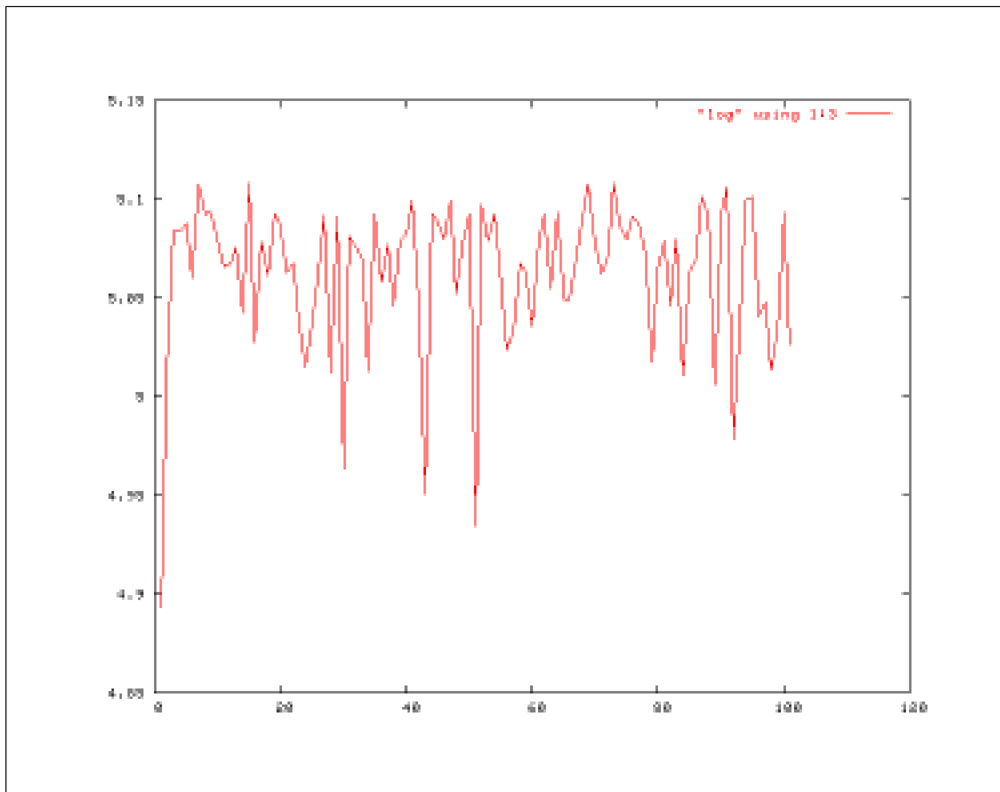


Abbildung 13: Fitnessverlauf einer Population von 10000 Individuen (Mutationsrate 20%)

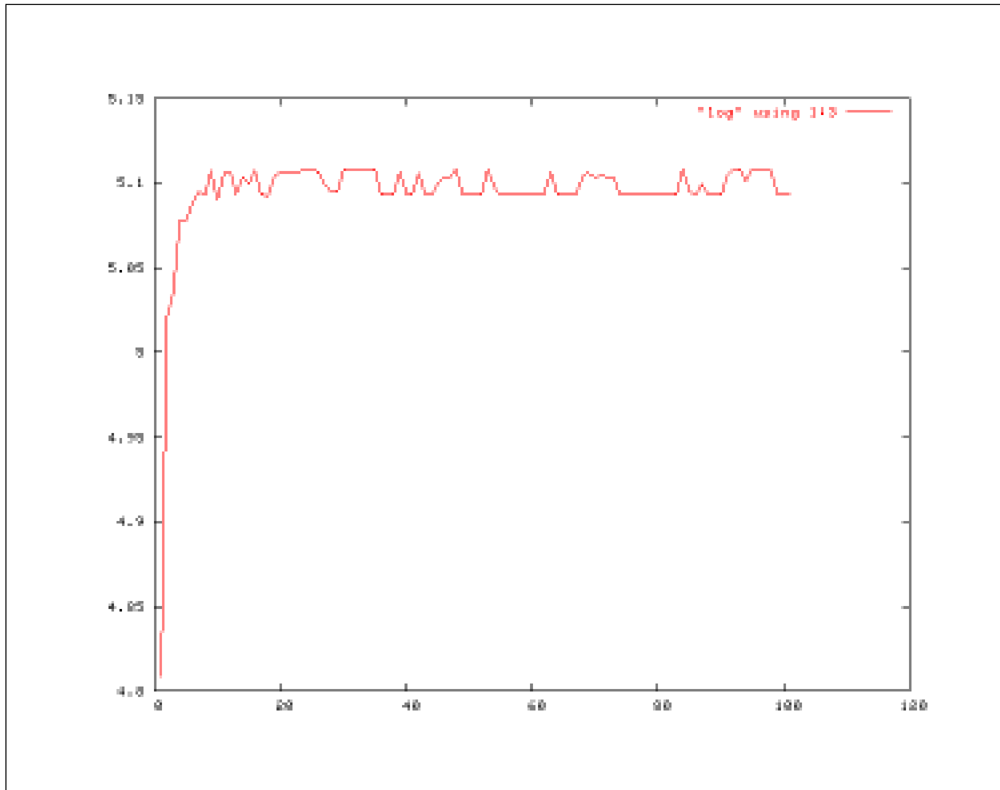


Abbildung 14: Fitnessverlauf einer Population von 10000 Individuen (Mutationsrate 1%)

Ebenfalls ist uns aufgefallen, dass das Optimum bei Durchläufen mit geringen Mutationsraten (unter 1%) erst nach vielen Generationen genau getroffen wird. Erstaunlich ist dabei, dass sehr schnell ein Wert gefunden wird, der sehr nahe an diesem Optimum liegt. Wir haben uns überlegt, weshalb dies geschieht, und sind zum Schluss gekommen, dass dieses Beinahe-Optimum eine sehr hohe Qualität hat und somit schon zu Beginn viele identische Individuen dieser Art existieren, welche alle anderen (schwächeren) Individuen verdrängen. Sind einmal alle Individuen in einer Population identisch, so kann nur durch eine Mutation ein neuartiges Individuum entstehen. Und je kleiner die Mutationsrate, desto unwahrscheinlicher ist eine solche Mutation.

