Paul E. Sevinç
Weidstrasse 14
9302 Kronbühl

Ein Framework für die Mehrzieloptimierung mit Genetischen Algorithmen

Semesterarbeit

Betreuende Professur:
Eckart Zitzler
Prof. Dr. Lothar Thiele
Institut für Technische Informatik und Kommunikationsnetze
Abteilung Elektrotechnik, ETH Zürich

5. Februar 1999

Abstract

The goal of this semester project was to design and implement a framework for multiobjective optimization with evolutionary algorithms, with special emphasis put on genetic algorithms. What resulted is *FEMO*, a *Framework for Evolutionary Multiobjective Optimization*.

FEMO features a set of building blocks that can be combined in flexible ways to adapt to the problem at hand: Three genetic algorithms for multiobjective optimization perform fitness assignment and selection. A container manages individuals and can be used for populations, mating pools, and Pareto sets. An abstract class defines the interface of individuals but only needs to be extended with problem-specific code. And different kinds of chromosomes with several crossover and mutation operators are provided and can be used within a single individual.
FEMO's open architecture also supports its own evolution over time.

This report introduces the field of evolutionary algorithms and serves as a tutorial and reference to FEMO.

Inhaltsverzeichnis

1. Einleitung

Der vorliegende Bericht entstand im Rahmen einer Semesterarbeit im Wintersemester 1998/1999 an der Abteilung Elektrotechnik der Eidgenössischen Technischen Hochschule Zürich (ETHZ). Ausgeschrieben und betreut wurde sie von *Eckart Zitzler* und *Prof. Dr. Lothar Thiele* vom Institut für Technische Informatik und Kommunikationsnetze (TIK).

Mehrzieloptimierung und evolutionäre Algorithmen (kurz EAs) bilden den Hintergrund dieser Arbeit. Von Mehrzieloptimierung spricht man, wenn mögliche Lösungen eines Problems nach mehreren, im Konflikt stehenden Kriterien beurteilt werden müssen. Evolutionäre Algorithmen zeichnen sich dadurch aus, dass sie parallel auf einer Menge von möglichen Lösungen operieren und zu deren Optimierung Mechanismen der Biologie nachahmen.

Den Kern der Arbeit bildet ein Software-Gerüst (*framework*), implementiert in C++, welches die Realisierung von Mehrzieloptimierern basierend auf genetischen Algorithmen (kurz GAs) erleichtern soll. Das Entwicklungsziel war ein einfach zu benutzendes, kompaktes Framework, welches beliebige Mehrzieloptimierungsprobleme effektiv unterstützt. Das realisierte Framework zeichnet sich durch folgende Eigenschaften aus:

- ein hierarchischer Aufbau, der die natürliche Struktur von EAs wiederspiegelt;
- allgemein gehaltene Schnittstellen und eine offene Architektur, damit eine Vielzahl von EAs darauf aufbauen können – die Mehrzieloptimierungsverfahren von [Zitzler und Thiele 1998] und [Fonseca und Fleming 1993] sind implementiert;
- Unterstützung beliebig strukturierter Chromosomen – lineare Chromosomen mit diversen Mutations- und Kreuzungsverfahren sind implementiert;
- Individuen, die auch aus mehreren Chromosomen bestehen können und sich insbesondere für Mehrzieloptimierung eignen;
- Populationen variabler Grösse, welche mittels Clustering reduzierbar sind und auch als Matingpools und Paretomengen fungieren können.

Dieser Bericht gliedert sich wie folgt: Die für das Verständnis des praktischen Teils notwendigen Fachbegriffe und Konzepte werden in den Kapiteln 2 und 3 eingeführt. Kapitel 2 befasst sich in erster Linie mit genetischen Algorithmen und

Kapitel 3 mit der Mehrzieloptimierung im allgemeinen und den oben erwähnten Verfahren im speziellen. Kapitel 4 stellt drei bestehende EA-Bibliotheken (*libraries*) und -Gerüste kurz vor. Das Ziel war einerseits zu erfahren, was im Bereich der Mehrzieloptimierung schon verfügbar ist und andererseits einen Überblick über Designmöglichkeiten und -grenzen zu gewinnen. Kapitel 5 beschreibt den Aufbau des Frameworks und illustriert seine Benutzung anhand eines verallgemeinerten *Travelling-Salesman*-Problems. Ein Vergleich der implementierten Mehrzieloptimierungsverfahren (ebenfalls anhand des verallgemeinerten *Travelling-Salesman*-Problems) ist Inhalt von Kapitel 6. Kapitel 7 schliesslich diskutiert, ob und wie die Anforderungen der im Anhang A abgedruckten Aufgabenstellung erfüllt wurden und zeigt mögliche Weiterentwicklungsrichtungen auf.

Zürich im Februar 1999

Paul E. Sevinç, 7. Semester IIIB

2. Grundlagen

Genetische Algorithmen sind eine Klasse *evolutionärer Algorithmen*; weitere Klassen sind die *genetische Programmierung*, *Evolutionsstrategien* und die *evolutionäre Programmierung* [Nissen 1997]. Diese vier Klassen können allerdings nicht strikt unterschieden werden – insbesondere definieren sie nicht alle Autoren gleich. Über das in §2.2. erläuterte Grundkonzept genetischer Algorithmen besteht jedoch weitgehend Einigkeit.

Wie es ihr Name schon andeutet, orientieren sich evolutionäre Algorithmen an der Biologie, vor allem Darwins Evolutionstheorie (Stichwort: *survival of the fittest*) und der klassischen Genetik. Viele Fachbegriffe evolutionärer Algorithmen entstammen deshalb der Biologie und werden analog verwendet; §2.1. vermittelt eine Übersicht.

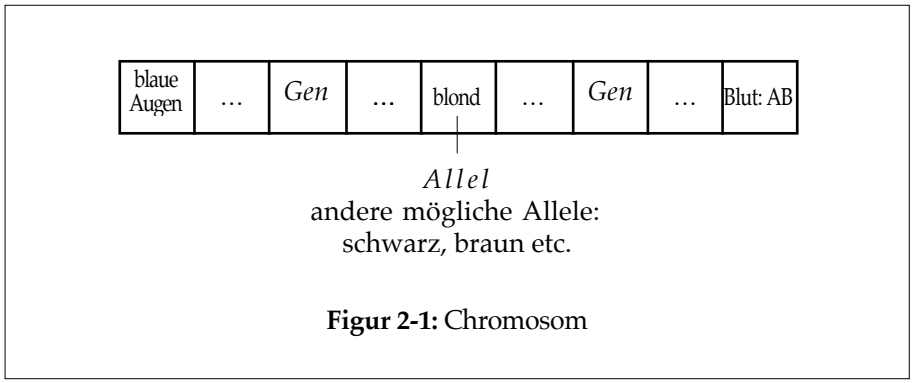
Anhand eines trivialen Problems (die Nullstelle einer linearen Funktion zu finden) wird als Abschluss dieses Kapitel in §2.3. die Arbeitsweise eines einfachen genetischen Algorithmus illustriert.

2.1. Wichtige Fachbegriffe

Die folgenden Definitionen basieren auf [Miram und Scharf 1988] (Biologie) und [Haupt und Haupt 1998], [Mitchell 1996] sowie [Nissen 1997] (evolutionäre Algorithmen).

Evolutionäre Algorithmen sind seit Jahren bekannt, und sie wurden und werden erfolgreich zur Lösung verschiedenster Optimierungsaufgaben eingesetzt. Dies gilt in besonderem Masse für genetische Algorithmen. Charakteristisch für GAs ist, dass sie jeweils auf einer Menge von Lösungskandidaten (wenige Dutzend bis mehrere Hundert) operieren. Ein Lösungskandidat wird dabei als *Individuum* bezeichnet, eine Menge als *Population*. Der Grundgedanke ist, aus (guten) Lösungen durch Rekombination und/oder Modifikation bessere Lösungen zu erhalten (Nachkommen, die die guten Eigenschaften ihrer Vorfahren erben und/oder verbessern).

In der Biologie ist ein *Gen* eine Erbanlage, die bestimmte Strukturen oder Funktionen eines Organismus kodiert und in den Zellkernen eines Individuums enthalten ist. Die genaue Ausprägung eines Gens wird *Allel* genannt. Die Gesamtheit aller Gene im Zellkern eines Individuums heisst *Genom*. Man spricht auch vom Erbbild oder *Genotyp*. Einzelne Gene können, wie im Beispiel der Blutgruppen des AB0-Systems, stets zur Ausbildung eines Merkmals führen. Insgesamt legt aber der Genotyp lediglich die Reaktionsnorm fest. Sie ist der Rahmen, innerhalb dessen sich das Erscheinungsbild, der *Phänotyp*, im Zusammenspiel von Genom und Umwelteinflüssen (Erziehung, Ernährung, Ausbildung etc.) herausbildet. *Chromosomen* sind Strukturen, auf denen Gene angeordnet sind (Figur 2-1).

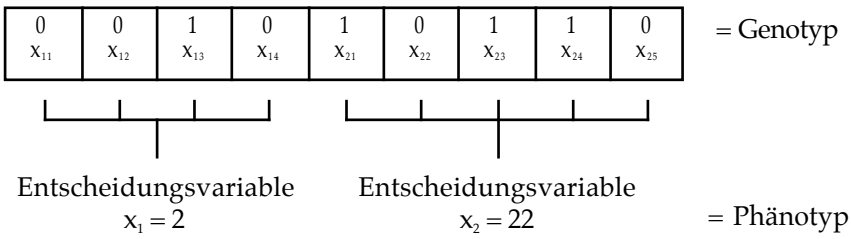


Figur 2-1: Chromosom

GA-Individuen sind natürlich nicht annähernd so komplex aufgebaut wie Lebewesen, sondern bestehen einfach aus *Chromosomen* und Chromosomen ihrerseits aus *Genen*. (Chromosomen sind typischerweise Arrays von Genen und Individuen Arrays von Chromosomen, wobei nicht selten ein Individuum aus einem einzigen Chromosom besteht.) Ein Gen kann durch ein einzelnes Bit modelliert werden oder auch von einem bestimmten Typ sein, z.B. eine Fliesskommazahl oder ein Buchstabe. Als *Allele* werden die Werte bezeichnet, die ein Gen annehmen kann (im Falle des Bits z.B. 0 und 1). Insbesondere bei binär kodierten Genen kann eine ganze Gruppe von Genen für eine *Entscheidungsvariable*, ein Argument der zu optimierenden Funktion, stehen (Figur 2-2).

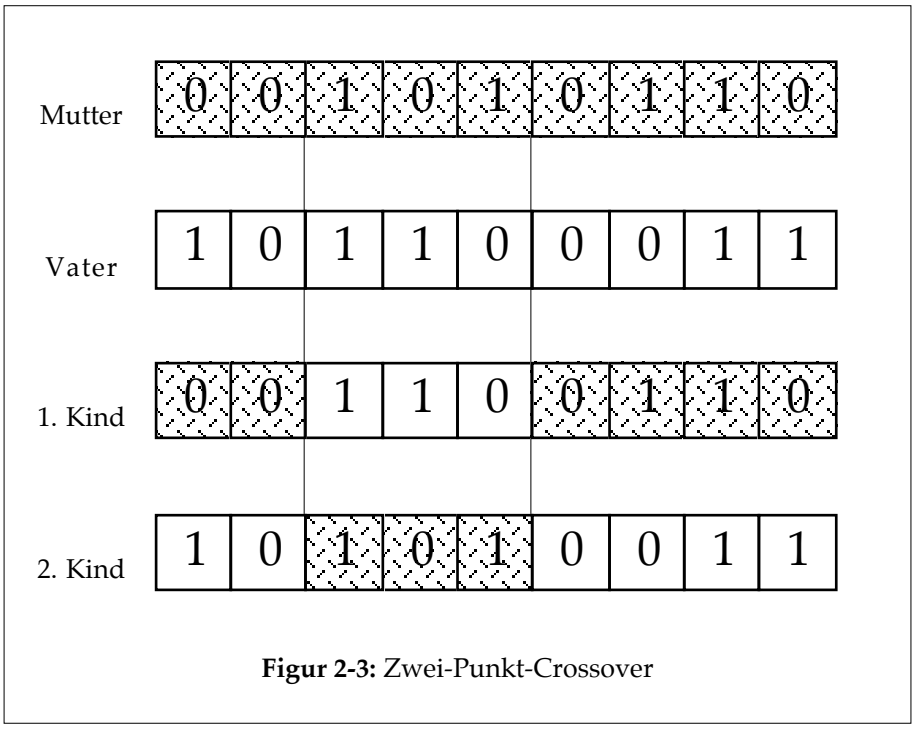
Der *Genotyp* ist das Wertetupel der Gene (in Figur 2-2 <0, 0, 1, 0, 1, 0, 1, 1, 0>) und stellt die Kodierung einer Problemlösung dar, der *Phänotyp*, also die

tatsächliche Lösung, wird zumeist durch das Wertetupel der Entscheidungsvariablen (in Figur 2-2 [binäre Kodierung] <2, 22>) beschrieben.



Figur 2-2: Ein-Chromosom-Individuum, binär codiert

Individuen, die das Resultat geschlechtlicher Fortpflanzung sind, enthalten sowohl Erbinformation der Mutter als auch des Vaters, d.h. Chromosomen treten jeweils paarweise auf, wobei i.a. nur eines der beiden sich entsprechenden Gene zur Geltung kommt. Wenn diese Individuen nun ihrerseits Keimzellen bilden, dann enthalten letztere jeweils einen Paarling aller Chromosomenpaare, manchmal aber auch auch eine Kombination davon, welche durch *crossing over* entstanden ist.



Figur 2-3: Zwei-Punkt-Crossover

Bei GAs ist nun Crossing-Over oder *Crossover* (Kreuzung) der Haupt-

mechanismus zur Erzeugung neuer Lösungskandidaten. Im Fall eines Elternpaars (bei GAs kann ein Nachkomme auch nur von einem oder mehr als zwei Vorfahren abstammen) würde ein Teil der Kindergene das Allel der Mutter übernehmen, der andere Teil dasjenige des Vaters (Figur 2-3).

Bei Nachkommen liegt nicht selten eine Erbgutveränderung oder *Mutation* vor. Man unterscheidet verschiedene Mutationstypen, je nachdem, ob die Anzahl der Chromosomen des Satzes (*Ploidiemutation*), die Struktur eines einzelnes Chromosoms (*Chromosomenmutation*) oder nur ein Gen (*Genmutation*) betroffen ist. Die Genmutation beruht auf einer chemischen Veränderung der Erbsubstanz, ohne dass eine sichtbare Strukturveränderung des Chromosoms vorliegt. Mutationen treten sprunghaft auf und sind ungerichtet. Ihre Häufigkeit wird durch die *Mutationsrate* ausgedrückt. Sie ist für ein einzelnes Gen sehr klein (zwischen 10^{-4} und 10^{-6}). Da aber die Zahl der Gene bei Vielzellern hoch ist (10^5 bis 10^6), gibt es insgesamt doch relativ viele Mutationen. Beim Menschen dürften 10% bis 40% aller Keimzellen eine Mutation aufweisen.