3.3 Fazit

3.3.1 Auflösung des Dreiecksproblemes

Wir haben das Programm mit Variierung von Populationsgrösse und Mutationsrate sehr lange laufen lassen und sind auf folgendes Resultat[6] gekommen:

Best fitness: 5.1082 ⇒ p=1.108 q=4

Uns interessiert jetzt natürlich, ob dies die gesuchte Sehne ist, wir setzen also ein:

$$\begin{aligned} \Delta U &= \left|p + q - \tfrac{a+b+c}{2}\right| = \left|1.108 + 4 - \tfrac{2+4+5}{2}\right| = 0.392 \\ \Delta A &= \left|\tfrac{\sin\gamma}{2} \cdot (p \cdot q - a \cdot b)\right| = \left|\tfrac{0.949918}{2} \cdot (1.108 \cdot 4 - \tfrac{2 \cdot 4}{2})\right| = 0.205 \end{aligned}$$

Die Abweichungen sind im Verhältnis zu denen beim gleichschenkligen Dreieck, die ja gegen Null konvergierten, sehr gross. **Daraus müssen wir schliessen, dass die gesuchte Sehne bei einem beliebigen Dreieck nicht existiert.** Es gibt nur eine Sehne, die das Dreieck mit den *optimalen* Abweichungen von Fläche und Umfang teilt.

[6] Wir bleiben mit den Berechnungen beim Dreieck mit a=2, b=4 und c=5, haben aber auch sehr viele andere Seitenkonstellationen durchprobiert

3.3.2 Vor- und Nachteile des Genetischen Algorithmus'

Während der Recherche und der vielen Versuche kamen folgende Vor- und Nachteile ans Tageslicht:

Vorteile

- Hohe Flexibilität: Der GA lässt sich einfach anpassen, da er nur zwei Informationen über das Problem besitzen muss: Zielfunktion und Suchintervalle.
- Der GA vereint die Vorteile verschiedener Optimierungsmethoden: Er findet sehr wohl das Optimum, erkundet aber auch die Umgebung nach anderen, besseren Optima.
- Grosse Zukunftsaussichten: Genetische Programmierung

Nachteile

- Die optimalen Parameter (Populationsgrösse und Mutationsrate) sind eher schwer einzustellen; oft braucht der Anwender dazu einige Anläufe.
- Die Hin- und Herkodierungen und Fitnessberechnungen verschwenden Rechenzeit, was eine zu diesem Problem eventuell existierende, numerische Optimierungsmethode nicht tut.
- Der Anwender kann nie aussschliessen, dass er nun das absolute Optimum seiner Zielfunktion gefunden hat. Je länger er sein Programm laufen lässt, desto unwahrscheinlicher ist es, dass das Optimum noch nicht gefunden worden ist. Diese Wahrscheinlichkeit geht zwar gegen 0, wird aber nie 0 sein!

A Das Binärsystem

Das Binärsystem[7] ist das bekannteste und verbreitetste duale Zahlensystem / Stellenwertsystem zur Darstellung von Zahlen. Es verwendet die Basis 2, ist also die dyadische (2-adische) Darstellung von Zahlen. In diesem Zahlensystem gibt es nur zwei Ziffern, die im Binärsystem mit 0 und 1, in anderen dualen Systemen mit anderen Zeichen (zum Beispiel mit L und H) gekennzeichnet werden. Zahlen, die im Binärsystem dargestellt sind, nennt man Binärzahlen.

A.1 Darstellung von Binärzahlen

Die Darstellung der Zahlen erfolgt ähnlich wie die Darstellung im gewöhnlich verwendeten Dezimalsystem, mit dem Unterschied, dass die Wertigkeit der Ziffern nicht durch die entsprechende Zehnerpotenz, sondern durch die passende Zweierpotenz bestimmt wird. Beispielsweise stellt die Folge 1101 nicht (wie im Dezimalsystem) die Tausendeinhundertundeins dar, sondern die Dreizehn, denn im Binärsystem berechnet sich der Wert durch:

$$1101_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13_{10}$$

und nicht wie im Dezimalsystem durch:

$$1 \cdot 10^3 + 1 \cdot 10^2 + 0 \cdot 10^1 + 1 \cdot 10^0 = 1101_{10}$$

Die Klammerung der Resultate mit der tiefgestellten 10 soll darauf hinweisen, dass die Resultate im gebräuchlichen Dezimalsystem dargestellt sind.