Die Genmutation wird auch in GAs verwendet. Die Bedeutung der Mutation liegt vor allem im Ermöglichen von Lösungen, die durch Kreuzung allein nicht auftreten würden (siehe §2.2.).

2.2. GA-Skelett

Beginnend mit einer (zufällig initialisierten) Ausgangspopulation erzeugt ein genetischer Algorithmus Folgepopulationen (nächste Generationen), bis darin eine genug gute Lösung gefunden wird oder die zur Verfügung stehende Zeit verstrichen ist. Von einer Generation zur nächsten werden dabei Individuen selektiert, gekreuzt und mutiert. Ein GA lässt sich damit in folgende Schritte einteilen:

- Initialisierung
- Selektion
- Kreuzung
- Mutation

• *Initialisierung*

Einmal, ganz zu Beginn der Optimierung, muss der GA initialisiert werden. Einerseits wird dabei eine Ausgangspopulation mit zufällig gesetzten Allelen erzeugt, andererseits werden wichtige Parameter bestimmt: minimale und maximale Grösse der Populationen (oftmals identische Werte, also konstante Populationsgrössen), Anzahl Generationen oder angestrebte Güte, Crossover- und Mutationswahrscheinlichkeiten (s.u.) etc.

• *Selektion*

Die Selektion läuft in zwei Schritten ab:
Zunächst wird die *Fitness* jedes Individuums bestimmt. Die Fitness ist Ausdruck der Güte eines Lösungskandidaten. Die Fitnessfunktion hängt natürlich vom zu optimierenden Problem ab, aber auch vom benutzten Fitnesszuordnungsverfahren.
Anschliessend werden diejenigen Individuen ausgewählt, deren Erbinformation weiterbestehen wird: entweder weil sie einfach in die nächste Generation übernommen werden oder weil sie sich mit anderen Individuen paaren dürfen. Für Individuen mit besserer Fitness ist die Wahrscheinlichkeit ausgewählt zu werden höher. Ausserdem handelt es sich i.a. um "Selektion mit Zurücklegen", d.h. bestimmte Individuen werden mehrfach ausgewählt.

• *Kreuzung*

Die bei der Selektion zu Eltern bestimmten Individuen wurden im sogenannten *mating pool* abgelegt. Dort werden nun "ohne Zurücklegen" typischerweise Paare gebildet, die durch Crossover (zwei) Kinder erzeugen: Beim Crossover werden Teilsegmente mehrerer Chromosomen zu neuen korrekt aufgebauten Chromosomen kombiniert. Wenn z.B. **ABCDEFG** und **abcdefg** je ein Individuum darstellen und 2-Punkt-Crossover verwendet wird (Figur 2-3), dann könnten **ABcdefG** und **abCDEFg** die beiden Nachkommen sein. Dabei wurden zufällig 2 und 6 als Schnittpunkte ausgewählt und die Gene dazwischen ausgetauscht. Vor allem wenn nicht generell Individuen unverändert in die nächste Generation übernommen werden, findet Crossover nur mit bestimmter Wahrscheinlichkeit statt; gewisse Kinder und Eltern sind dann identisch.

• *Mutation*
Ist die nächste Generation nun erzeugt, wird jedes Gen mit kleiner Wahrscheinlichkeit mutiert: ein Bit wird invertiert, ein Buchstabe zufällig durch einen anderen ersetzt etc. Wenn gewisse Allele eines Gens in der Population nicht (mehr) vorkommen –und die Kreuzung damit nicht den gesamten Lösungsraum abdecken kann– können sie durch Mutation wieder reaktiviert werden.

Nachdem Selektion, Kreuzung und Mutation durchgeführt wurden, ist eine Generation vollzogen, eine neue Population entstanden. Sind die Lösungen noch nicht zufriedenstellend, und steht noch Zeit zur Verfügung, dann wird die Iteration beim Selektionsschritt wieder aufgenommen.

[Nissen 1997] beschreibt eine Vielzahl von Ersetzungsstrategien sowie Crossover- und Mutationsoperatoren. Auch werden Angaben für die Wahl der Wahrscheinlichkeiten und zur Implementation der Auswahlverfahren gemacht. Die im Framework implementierten Algorithmen werden in §3.3. erläutert.

2.3. Ein einfaches Beispiel[1]

Gegeben sei die Funktion $f(x)=|x-4|$, und gesucht sei ihre Nullstelle. Als Fitnessfunktion wählen wir einfach $f(x)$; je kleiner die Fitness, umso besser die Lösung. Die Lösungskandidaten kodieren wir als Bitstrings der Länge 5, so dass z.B. $01011_2=11_{10}$.

Die Ausgangspopulation bestehe aus folgenden sechs Individuen:
$01101_2=13_{10}$, $00110_2=6_{10}$, $00111_2=7_{10}$, $10011_2=19_{10}$, $00010_2=2_{10}$, $10100_2=20_{10}$.
Zunächst überprüfen wir, ob eines der Individuen die gesuchte Nullstelle liefert: $f(13)=9$, $f(6)=2$, $f(7)=3$, $f(19)=15$ $f(2)=2$, $f(20)=16$.

Da dies nicht der Fall ist, müssen nun die Selektionswahrscheinlichkeiten der Individuen bestimmt werden: Den Individuen mit kleinerer Fitness

[1] Die Population wurde bewusst klein gewählt, und als "Zufallsgenerator" fungierte der Autor.

wollen wir bessere Chancen einräumen, gewählt zu werden. Deshalb wird zunächst jedem Individuum der Kehrwert seiner Fitness zugeordnet. Dann wird dieser Wert skaliert, damit die Summe der Wahrscheinlichkeiten 1 ergibt. Individuum x_i wird damit mit Wahrscheinlichkeit $p_i = \frac{\frac{1}{f(x_i)}}{\sum_{j=1}^{6} \frac{1}{f(x_j)}}$ selektiert. Und im Schnitt hat Individuum x_i bei n Selektionen $p_i \cdot n$ Nachkommen.

In unserem Beispiel wird 01101 einmal gewählt (Erwartung: $\frac{0.11}{1.56} \cdot 6 = 0.4$), 00110 einmal ($\frac{0.50}{1.56} \cdot 6 = 1.9$), 00111 zweimal ($\frac{0.33}{1.56} \cdot 6 = 1.3$), 10011 keinmal ($\frac{0.06}{1.56} \cdot 6 = 0.2$), 00010 zweimal ($\frac{0.50}{1.56} \cdot 6 = 1.9$) und 10100 keinmal ($\frac{0.06}{1.56} \cdot 6 = 0.2$).

Nun werden zufällig Zweierpaare gebildet und dann ein Crossover-Punkt bestimmt; das erste Paar wird nach dem zweiten Bit gekreuzt, das zweite Paar nach dem dritten und das dritte Paar überhaupt nicht:

Eltern-	01|101	001|10	00111
paar	00|111	000|10	00010
1. Kind	01 111	001 10	00111
2. Kind	00 101	000 10	00010

Mutiert werden noch 01111 zu 11111 und 00010 zu 00011, so dass die neue Population schliesslich aus folgenden Individuen besteht:
$11111_2=31_{10}$, $00101_2=5_{10}$, $00110_2=6_{10}$, $00010_2=2_{10}$, $00111_2=7_{10}$ und $00011_2=3_{10}$.

Auch wenn noch kein Individuum die gesuchte Nullstelle repräsentiert, so hat sich doch die Gesamtfitness schon von 47 auf 36 hin verbessert.

3. Mehrzieloptimierung

3.1. Einziel- vs. Mehrzieloptimierung

In §2. wurde vereinfachend davon ausgegangen, dass eine *skalare* Funktion optimiert wird (z.B. kürzester Reiseweg für einen Vertreter). Man spricht dann von *Einzieloptimierung*.

Aber manchmal wollen mehrere Kriterien bedacht werden, die sich nicht einfach gewichten und in einem einzigen "Überkriterium" zusammenfassen lassen. Beim Kauf eines Notebooks beispielsweise sind sicher das Gewicht, die Grösse des Bildschirms, die Arbeitsleistung, die Lebenszeit der Batterie und der Preis einige der Kriterien. Und einzelne Kriterien stehen dabei im Konflikt miteinander; ein schneller Prozessor z.B. treibt den Preis in die Höhe. Dies ist ein typisches Beispiel für *Mehrzieloptimierung*.

3.2. Definition eines Mehrzieloptimierungsproblems

Die nun folgenden Fachbegriffe ergänzen diejenigen aus §2.1. und basieren auf [Zitzler und Thiele 1998].

In einem Mehrzieloptimierungsproblem soll eine *vektorielle* Funktion optimiert werden, d.h. eine Funktion, welche ein Tupel von Entscheidungsvariablen (wie bisher) auf ein Tupel von Zielwerten (neu) abbildet:
optimiere $f\colon X_1 \times \dots X_n \to Y_1 \times \dots Y_m,\ \vec{x} \mapsto f(\vec{x})$

$$\vec{x} = (x_1, \dots x_n),\ f(\vec{x}) = (f_1(\vec{x}), \dots f_m(\vec{x})) = (y_1, \dots y_m) = \vec{y}\,.$$

$\vec{x}$ wird *Entscheidungsvektor* genannt und $\vec{y}$ *Zielvektor*.

Die Lösungs*menge* eines Mehrzieloptimierungsproblems besteht aus sämtlichen Entscheidungsvektoren, welche nicht in einem Kriterium verbessert werden können, ohne ein anderes zu verschlechtern. (Notebook-Beispiel: Wir wollen einen grösseren Bildschirm, doch dieser ist teurer und verkürzt die Lebenszeit der Batterie.) Diese Vektoren werden als *Pareto-optimal* bezeichnet.

Mathematisch formuliert:
Ohne Einschränkung der Allgemeinheit sollen ein Maximierungsproblem sowie zwei Entscheidungsvektoren $\vec{a}$ und $\vec{b}$ betrachtet werden:

$\vec{a}$ *dominiert* $\vec{b}$ ($\vec{a} \succ \vec{b}$) genau dann, wenn

$\forall i \in \{1,\ldots m\}: \; f_i(\vec{a}) \geq f_i(\vec{b}) \; \wedge \; \exists j \in \{1,\ldots m\}: \; f_j(\vec{a}) > f_j(\vec{b})$.

Alle Entscheidungsvektoren einer Menge, welche von keinem Vektor dieser Menge dominiert werden, werden *nichtdominiert* genannt. Die Menge aller nichtdominierten Lösungen des gesamten Suchraums ist die sogenannte *Pareto-optimale Menge* (kurz *Paretomenge*) oder *Pareto-optimale Front*.

3.3. Zwei Verfahren

3.3.1. FFGA

Der erste Mehrzieloptimierungs-GA, den wir betrachten, wurde entwickelt von *Carlos M. Fonseca* und *Peter J. Fleming* [Fonseca und Fleming 1993]. Seine Entwickler gaben ihm kein Akronym, wir nennen ihn deshalb einfach *FFGA*.

Gegeben ist eine Population der t-ten Generation und gesucht ist die Population der (t+1)-ten Generation.

- *Rangzuordnung und Sortierung*

Zunächst wird jedem Individuum in der Population ein *Rang* zugeordnet. Und zwar ist $1 + p$ der Rang eines Individuums, welches von p Individuen der aktuellen Population dominiert wird. (Daraus folgt, dass alle nichtdominierten Individuen Rang 1 haben.) Dann wird die Population nach Rang in aufsteigender Reihenfolge sortiert.

- *Fitnesszuordnung und Fitnessaufteilug*

Die Individuen erhalten nun durch Interpolation vom ersten Individuum bis zum letzten eine Fitness. Welches Fitness-zuordnungsverfahren zu gebrauchen ist, wird in [Fonseca und Fleming 1993] nicht spezifiziert. Wir verwenden *exponential ranking*

[Nissen 1997, S. 70][2].
Nach dem Exponential-Ranking haben Individuen mit demselben Rang nicht dieselbe Fitness. Deshalb wird von solchen Individuen jeweils rangweise der Durchschnitt der Fitnesswerte berechnet und den betreffenden Individuen zugeordnet. Die Gesamtfitness über die ganze Population ändert sich dadurch offensichtlich nicht.
Im Anschluss daran wird *fitness sharing* durchgeführt [Nissen 1997, S. 76-77], um die Diversität in der Population aufrecht zu erhalten.

- *Füllen des Matingpools*

Nach der Fitnesszuordung werden die Lösungskandidaten unter Verwendung von *stochastic universal sampling* [Nissen 1997, S. 65] in den Matingpool kopiert. Dazu müssen die Fitnesswerte zuvor noch mit der Grösse des Matingpools skaliert werden.

- *Erzeugen der nächsten Generation*

Schliesslich werden im Matingpool zufällig Elternpaare gebildet, diese (mit "grosser" Wahrscheinlichkeit) miteinander gekreuzt[3] und ihre Kinder in die Folgepopulation gelegt. Die Kinder werden zuallerletzt noch (mit "kleiner" Wahrscheinlichkeit) mutiert.

3.3.2. SPEA

Der zweite hier betrachtete Mehrzieloptimierungs-GA heisst *SPEA*, *Strength Pareto Evolutionary Algorithm*, und wurde entwickelt von *Eckart Zitzler* und *Lothar Thiele* [Zitzler und Thiele 1998].

SPEA nutzt ein Konzept genannt *Elitismus*. D.h., dass die besten in vorherigen Generationen gefundenen Individuen immer erhalten bleiben, indem sie z.B. von einer Generation zur nächsten einfach übernommen oder in einer "Elitepopulation" gespeichert werden. Auch bei SPEA überleben solche Individuen in einer separaten, in der

[2] Mündlicher Hinweis des Betreuers: Fonseca und Fleming verwenden in anderen Papers ebenfalls *exponential ranking*.
[3] Die Eltern werden einfach geklont, wenn sie nicht gekreuzt werden.

Grösse jedoch beschränkten Population, welche Paretomenge[4] genannt wird und sich direkt auf die Fitness der Population auswirkt.

Gegeben ist ebenfalls eine Population der t-ten Generation sowie eine Paretomenge, welche zu Beginn leer sein kann. Gesucht ist die Population der (t+1)-ten Generation sowie die aktualisierte Paretomenge.