[7] Ein Auszug aus [8]

B Literaturverzeichnis

Tutorials zu Genetischen Algorithmen

[1] http://www-ti.informatik.uni-tuebingen.de/~heim/lehre/proseminar_ss99/ausarbeitung/andreas_korsten/ausarbeitung.pdf: Tutorial von Andreas Korsten

[2] http://www.schatten.info/info/ga_lecture_notes/ga.pdf: Tutorial von Alexander Schatten

[3] http://www.ki.informatik.hu-berlin.de/lehre/ss02/EvoTechniken/V140502-1u2.pdf: Folien von L. Ivantysynova / L. Dölle

[4] http://www.stat.uni-muenchen.de/~krause/COURSES/Seminar_SS02/material/manuela_handout_ga.pdf: Zusammenfassung von Manuela Hummel

Biologische Grundlagen

[5] Bresch Hausmann: klassische und molekulare Genetik, 3. Auflage, 1972, ISBN: 3-540-05802-8 Springer Verlag Berlin, 0-387-05802-8 Springer Verlag New York

Anwendungsbeispiele

[6] http://home.pacbell.net/s-max/scott/simevol.html: Simevol, Anschauliches Java Applet mit Käfern, die geradlinig gehen lernen

Programmierung

[7] Guido Krüger: Go To C-Programmierung 2. Auflage, 1998, ISBN: 3-8273-1368-6 Adison-Wesley-Longman Bonn

Mathematische Grundlagen

[8] http://de.wikipedia.org/wiki/Bin%E4rsystem: Kurze Einführung in das Binärsystem aus dem freien Lexikon Wikipedia

Tools

[9] Mencoder ist ein Teil des mplayer Projekts: http://www.mplayerhq.hu/

[10] Gnuplot ist ein vielfältiges Werkzeug um Daten zu visualisieren: http://www.gnuplot.info/

[11] Mit LaTeXwurde die schriftliche Arbeit gemacht: http://www.latex-project.org/

[12] GCC ist der C-Compiler, den wir verwendet haben: http://gcc.gnu.org/

[13] Wir benutzten das altbewährte CVS, um unseren Code zu verwalten: http://www.cvshome.org/

C Der Inhalt der CD

Folgende Verzeichnisse sind auf der beigelegten CD und auf der Homepage `http://www.larynx.ch/evolopt/` zu finden:

- / Hauptprogramm *ga.c, Makefile* und *README*
- **/CVS** Informationen über die Version auf der CD
- **/analyze** Enthält die unter **Analysetools** angesprochenen Hilfsmittel
- **/binaries** Hier sind Binaries für Windows und Linux zu finden (i386-Architektur)
- **/code_snipplets** Code-Schnippsel zur Erläuterung diverser Funktionen
- **/docs** Die schriftliche Arbeit als Latex-Source und pdf
- **/docs/pictures** Bilder zur schriftlichen Arbeit
- **/include** Enthält die verschiedenen Layer, welche zusammen die Lib bilden

D Programmcode

D.1 ga.c

```
/* ---------------------------------------------------------------------------
File:        ga.c
Description: Main file, where the whole algorithm is assembled
Version:     See CVS
Authors:     Cédric Huwyler, Mathias Weyland
Copyright:   (c) 2003 by Cédric Huwyler, Mathias Weyland
License:     General Public License (GPL)
-------------------------------------------------------------------------- */



/*
 This program runs a genetic algorithm. For information about how a genetic
 algorithm works, use the documentation included in docs/ or
 http://www.larynx.ch/evolopt/.
 This genetic algorithm solves the triangle problem numerically. Please
 consider the documentation for further information about the triangle
problem
 (Chap. 4).

 The triangle's sides are: a=2, b=4, c=5
 (Consider code_snipplets/compute_triangle.c if you like to use other
triangles)
*/

/* Define the genetic algorithm's parameters */
#define POPULATION 5000    /* even number! */
#define MUTATION_RATE 0.2 /* between 0 and 1, usually about 10%-20% */
#define MAX_GENERATIONS 0 /* Break and return after this number of generations
(0 means infinite) */
#define F_OFF 0        /* Truncate the algorithm when quality function returns
a number lower than F_OFF */
/* #define DEBUG */    /* Uncomment this to allow verbose mode */

/* Include common C libraries */
#include <stdlib.h>
#include <time.h>
#include <math.h>

/* Genetic Algorithm header files (the commented ones don't work yet) */
#include "include/encoding.h"
#include "include/binary.h"
```

```
#include "include/selection.h"
#include "include/genetic_operators.h"
#include "include/evaluation.h"
#include "include/debug.h"


/* quality function for the triangle problem */
double qf(double *params)
{
 /* -> a=3 b=4 c=4 .. isn't it ? */
 /* double sin_gamma=0.927025, A=2.78107, U=5.5; // equilateral triangle */
 double sin_gamma=0.949917759598, A=1.8998355192, U=5.5; // general triangle
 double A_pq=(params[0]*params[1]*sin_gamma) / 2;
 double U_pq=params[0]+params[1];

 /* The interval is too big; we don't like p,q to be larger than a,b */
 if(params[0]>2) return 0.0000000001;
 if(params[1]>4) return 0.0000000001;

 return 1 / (pow(A-A_pq,2) + pow(U-U_pq,2));
}

int main()
{
 chsom *population, *pool;  /* byte matrix to store the population's and
genome pool's chromosomes */
 chsom fittest; /* Store the fittest chromosome to prevent losing the best
value by mutation or crossover */
 parameter parameters[2]; /* The parameter specifications (see below) */
 unsigned int chsom_size; /* The size of a chromosome in bytes: often used */
 int generation=1; /* The current generation */
 int p_amount; /* The number of parameters */

 /* Define parameters (low := lower bound, high := upper bound, prec := post
decimal positions) */

 /* p */
 parameters[0].name="p";
 parameters[0].low=0;
 parameters[0].high=2;
 parameters[0].prec=3;

 /* q */
 parameters[1].name="q";
 parameters[1].low=0;
 parameters[1].high=4;
```

```
 parameters[1].prec=3;

 /* Compute amount of parameters - often needed */
 p_amount=sizeof(parameters)/sizeof(parameter);

 /* Encode chromosomes and create a population and a genome pool array */
 population=encode(parameters, p_amount);
 pool=encode(parameters, p_amount);

 /* Compute the size of a chromosome in bytes - this value is often needed to
avoid pointer problems */
 chsom_size=compute_bytes(parameters, p_amount);

 /* Initialize population with random values */
 init(population, chsom_size);

 /* Print some informational stuff */
 printf("\n\n");
 printf("Population size:        %d Chromosomes\n", POPULATION);
 dump_length(parameters, p_amount);
 printf("Chromosome length:      %d Bytes\n", chsom_size);
 printf("Population initialized: Starting algorithm...\n\n\n");


 /* Run life ^^ */
 while(evaluate(population, parameters, p_amount, generation))
 {
  #ifdef DEBUG
  debug_population(population, chsom_size, parameters, p_amount, generation);
  getchar();
  #endif

  dump_extremes(population, parameters, p_amount, generation);
  generation++;
  selection(population, pool, chsom_size, parameters, p_amount);
  reproduce(population, pool, chsom_size);
 }

 printf("\nThe program has computed the following result:\n");
 dump_extremes(population, parameters, p_amount, generation);

 /* Be proper and free the allocated memory */
 free(population), free(pool);
}
```