- *Aktualisieren der Paretomenge*

Zunächst werden sämtliche nichtdominierten Lösungen der Population in die Paretomenge kopiert und dann diejenigen Individuen aus der Paretomenge ausgeschlossen, welche dominiert sind.
Überschreitet die Paretomenge eine gewisse festgelegte Grösse, muss sie mittels *Clustering* reduziert werden, ohne ihre Charakteristik zu zerstören. Der Clustering-Algorithmus ist in Listing 3-1 abgebildet. *ClusterDistance(X, Y)* ist dabei die Summe der Abstände sämtlicher Paare (x, y) mit x aus X und y aus Y geteilt durch die Anzahl solcher Paare, und *GetCentroid(cluster)* retourniert dasjenige Individuum aus *cluster*, dessen Summe der Abstände zu den anderen Individuen in *cluster* minimal ist.

- *Fitnesszuordnung*

Die Fitnesszuordnung muss sowohl für die Paretomenge als auch für die Population erfolgen.
Die Fitness eines Individuums der Paretomenge ist die Anzahl der Individuen der Population, welche von diesem dominiert werden geteilt durch 1 plus die Mächtigkeit der Population.
Die Fitness eines Individuums der Population ist 1 plus die Summe der Fitnesswerte der Individuen der Paretomenge, welche dieses dominieren.

- *Füllen des Matingpools*

Aus der Vereinigung von Population und Paretomenge werden jeweils zufällig zwei Individuen gezogen und mittels *binary tournament selection* bestimmt, wer in den Matingpool kopiert wird;

[4] Streng genommen handelt es sich um eine Menge nichtdominierter Individuen und nicht um *die* Paretomenge.

d.h. dasjenige Individuum wird kopiert, dessen Fitness besser ist. (Man beachte, dass wenn ein Individuum aus der Population stammt und eines aus der Paretomenge, immer letzteres gewinnt.)

```
PROCEDURE ReduceParetoSet
IN/OUT:
      paretoSet;
BEGIN
      (* initialization: each Pareto point forms a cluster *)
      clusterSet := {};
      FOR paretoInd IN paretoSet DO
            clusterSet := clusterSet ∪ { {paretoInd} };
      OD
      (* join clusters until number of clusters remains under maximum *)
      WHILE |clusterSet| > maxParetoPoints DO
            (* select two clusters which have minimum distance *)
            minDistance := ∞;
            FOR {X, Y} SUBSET OF clusterSet DO
                  IF ClusterDistance(X, Y) < minDistance THEN
                        cluster1 := X;
                        cluster2 := Y;
                        minDistance :=
                         ClusterDistance(cluster1, cluster2);
                  FI
            OD
            (* join the two selected clusters *)
            newCluster := cluster1 ∪ cluster2;
            clusterSet := clusterSet \ {cluster1, cluster2};
            clusterSet := clusterSet ∪ {newCluster};
      OD
      (* for each cluster pick out representative solution (centroid) *)
      paretoSet := {};
      FOR cluster IN clusterSet DO
            paretonInd := GetCentroid(cluster);
            paretoSet := paretoSet ∪ {paretoInd};
      OD
END
```

Listing 3-1: Average Linkage Clustering (aus [Zitzler und Thiele 1998, S. 15])

- *Erzeugen der nächsten Generation*

Dieser letzte Schritt ist identisch demjenigen bei FFGA.

4. Bestehende Bibliotheken

Die bis heute veröffentlichten Varianten genetischer Algorithmen und vor allem ihre problemspezifischen Implementierungen unterscheiden sich natürlich im Detail. Dennoch lassen sich grundsätzliche Gemeinsamkeiten feststellen. Und in welchem Gebiet der Informatik auch immer unterschiedliche Applikationen Gemeinsamkeiten aufwiesen, wurde versucht, diese herauszuziehen und durch Prozeduren, abstrakte Datentypen, Klassen etc. in Bibliotheken oder als Frameworks wiederverwendbar zu machen.

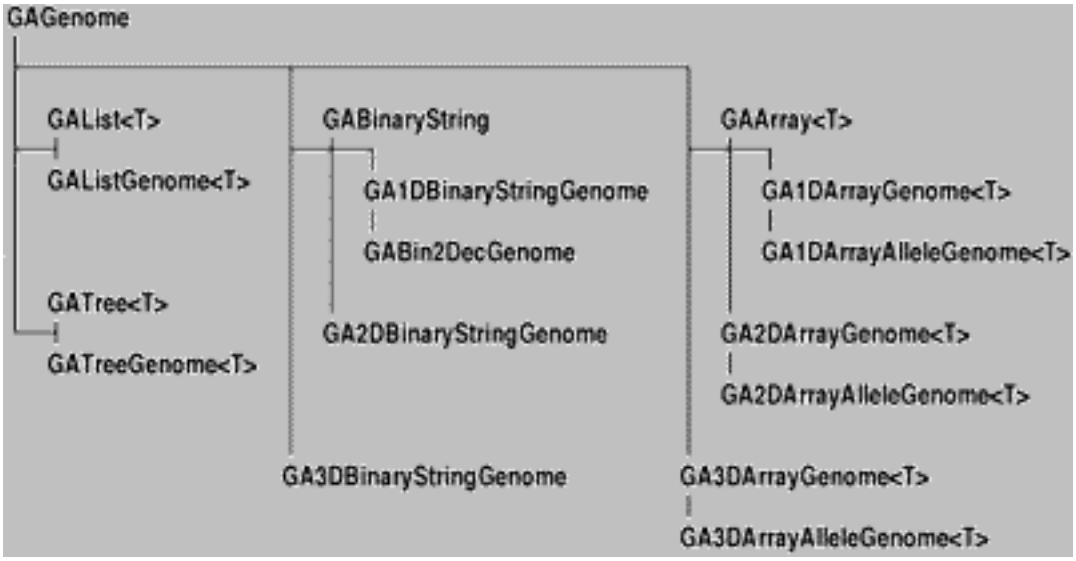
Teil der Analysephase dieser Semesterarbeit war eine knappe Untersuchung bestehender Bibliotheken und Frameworks im Bereich evolutionärer Algorithmen (siehe [URL 4-1] für eine Auswahl). Besonders interessiert wären wir natürlich an Frameworks für die Mehrzieloptimierung mit genetischen Algorithmen gewesen, doch existieren solche unseres Wissens nicht. (Vielleicht unterstützt aber die GA-MATLAB-Toolbox von *Andrew Chipperfield*, *Carlos Fonseca*, *Peter Fleming* und *Hartmut Pohlheim* [URL 4-2] Mehrzieloptimierung, sie stand uns jedoch nicht zur Verfügung.)

Den meisten Bibliotheken gemeinsam ist die Tatsache, dass sie in C implementiert sind. – Trotzdem sind in gewissen Fällen objektorientierte Ansätze vorhanden (via Prozedurvariablen/Zeiger auf Funktionen). YAGPLIC ist eine solche Bibliothek und wird in §4.3. vorgestellt. In Java scheinen einzig einige Demonstrationsprogramme entwickelt worden zu sein (z.B. *The GA Playground* [URL 4-3]). Die klar dominierende objektorientierte Sprache für evolutionäre Algorithmen ist C++. Auch die in §4.1. und §4.2. präsentierten GAlib und EO wurden damit implementiert.

4.1. GAlib

GAlib ist eine Klassenbibliothek, die eine vielfältige Sammlung von Bausteinen zur Verfügung stellt, welche problemspezifisch zusammengestellt und –wo notwendig– modifiziert resp. erweitert werden können. Dabei macht GAlib auch Gebrauch von C++-typischen Konstrukten und Techniken wie Templates und mehrfacher Vererbung.