D.2 include/binary.h

```
/* ---------------------------------------------------------------------------
File:        binary.h
Description: Subroutines for binary manipulations on chromosomes
Version:     See CVS
Authors:     Cédric Huwyler, Mathias Weyland
Copyright:   (c) 2003 by Cédric Huwyler, Mathias Weyland
License:     General Public License (GPL)
--------------------------------------------------------------------------- */



/* Read a specific bit of a given chromosome */
int read_bit(int pos, chsom chromosome)
{
 int byte, bit;

 /* Compute the current byte and bit in the chromosome */
 byte=(int) (((float)pos-1.0)/8);
 bit=pos-byte*8;

 /* See code_snipplets/bol/readbit.c for explanations */
 return ((chromosome[byte]&(1<<(bit-1)))==0) ? 0 : 1;
}

/* Set a specific bit of a given chromosome to a specific value */
int set_bit(int pos, chsom chromosome, int value)
{
 int byte, bit;

 /* Compute the current byte and bit in the chromosome */
 byte=(int) (((float)pos-1.0)/8);
 bit=pos-byte*8;

 /* See code_snipplets/bol/setbit.c for explanations */
 if(value) chromosome[byte]=chromosome[byte]|(1<<(bit-1));
 else chromosome[byte]=chromosome[byte]&(~(1<<(bit-1)));
}

/* Randomize every bit of a given chromosome */
int rand_bits(chsom chromosome, unsigned int length)
{
 int i;
 for(i=1;i<=length;i++) set_bit(i, chromosome, rand() % 2);
}
```

```
/* A subroutine to invert a particular bit in a chromosome (=> mutation) */
int invert_bit(int pos, chsom chromosome)
{
 set_bit(pos, chromosome, !read_bit(pos, chromosome));
}

/* Swap bits of two given bitstrings at a given range (pos1-pos2) */
chsom *swap_bits(int pos1, int pos2, chsom parent1, chsom parent2, unsigned
int chsom_size)
{
 int i;
 chsom *children=create_population(2, chsom_size);

 /* Copy the parents to the children */
 memcpy(children[0], parent1, chsom_size);
 memcpy(children[1], parent2, chsom_size);

 /* Swap bits between pos1 and pos2 */
 for(i=pos1;i<=pos2;i++)
 {
  set_bit(i, children[0], read_bit(i, parent2));
  set_bit(i, children[1], read_bit(i, parent1));
 }

 return children;
}

/* Initialize population by randomizing its chromosomes */
int init(chsom *population, unsigned int chsom_size)
{
 unsigned int i;

 /* Set random seed to time */
 srand((unsigned)time((time_t*)0));

 /* Randomize every bit in the population matrix (POPULATION, chsom_size*8)
*/
 for(i=0;i<POPULATION;i++) rand_bits(population[i], chsom_size*8);
}
```

D.3 include/debug.h

```
/* ------------------------------------------------------------------------
File:        debug.h
Description: Subroutines for debug outputs and dumps
Version:     See CVS
Authors:     Cédric Huwyler, Mathias Weyland
Copyright:   (c) 2003 by Cédric Huwyler, Mathias Weyland
License:     General Public License (GPL)
--------------------------------------------------------------------- */


/* Global debug subroutine */
int debug_population(chsom *population, int chsom_size, parameter *parameters,
int p_amount, int generation)
{

printf("\n\n---------------------------------------------------------------\n");
 printf(" Generation #
printf("---------------------------------------------------------------\n");
 dump_population(population, chsom_size);
 dump_parameters(population, parameters, p_amount);
 dump_fitness(population, parameters, p_amount);
 dump_fitness_interval(population, parameters, p_amount);
}

/* Dump the amount of bits needed for every parameter */
int dump_length(parameter *parameters, int p_amount)
{
 int i;

 printf("\nParameter bit lengths:\n");
 for(i=0;i<p_amount;i++) printf("Parameter %s: %d bits\n", parameters[i].name,
parameters[i].bits);
 printf("\n");
}

/* Dump the chromosoms as hexadecimal numbers (for every byte) */
int dump_population(chsom *population, unsigned int chsom_size)
{
 int i,j;

 printf("\n\nChromosomes:   \n--------------------------------\n");

 for(i=0;i<POPULATION;i++)
 {
```

```
  printf("(%4d)  ", i+1);
  for(j=0;j<chsom_size;j++)
  {
   printf("%02x ", population[i][j]);
  }
  printf("\n");
 }
}

/* Dump the chromosomes' decoded parameters */
int dump_parameters(chsom *population, parameter *parameters, int p_amount)
{
 int i,j;
 double *params;

 printf("\n\nParameters:    \n--------------------------------\n");

 for(i=0;i<POPULATION;i++)
 {
  printf("(%4d)  ", i+1);
  params=decode(population[i], parameters, p_amount);
  for(j=0;j<p_amount;j++)
  {
   printf("%s=%lg  ", parameters[j].name, params[j]);
  }
  free(params);
  printf("\n");
 }
}

/* Dump the chromosomes' fitness values */
int dump_fitness(chsom *population, parameter *parameters, int p_amount)
{
 int i;
 double *ind_fitness;

 printf("\n\nFitness values:\n--------------------------------\n");

 ind_fitness=compute_fitness(population, parameters, p_amount);

 for(i=0;i<POPULATION;i++)
 {
  printf("(%4d)  %lg\n", i+1, ind_fitness[i]);
 }
}
```

```
/* Dump the fitness interval */
int dump_fitness_interval(chsom *population, parameter *parameters, int
p_amount)
{
 int i;
 double *interval;

 printf("\n\nFitness interval:\n------------------------------\n");

 interval=fitness_interval(population, parameters, p_amount);

 for(i=0;i<POPULATION;i++)
 {
  printf("(%4d)  %lg\n", i+1, interval[i]);
 }
}

/* Compute fitness average */
double compute_average(chsom *population, parameter *parameters, int p_amount)
{
 int i;
 double average=0;

 for(i=0;i<POPULATION;i++)
 {
  average += fitness(population[i], parameters, p_amount);
 }

 average = average / ((double) POPULATION);
 return average;
}


/* Dump fittest and worst chromosome */
int dump_extremes(chsom *population, parameter *parameters, int p_amount, int
generation)
{
 /* Get best fittness */
 int i=get_fittest(population, parameters, p_amount), j;

 /* Get average fitness */
 double *params=decode(population[i], parameters, p_amount);
 double avg_fitness=compute_average(population, parameters, p_amount);

 /* Dump best fitness */
 printf("(%4d) Best fitness: %lg => ", generation, fitness(population[i],
```

```
parameters, p_amount), i+1);
 for(j=0;j<p_amount;j++)
 {
  printf("%s=%lg ", parameters[j].name, params[j]);
 }

 /* Get worst fitness */
 i=get_worst(population, parameters, p_amount);
 params=decode(population[i], parameters, p_amount);

 /* Dump worst fitness */
 printf(" | worst fitness: %lg => ", fitness(population[i], parameters,
p_amount), i+1);
 for(j=0;j<p_amount;j++)
 {
  printf("%s=%lg ", parameters[j].name, params[j]);
 }

 /* Dump average fitness */
 printf(" | average fitness: %lg\n", avg_fitness);

 /* Free the allocated memory */
 free(params);
}

/* Routine to track Segfaults */
int mark()
{
 printf("\n\n--- MARK ---\n\n");
}
```

D.4 include/encoding.h

```
/* -------------------------------------------------------------------------
File:        encoding.h
Description: Subroutines to encode parameters into a population matrix and
             to decode parameters from chromosomes.
Version:     See CVS
Authors:     Cédric Huwyler, Mathias Weyland
Copyright:   (c) 2003 by Cédric Huwyler, Mathias Weyland
License:     General Public License (GPL)
------------------------------------------------------------------------- */



/* Create the chromosom data type (a chromosom is a char (byte) array) */
typedef unsigned char *chsom;

/* Create a structure type for parameters */
typedef struct {
 double low;  /* lower boundary */
 double high; /* upper boundary */
 int prec;    /* post decimal positions */
 int bits;    /* amount of bits needed -> will be computed */
 char *name;  /* name of this parameter -> for debug routines */
} parameter;

/* Compute the bits needed for every parameter */
int compute_bits(parameter *parameters, int p_amount)
{
 int i;

 /* See documentation for an explanation of this calculation */
 for(i=0;i<p_amount;i++) parameters[i].bits=(int)
ceil(log((parameters[i].high-parameters[i].low)*pow(10,parameters[i].prec))/log(2));
}

/* This subroutine computes the size of a chromosom in bytes (rounded up) */
int compute_bytes(parameter *parameters, int p_amount)
{
 int i;
 double sum=0;

 /* Compute the bits needed for every chromosome first */
 compute_bits(parameters, p_amount);

 /* Add up the bits to the chromosome length */
```

```
 for(i=0;i<p_amount;i++) sum+=parameters[i].bits;

 /* Convert bits to bytes and round up if necessary */
 return (int) ceil(sum/8);
}

/* Create a two-dimensional array */
chsom *create_population(population_size, chsom_length)
{
 unsigned char **p;
 int i;

 p = (unsigned char**) malloc(sizeof(unsigned char*) * population_size);
 for (i=0;i<population_size;i++)
  p[i] = (unsigned char*) malloc(sizeof(unsigned char) * chsom_length);
 return p;
}

/* Create a population fitting to the parameters */
chsom *encode(parameter *parameters, int p_amount)
{
 return create_population(POPULATION, compute_bytes(parameters, p_amount));
}

/* Decode a chromosome back into parameters */
double *decode(chsom chromosome, parameter *parameters, int p_amount)
{
 int i, j, offset=1;
 double *retval=(double*) malloc(sizeof(double)*p_amount);

 for(i=0;i<p_amount;i++)
 {
  retval[i]=0;

  /* See documentation for an explanation of this calculation */
  for(j=0;j<parameters[i].bits;j++)
  {
   int bit=read_bit(offset+j, chromosome);
   retval[i]+=bit*pow(2,j);
  }

  offset+=parameters[i].bits;
  retval[i]=retval[i]/pow(10,parameters[i].prec)+parameters[i].low;
 }

 return retval;
```

```
}
```