Lösungskandidaten sind Instanzen einer Unterklasse von *GAGenome* und bestehen i.a. aus einem einzigen Chromosom (d.h. die vordefinierten Funktionen gehen davon aus). Diverse Unterklassen sind bereits Bestandteil der Bibliothek, z.B. für Individuen in Form von Binärstrings oder Bäumen (Figur 4-1).

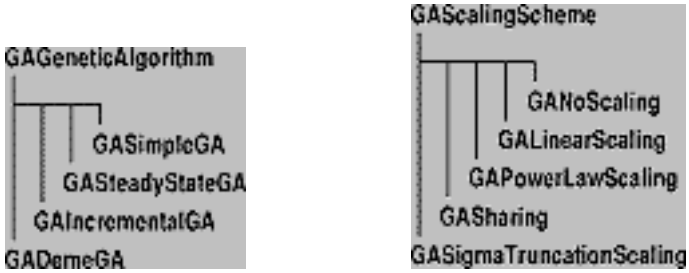


Figur 4-1: Hierarchie von *GAGenome* (Quelle: [URL 4-4])

Bevor ein genetischer Algorithmus aufgerufen wird, muss die Bibliotheksbenutzerin im Sinne eines Prototyps die Struktur des auf ihr Optimierungsproblem angepassten *GAGenome*s definieren. Diese "Prototypinstanz" wird dann der GA-Klasse (siehe unten) übergeben und von ihr geklont (und die "Kloninstanz" zufällig initialisiert), um so die Ausgangspopulation zu erhalten.

Da nur die Benutzerin die konkrete Struktur der Lösungskandidaten und ihre Bedeutung kennt, muss auch sie die Zielfunktion definieren. GAlib schreibt einzig die Signatur dafür vor: *float Objective(GAGenome&);* Man beachte, dass erstens ein Skalar zurückgegeben wird und zweitens die Zielfunktion nicht mit einer Dimension indiziert werden kann, GAlib also für Einzieloptimierung konzipiert ist. Anhand des Rückgabewerts berechnet dann der GA die Fitness.

Um mit verschiedenen GAs arbeiten zu können, sind deren Gemeinsamkeiten in der Basisklasse *GAGeneticAlgorithm* zusammengefasst, von der vier Unterklassen zu GAlib gehören. *GAGeneticAlgorithm* definiert jedoch nicht nur die Optimierung an sich, sondern stellt u.a. auch Methoden für statistische Zwecke zur Verfügung. Genügen der GAlib-Benutzerin die angebotenen GAs nicht, wird sie durch GAlib auch darin unterstützt, weitere *GAGeneticAlgorithm*-Unterklassen zu realisieren, da mehrere Hilfsklassen, z.B. für die Selektion (*GASelectionScheme*) und die Fitnessskalierung (*GAScalingScheme*), zur Bibliothek gehören (Figur 4-2).



Figur 4-2: *GAGeneticAlgorithm* und *GAScalingScheme* (Quelle: [URL 4-4])

Für weitere Informationen (inkl. Quellcode und Beispiele) sei auf die GAlib-Homepage verwiesen [URL 4-5].

4.2. EO

EO, das *Evolutionary Computation Framework*, ist allgemeiner ausgelegt als GAlib, indem über GAs hinaus noch weitere EAs unterstützt werden, doch der Schwerpunkt liegt klar bei ersteren.

Individuen bestehen auch hier nur aus einem Chromosom, und die Optimierung erfolgt gemäss einem Ziel. Ungleich GAlib ist die Funktionalität aber auf viele kleinere Klassen verteilt, welche zwar die Flexibilität erhöhen, aber auch die Komplexität.

EOs Quellcode und Dokumentation sind ebenfalls frei erhältlich [URL 4-6].

4.3. YAGPLIC

YAGPLIC steht für *Yet Another Genetic Programming Library In C*, ist also für genetische Programmierung konzipiert und nicht genetische Algorithmen.

Die Optimierungsschritte sind in der durch YAGPLIC vordefinierten Hauptfunktion festgelegt, welche bei Bedarf problemspezifische Funktionen aufruft. Mehr Flexibilität bei der Optimierung ist nur mit geringem Aufwand leider nicht möglich.

Innerhalb einer –ebenfalls vorgegebenen– Initialisierungsfunktion werden die Struktur der Individuen definiert sowie gewisse Einstellungen, wie Populationsgrösse, Abbruchbedingung etc., vorgenommen. Dieser Initialisierungsfunktion muss auch eine Instanz einer Datenstruktur zur Beschreibung des Problems übergeben werden.

Die Fitness eines Individuums wird von einer separaten Funktion berechnet. In diesem Zusammenhang ist zu erwähnen, dass YAGPLIC generell als Funktionsminimierer arbeitet, also für bessere Individuen kleinere Fitnesswerte erwartet.

Viele genetische Operatoren sind Bestandteil von YAGPLIC: Der Benutzer kann jeweils aus mehreren Funktionen wählen für die Reproduktion, die Kreuzung, die Mutation u.s.w. Zahlreiche weitere Funktionen erlauben eine etwas feinere Abstimmung und Auswertung (Histogramme, Statistiken) der Optimierung und sind inklusive Beispiele in [Blickle 1996] beschrieben.

5. FEMO

5.1. Vorbemerkungen

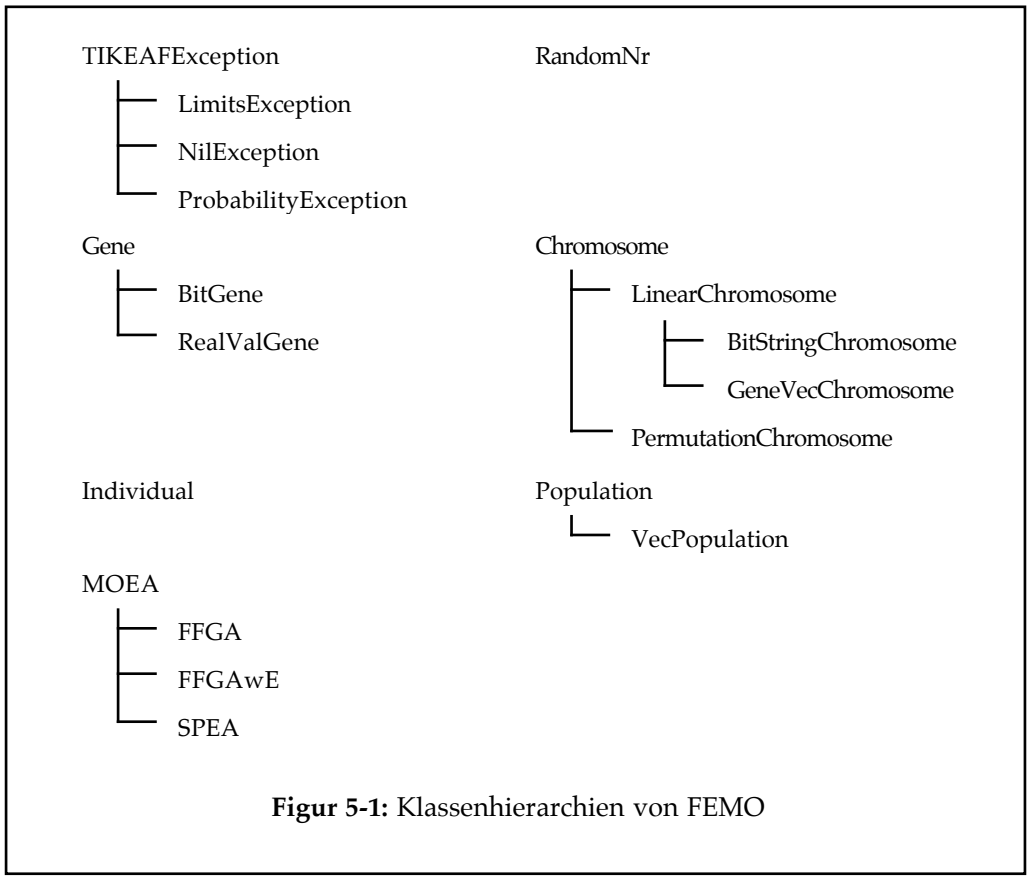
Wie der Aufgabenstellung in Anhang A zu entnehmen ist, sollte *FEMO*, das *Framework for Evolutionary Multiobjective Optimization*, welches aus dieser Semesterarbeit hervorging, ursprünglich in *Java* realisiert werden. Folgende Überlegungen zur Effizienz veranlassten uns jedoch noch zu Beginn der Arbeit, trotzdem *C++* als Implementationssprache vorzuziehen: Die heute verfügbaren *Just-In-Time*-Compiler, welche Java-Bytecode vor Ausführung eines Java-Programms in Maschinensprache übersetzen, erzeugen noch keinen Maschinencode, der es in Geschwindigkeit mit direkt aus C++-Quellcodes erzeugtem Maschinencode aufnehmen kann. Mit *HotSpot* [URL 5-1] verspricht *Sun Microsystems* zwar eine Java-Laufzeitumgebung, welche Bytecode vor seiner Ausführung nicht nur in Maschinensprache übersetzt, sondern den Maschinencode noch während der Abarbeitung optimiert (Java-Programme sollen dadurch etwa gleich schnell wie entsprechende C++-Programme sein), aber HotSpot ist noch nicht verfügbar, und Suns Erwartungen müssen deshalb mit Vorsicht genossen werden. Ausserdem sind C und C++ wie bereits in §4. angedeutet die verbreitesten Programmiersprachen in der EA-Gemeinde.