D.5 include/evaluation.h

```
/* ------------------------------------------------------------------------
File:        evaluation.h
Description: Subroutines to evaluate a population and decide whether a
             truncation criteria is fulfilled.
Version:     See CVS
Authors:     Cédric Huwyler, Mathias Weyland
Copyright:   (c) 2003 by Cédric Huwyler, Mathias Weyland
License:     General Public License (GPL)
------------------------------------------------------------------------ */



/* Return 0 if one of the break conditions is reached */
int evaluate(chsom *population, parameter *parameters, int p_amount, int
generation) /* more parameters later */
{
 int i;

 /* A defined generation (MAX_GENERATIONS) reached */
 if(MAX_GENERATIONS!=0 && generation>MAX_GENERATIONS)
 {
  printf("Algorithm truncated by reason: MAX_GENERATIONS reached\n");
  return 0;
 }

 /* The fitness function returns a value bigger than F_OFF */
 i=get_fittest(population, parameters, p_amount);
 if(fitness(population[i], parameters, p_amount) < F_OFF)
 {
  printf("Algorithm truncated by reason: F_OFF reached\n");
  return 0;
 }

 /* No truncation criterion reached: continue */
 return 1;
}
```

D.6 include/genetic_operators.h

```
/* -------------------------------------------------------------------------
File:        genetic_operators.h
Description: Subroutines to reproduce chromosomes into genetically new ones
Version:     See CVS
Authors:     Cédric Huwyler, Mathias Weyland
Copyright:   (c) 2003 by Cédric Huwyler, Mathias Weyland
License:     General Public License (GPL)
------------------------------------------------------------------------- */


/* mutate every bit in chromosome with the probability defined in
MUTATION_RATE */
/* See code_snipplets/mutation_rate.c */
int mutate(chsom *population, unsigned int chsom_size)
{
 int i,j;
 for(i=0;i<POPULATION;i++)
 {
  for(j=1;j<=chsom_size*8;j++)
  {
   if(((double) rand()) / ((double)RAND_MAX+1.0) < MUTATION_RATE)
invert_bit(j, population[i]);
  }
 }
}

/* Do crossover at given points and return a population with two children */
chsom *crossover(chsom parent1, chsom parent2, unsigned int chsom_size)
{
 int pos1, pos2;

 /* A random starting point in the chromosome */
 pos1=rand() % (chsom_size*8) + 1;

 /* Take one of the remaining bits as upper crossover bound */
 if(pos1==chsom_size*8) pos2=pos1; /* x % 0 leads to a division by zero */
 else pos2=pos1+rand() % (chsom_size*8-pos1);

 /* Swap bits at between positions */
 return swap_bits(pos1, pos2, parent1, parent2, chsom_size);
}

/* Reproduce the genome pool with crossover and mutation */
chsom *reproduce(chsom *population, chsom *pool, unsigned int chsom_size)
```

```
{
 int i;
 chsom *children;

 /* Do possible mutations */
 mutate(pool, chsom_size);

 /* do crossover and store new generation in the population array */
 for(i=0;i<POPULATION;i+=2)
 {
  children=crossover(pool[i], pool[i+1], chsom_size);

  /* Copy children into population (C can't assign arrays to arrays..) */
  memcpy(population[i], children[0], chsom_size);
  memcpy(population[i+1], children[1], chsom_size);

  /* This part of the memory isn't used anymore => we don't want a memory leak
*/
  free(children[0]);
  free(children[1]);
  free(children);
 }
}
```

D.7 include/selection.h

```
/* ------------------------------------------------------------------------
File:        selection.h
Description: Subroutines to select the fittest individuals for reproduction
Version:     See CVS
Authors:     Cédric Huwyler, Mathias Weyland
Copyright:   (c) 2003 by Cédric Huwyler, Mathias Weyland
License:     General Public License (GPL)
------------------------------------------------------------------------ */



/* Declare functions only defined later */
double qf();

/* Compute a chromosome's fitness value ---> Really a double ? */
double fitness(chsom chromosome, parameter *parameters, int p_amount)
{
 /*
 The quality functions returns the residuals => compute the reciprocal to get
 a better fitness for smaller residuals
 */

 /* Compute residual */
 double *params=decode(chromosome, parameters, p_amount);
 double residual=qf(params);
 free(params);

 return residual;
}

/* Return an array containing fitness values for each chromosom */
double *compute_fitness(chsom *population, parameter *parameters, int
p_amount)
{
 int i;
 double *ind_fitness=(double*) malloc(sizeof(double)*POPULATION);

 for(i=0;i<POPULATION;i++) ind_fitness[i]=fitness(population[i], parameters,
p_amount);

 return ind_fitness;
}

/* Get the fittest chromome */
```

```
int get_fittest(chsom *population, parameter *parameters, int p_amount)
{
 int i, fittest=0;

 /* Initialize the maximum with the population's first element */
 double f, max=fitness(population[0], parameters, p_amount);

 /* Compute maximum */
 for(i=1;i<POPULATION;i++)
 {
  f=fitness(population[i], parameters, p_amount);
  if(f>max)
   {
    max=f;
    fittest=i;
   }
 }

 return fittest;
}


/* Get the worst chromosome */
int get_worst(chsom *population, parameter *parameters, int p_amount)
{
 int i, worst=0;

 /* Initialize the minimum with the population's first element */
 double f, min=fitness(population[0], parameters, p_amount);

 /* Compute minimum */
 for(i=1;i<POPULATION;i++)
 {
  f=fitness(population[i], parameters, p_amount);
  if(f<min)
  {
   min=f;
   worst=i;
  }
 }

 return worst;
}


/* Create array with each individual's upper fitness bound in [0;1] */
```

```
double *fitness_interval(chsom *population, parameter *parameters, int
p_amount)
{
 int i;
 double *ind_fitness;
 double sum=0, total=0;

 /* Compute relative fitness values (division by total fitness) */
 ind_fitness=compute_fitness(population, parameters, p_amount);
 for(i=0;i<POPULATION;i++) total+=ind_fitness[i];
 for(i=0;i<POPULATION;i++) ind_fitness[i]/=total;

 /* Compute upper bound in fitness interval for each chromosome */
 for(i=0;i<POPULATION;i++)
 {
  sum+=ind_fitness[i];
  ind_fitness[i]=sum;
 }

 return ind_fitness;
}

/* Get the chromosome with this (randomized) upper interval bound [value] */
int get_from_interval(chsom *population, double *fitness_interval, double
value)
{
 int i;

 /* Count interval up as long as value doesn't fit in */
 for(i=0;i<POPULATION;i++)
 {
  if(value<fitness_interval[i]) return i;
 }
}

/* Create the genome pool */
chsom *genome_pool(chsom *population, chsom *pool, double *fitness_interval,
unsigned int chsom_size)
{
 int i,j;
 double rand_value;

 /* bomb interval with random values */
 for(i=0;i<POPULATION;i++)
 {
  rand_value=((double) rand()) / ((double) RAND_MAX+1);
```

```
  j=get_from_interval(population, fitness_interval, rand_value);

  /* Copy the chromosome string to the genome pool */
  memcpy(pool[i], population[j], chsom_size);
 }

 /* This part of the memory isn't used anymore => we don't want a memory leak
*/
 free(fitness_interval);
}

/* Do the selection process and return a genome pool with selected chromosomes
*/
chsom *selection(chsom *population, chsom *pool, unsigned int chsom_size,
parameter *parameters, int p_amount)
{
 genome_pool(population, pool, fitness_interval(population, parameters,
p_amount), chsom_size);
}
```

E Analysetools

Wie wir im Abschnitt über die Tests erwähnt haben, haben wir einige Tools erstellt, um uns die Arbeit ein wenig zu erleichtern.

Diese Tools enthalten relativ viel auskommentierten Code, damit wir bei Bedarf auf alternative Funktionen zugreiffen können (Zum Beispiel Parameter statt Fitness loggen).

E.1 triangle.pl

Generiert ein Bild von jeder Generation mit dem Dreieck, der besten und der schlechtesten Gerade. Gegebenenfalls müssen die Seiten, die Höhe h_c und der Umrechnungsfaktor zwischen Einheiten und Pixel und die Auflösung des Bildes (ab Programmzeile 46) angepasst werden.

```
#!/usr/bin/perl

# Usage: ./ga | analyze/triangle.pl

# Let's make some funny pictures:
# mencoder "mf://*.jpg" -mf fps=25 -o output.avi -ovc divx4
# to create an avi video

use GD;

# Define the image format (png, jpeg)

$format="jpeg";

# We need i for the image's filename
$i =0;

while(<STDIN>)
{

 if($_=~/Best fitness/)
 {
        # Extract p and q

        $pmax = $_;
        $pmax =~ s/^[^p]+p=//;
        $pmax =~ s/ .+$//;

        $qmax = $_;
        $qmax =~ s/^[^q]+q=//;
        $qmax =~ s/ .+$//;

        $pmin = $_;
        $pmin =~ s/^.+worst//;
```

```
$pmin =~ s/^[^p]+p=//;
$pmin =~ s/ .+$//;

$qmin = $_;
$qmin =~ s/^.+worst//;
$qmin =~ s/^[^q]+q=//;
$qmin =~ s/ .+$//;


# Define our triangle... $h is hc

$h = 1.5;
$a = 2;
$c = 5;

#$p = 1.9;
#$q = 4;

# Relation between units and pixels

$factor = 70;

# Render new image, resolution 500x200

$image = new GD::Image(500,200);

$d=sqrt(($a*$a)-($h*$h));

#We need some colours

$white = $image->colorAllocate(255,255,255);
$black =$image->colorAllocate(0,0,0);
$blue = $image->colorAllocate(0,160,255);
$red = $image->colorAllocate(255,0,0);

# Calcuate black triangle

$poly = new GD::Polygon;
$poly -> addPt(50,50);
$poly -> addPt(50+$factor*$c,50);
$poly -> addPt(50+$factor*$d,50+$factor*$h);

# Calculate blue triangle (in motion)

$filled = new GD::Polygon;
$filled -> addPt(50+$d*$factor,50+$factor*$h);

$x = $pmax/sqrt(1+(($h*$h)/($d*$d)));
$y = $x * $h/$d;

$filled -> addPt(50+($d-$x)*$factor, 50+($h-$y)*$factor);
```

```
        $e = $c - $d;
        $x = $qmax/sqrt(1+(($h*$h)/($e*$e)));
        $y = $x * $h/$e;
                                                                        90
        $filled -> addPt(50+($d+$x)*$factor, 50+($h-$y)*$factor);

        # Calculate red triangle (worst)

        $worst = new GD::Polygon;
        $worst -> addPt(50+$d*$factor,50+$factor*$h);

        $x = $pmin/sqrt(1+(($h*$h)/($d*$d)));
        $y = $x * $h/$d;
                                                                        100
        $worst -> addPt(50+($d-$x)*$factor, 50+($h-$y)*$factor);

        $e = $c - $d;
        $x = $qmin/sqrt(1+(($h*$h)/($e*$e)));
        $y = $x * $h/$e;

        $worst -> addPt(50+($d+$x)*$factor, 50+($h-$y)*$factor);

        # Render the triangles
                                                                        110
        $image -> filledPolygon($filled, $blue);
        $image -> filledPolygon($worst, $red);
        $image -> polygon($poly, $black);

        # The image is upside down; the origin is in the upper left corner
        # Flip the image to correct that.

        $image -> flipVertical();

        # Save the image                                                120

        print "Creating image #" . sprintf("%04d", $i) . "...\n";
        open(INFO, ">./image" . sprintf("%04d", $i) . ".$format"); # Open for output
        print INFO $image->$format;
        close INFO;

        $i++;
  }
}
```