Trotz alldem wurde beim Entwurf und der Implementation von FEMO einer allfälligen Java-Portierung Rechnung getragen, sofern dies dem Verständnis des Frameworks und seiner Effizienz nicht abträglich war. Z.B. konnte problemlos auf Konstrukte wie Schablonen (*templates*) oder Techniken wie mehrfache Vererbung (*multiple inheritance*) verzichtet werden.

Zuletzt sei noch darauf hingewiesen, dass es sich beim verwendeten Compiler um *egcs* [URL 5-2] in der Version 1.1.1 handelt, welcher für diverse Plattformen frei erhältlich ist. Da bei der Implementation von FEMO aber ohnehin der C++-Standard [Stroustrup 1997] respektiert wurde, sollten sich mit heutigen und zukünftigen Compilern sowohl in Bezug auf die Wartung als auch die Portabilität keine nennenswerten Schwierigkeiten ergeben.

5.2. FEMO-Überblick

FEMO besteht aus sieben Gruppen von Klassen, welche im Detail in Anhang C beschrieben sind. Figur 5-1 zeigt sie in der Übersicht.



Figur 5-1: Klassenhierarchien von FEMO

Die simpelsten Klassen sind *TIKEAFException* und ihre drei Unterklassen, welche für die Ausnahmenbehandlung (*exception handling*), d.h. für die Meldung von Fehlern vom Framework zu dessen Benutzer, gebraucht werden.

Der Zufall spielt in EAs eine wichtige Rolle. Innerhalb von FEMO besorgen Instanzen von *RandomNr* die Generierung von Pseudozufallszahlen. Soll dafür ein anderer Algorithmus verwendet werden, kann die entsprechende *RandomNr*-Methode in einer Unterklasse problemlos überschrieben werden.

Individuen fungieren als potentielle Lösungen eines evolutionären Optimierers. Da erstere aber problemabhängig sind, definiert die abstrakte

Klasse *Individual* im wesentlichen nur ihre Schnittstelle. In einer vom Frameworkbenutzer zu definierenden Unterklasse erfolgt dann die eigentliche Problemkodierung. Diese wird sich im allgemeinen auf Chromosomen abstützen.

Chromosomen haben in der abstrakten Klasse *Chromosome* ihre gemeinsame Basis, unabhängig davon, ob sie nun linear, baumartig oder sonstwie organisiert sind. Die noch immer abstrakte Unterklasse *LinearChromosome* dient den linearen Chromosomen als Basisklasse, vor allem aber definiert sie Crossoveroperatoren, welche konkrete Unterklassen wie *BitStringChromosome* und *GeneVecChromosome* nutzen können. *GeneVecChromosome* ist in erster Linie dafür gedacht, mit Hilfe von *Gene* und ihren Unterklassen heterogene Chromosomen (Chromosomen mit verschieden kodierten Entscheidungsvariablen) zu unterstützen. *PermutationChromosome* eignet sich ausschliesslich für Permutationsprobleme, dafür sind gleich mehrere Crossover- und Mutationsoperatoren auf ihr definiert.

Die abstrakte Klasse *Population* enthält im Grunde genommen schon die gesamte Funktionalität ihrer Unterklassen (derzeit einzig *VecPopulation*). Letztere besorgen nur noch die eigentliche Speicherung der *Individual*-Zeiger. *(Vec)Populations* verwalten Individuen in verschiedenen Funktionen: als Population an sich, als Matingpool oder als Menge nichtdominierter Individuen.

Die drei Unterklassen von *MOEA* schliesslich übernehmen die Selektion, indem Individuen einer Population Fitnesswerte zugeordnet werden und dann ein Matingpool gefüllt wird.

5.3. FEMO-Tutorial

In diesem Abschnitt soll das Zusammenspiel einiger in §5.2. eingeführter und im Anhang C dokumentierter FEMO-Klassen anhand eines konkreten Mehrzieloptimierungsproblems illustriert werden. Die Beispielprogramme sind in allgemeinerer Form vollständig in Anhang B abgedruckt.

5.3.1. Das Problem

Ausgangspunkt jeder Optimierung ist eine adäquate Formulierung des gestellten Problems. Unseres lautet wie folgt:
Gegeben sind n (n > 2) Städte. Jede Stadt ist von allen anderen aus direkt erreichbar, und eine Verbindung zwischen zwei Städten ist durch zwei Zahlen charakterisiert, welche die Reisekosten und die Reisedauer beschreiben.
Gesucht ist eine Rundreiseroute so, dass jede Stadt genau einmal besucht wird und sowohl die Reisekosten als auch die Reisedauer minimiert werden.
Dies ist eine verallgemeinerte Version des klassischen *Travelling-Salesman*-Problems (TSP).

Offensichtlich handelt es sich beim TSP um ein Permutationsproblem, wobei es keine Rolle spielt, wo die Rundreise beginnt und ob sie "vorwärts" oder "rückwärts" durchgeführt wird; damit bleiben $\frac{n!}{n \cdot 2} = \frac{(n-1)!}{2}$ verschiedene Rundreisen.

Es ist auch ein Mehrzieloptimierungsproblem, denn wenn zwischen zwei Städten eine schnelle Verbindung besteht, bedeutet dies nicht gezwungenermassen, dass die Städte Nahe beieinander liegen, sondern vielleicht einfach nur, dass es sich um eine teuere Flugverbindung handelt.

5.3.2. Das Individuum

Lösungskandidaten sind Instanzen einer *Individual*-Unterklasse. Für das TSP müsste ihre Schnittstelle etwa wie in Listing 5-1 gezeigt aussehen.