E.2 wrap_gnuplot.pl

Formt den Output des Hauptprogrammes um, so dass er von einem Plot-Programm wie *gnuplot* ([10]) verarbeitet werden kann. Der Output dieses Tools kann in ein Logfile umgeleitet werden.

```perl
#!/usr/bin/perl

# Make use of this tool if you want to throw
# the values into a logfile
# syntax: ./ga | analyze/wrap_gnuplot.pl > log.txt

$i =1;
while(<STDIN>)
{

# print;

 if($_=~/Best fitness/)
 {
        # Extract p and q

        #$pmax = $_;
        #$pmax =~ s/^[^p]+p=//;
        #$pmax =~ s/ .+$//;
        #chomp $pmax;

        #$qmax = $_;
        #$qmax =~ s/^[^q]+q=//;
        #$qmax =~ s/ .+$//;
        #chomp $qmax;

        #$pmin = $_;
        #$pmin =~ s/^.+worst//;
        #$pmin =~ s/^[^p]+p=//;
        #$pmin =~ s/ .+$//;
        #chomp $pmin;

        #$qmin = $_;
        #$qmin =~ s/^.+worst//;
        #$qmin =~ s/^[^q]+q=//;
        #$qmin =~ s/ .+$//;
        #chomp $qmin;

        $max = $_;
        $max =~ s/^.+Best fitness: //;
        $max =~ s/ .+$//;
        chomp $max;

        $min = $_;
        $min =~ s/^.+worst fitness: //;
        $min =~ s/ .+$//;
        chomp $min;

        $average = $_;
        $average =~ s/^.+average fitness: //;
        chomp $average;
```

```
        print "$i $min $max $average\n";

        $i++;
 }


}
```

E.3 compute_triangle.c

Berechnet die halbe Fläche, den halben Umfang und den Winkel γ anhand der drei gegebenen Dreiecksseiten. Diese Werte können anschliessend ins Hauptprogramm (ga.c) eingefügt werden (Siehe Abschnitt **Auswertung**).

```
001 #define _GNU_SOURCE
002 #include <stdlib.h>
003 #include <math.h>
004
005 int main(int argc, char **argv)
006 {
007  double a,b,c,A,U,sin_gamma;
008
009  if(argc!=4)
010  {
011   printf("Usage: compute_triangle <a> <b> <c>\n\n");
012   exit(1);
013  }
014
015  else
016  {
017   a=atof(argv[1]);
018   b=atof(argv[2]);
019   c=atof(argv[3]);
020
021   printf("a: %lg  b: %lg  c: %lg\n", a,b,c);
022
023   sin_gamma=sin(acos((a*a+b*b-c*c)/(2*a*b)));
024
025   U=(a+b+c)/2;
026   A=(a*b*sin_gamma)/4;
027
028   printf("sin(gamma): %lg\nA/2: %lg\nU/2: %lg\n", sin_gamma, A, U);
029  }
030 }
031

032
```

F Die Autoren

Cédric Huwyler
Waldstrasse 1
5222 Umiken
Email: me@larynx.ch

Mathias Weyland
Rebmoosweg 43d
5200 Brugg
Email: mathias@weyland.ch

Mentor: Michael Kalkhi
Gen. Guisan - Strasse 52
5000 Aarau
Email: michael@kalkhi.ch

G Danksagungen

Wir möchten uns bei folgenden Personen und Institutionen bedanken:

Michael Kalkhi

Michael Kalkhi ist der Mentor unseres Projektes und hat uns bei der Erstellung dieser wissenschaftlichen Arbeit aufopferungsvoll unterstützt.

Stephan Näf

Stephan Näf hat sich dazu bereit erklärt, als zweite Beurteilungsperson an unserem Projekt mitzuwirken.

Stephan Berner

Stephan Berner hat uns seinen Server für das Hosting unserer Projekthomepage zur Verfügung gestellt.

Dr. Remo Badii

Remo Badii hat uns tatkräftig im Bereich der angewendeten Seitenbeschreibungssprache LATEXunterstützt.

Linux Users Group Switzerland (LUGS)

Auch Mitglieder der LUGS haben uns unermüdlich in Sachen LATEXunterstützt.

Thierry Dussuet

Er hat sich am späten Abend noch bereit erklärt, unsere Arbeit nach Schreib- und Denkfehlern zu durchleuchten.

Opensource Community

Wir möchten auch der ganzen Opensource Community für die Bereitstellung von freier Software danken. Ein besonderes Dankeschön geht an die Entwickler von *GCC* ([12]), *CVS* ([13]), *GNUplot* ([10]) und *LATEX* ([11]).