Individual hat zwei Felder, die für *TSPInd* von Interesse sind:

randomNr und *nrOfObjectives*. Das erste ist eine Referenz auf einen Zufallsgenerator, das zweite eine Konstante, in der die Anzahl Ziele gespeichert sind. *TSPInd* merkt sich zusätzlich die Anzahl Städte (*nrOfTowns*) und macht Gebrauch von *PermutationChromosome*, einer Chromosomen-Klasse für Permutationsprobleme. Abgesehen von den Konstruktoren definiert *TSPInd* nur Methoden, welche in *Individual* abstrakt sind und überschrieben werden müssen.

```
class TSPInd : public Individual
{
      private:
            const size_t            nrOfTowns;
            PermutationChromosome*  chromosome;

            // private constructor
            TSPInd( RandomNr& rn, size_t not,
                        PermutationChromsome* pc );

      protected:
            void        subInitRandom();
            bool        subMutate();
            double      subGetObjective( size_t index );

      public:
                        TSPInd( RandomNr& rn, size_t not );
                        ~TSPInd();
            Individual* clone();
            double      distance( Individual* ind );
            bool        covers( Individual* ind );
            bool        dominates( Individual* ind );
            void        mateWith( Individual* ind,
                              vector<Individual*>& children );
};
```

Listing 5-1: Schnittstelle eines TSP-Individuums

Die Konstruktoren und die damit eng verbundene Methode *clone()* sind in Listing 5-2 abgedruckt. Beide Konstruktoren müssen zuerst denjenigen von Individual aufrufen, um *randomNr* und *nrOfObjectives* zu initialisieren. Der öffentliche Konstruktor alloziert dann ein Permutationschromosom der Grösse *nrOfTowns*. Der private erhält das Chromosom von *clone()*, nachdem dem Chromosom aufgetragen wurde, sich zu klonen.

```
void
TSPInd( RandomNr& rn, size_t not, PermutationChromsome* pc )
      : Individual( rn, 2 ), // 2 objectives
      nrOfTowns( not ), chromosome( pc )
{
}

void
TSPInd( RandomNr& rn, size_t not )
      : Individual( rn, 2 ), nrOfTowns( not )
{
      chromosome = new PermutationChromosome( rn, not );
}

Individual*
clone()
{
      PermutationChromosome*  pc = chromosome->clone();
      return new TSPInd( randomNr, nrOfTowns, pc );
}
```

Listing 5-2: Konstruktoren und *clone()*

Die Berechnung eines Zielwerts kann eine äusserst teuere Angelegenheit sein. *Individual* merkt sich deshalb die Zielwerte, welche bereits berechnet wurden und gibt einen *getObjective()*-Auftrag nur dann an *subGetObjective()* der Unterklasse weiter, wenn der betreffende Zielwert noch nicht bekannt ist. *initRandom()* andererseits ruft immer *subInitRandom()* auf, invalidiert jedoch alle zuvor berechneten Zielwerte. Auch *mutate()* ruft immer *subMutate()* auf, invalidiert jedoch nur, wenn die Mutation auch wirklich stattgefunden hat, d.h. genau dann, wenn *subMutate()*s Rückgabewert *true* ist (Listing 5-3).

Beim TSP muss jeder Zielwert minimiert werden. Für diesen Spezialfall und den umgekehrten (Maximierungsproblem) können *covers()* und *dominates()* einfach implementiert werden: *Individual* definiert entsprechende Methoden, denen die Aufgabe übertragen werden kann, indem man ihnen zusätzlich einzig mitteilt, ob es sich um ein Maximierungs- oder ein Minimierungsproblem handelt (Listing 5-4).

```
void
subInitRandom()
{
      chromosome->initRandom();
}

bool
subMutate()
{
      if ( randomNr.flipP( 0.1 ) ) // mutate with probability 0.1
      {
            chromosome->swapMutation();
            return true;
      }
      else
            return false;
}

double
subGetObjective( size_t index )
{
      double      objective = 0;

      if ( index == 0 ) // calculate cost
      {
            for ( size_t s = 0; s < nrOfTowns; ++s )
            {
                  int   t1 = chromosome->get( s ),
                        t2 =
                        chromosome->get( (s + 1) % nrOfTowns );

                  objective += costFromTo( t1, t2 );
            }
      }
      else              // calculate duration
            …

      return objective;
}
```

Listing 5-3: Die via Individual aufgerufenen Methoden

Zwei Methoden, *distance()* und *mateWith()*, müssen noch überschrieben werden. Erstere berechnet die Distanz zwischen zwei Indivi-

duen (ofmals die euklidsche im Zielraum). Letztere erzeugt aus einem Elternpaar Kinder (Listing 5-5).

```
bool
covers( Individual* ind )
{
      return covers( false, ind ); // true for max problem
}

bool
dominates( Individual* ind )
{
      return dominates( false, ind ); // true for max problem
}
```

Listing 5-4: *covers()* und *dominates()* für Mini- und Maximierungsprobleme

```
void
mateWith( Individual* ind, vector< Individual* >& children )
{
      if ( randomNr.flipP( 0.8 ) ) // mate with probability 0.8
      {
            PermutationChromosome*  daughter =
            new PermutationChromosome( randomNr, nrOfTowns );
            PermutationChromosome*  son =
            new PermutationChromosome( randomNr, nrOfTowns );
            TSPInd*     tspInd = (TSPInd*)( ind );

            PermutationChromosome::uniformOrderBasedCrossover(
                  randomNr, chromosome, tspInd->chromosome,
                  daughter, son );

            children.push_back(
                  new TSPInd(randomNr, nrOfTowns, daughter) );
            children.push_back(
                  new TSPInd(randomNr, nrOfTowns, son) );
      }
      else
      {
            children.push_back( clone() );
            children.push_back( ind->clone() );
      }
}
```

Listing 5-5: Kreuzen zweier Individuen

5.3.3. Die Optimierungsroutine

Ist das Individuum erst einmal definiert, ist die Lösungssuche eine nahezu triviale Angelegenheit. Schwierig ist höchstens, sinnvolle Werte für die Parameter, u.a. die Populationsgrösse und die Anzahl Generationen, zu bestimmen.

Die Mehrzieloptimierung kann natürlich in irgendeiner Unterroutine untergebracht werden. Ist erstere aber nicht Teil eines umfangreicheren Programms, bietet sich eigentlich die Hauptfunktion an.

Folgende Datentypen werden benötigt: ein Zufallsgenerator, ein Mehrzieloptimierungs-GA und eine Population. Typischerweise werden alle nichtdominierten Individuen ausserdem über alle Generationen hinweg in einer weitere Population gesichert.

Das Generieren der Ausgangspopulation ist der letzte der Vorbereitungsschritte. Es ist wichtig nicht zu vergesssen, dass der Genotyp durch den Individuumskonstruktor normalerweise nicht zufällig gesetzt wird. Ein Aufruf von *initRandom()* darf deshalb nicht fehlen.

Von einer Generation zur nächsten gelangt man nun durch Füllen des Matingpools (Aufgabe des GA), Kreuzen der Individuen (Aufgabe des Matingpools) und Mutieren der Individuen (Aufgabe der Population). Diese drei Schritte könne beliebig oft wiederholt werden.

Schliesslich können die nichtdominierten Individuen in irgendeiner Art und Weise weiterverarbeitet werden. Allfällig dynamisch allozierter Speicherplatz sollte zum Schluss wieder freigegeben werden.

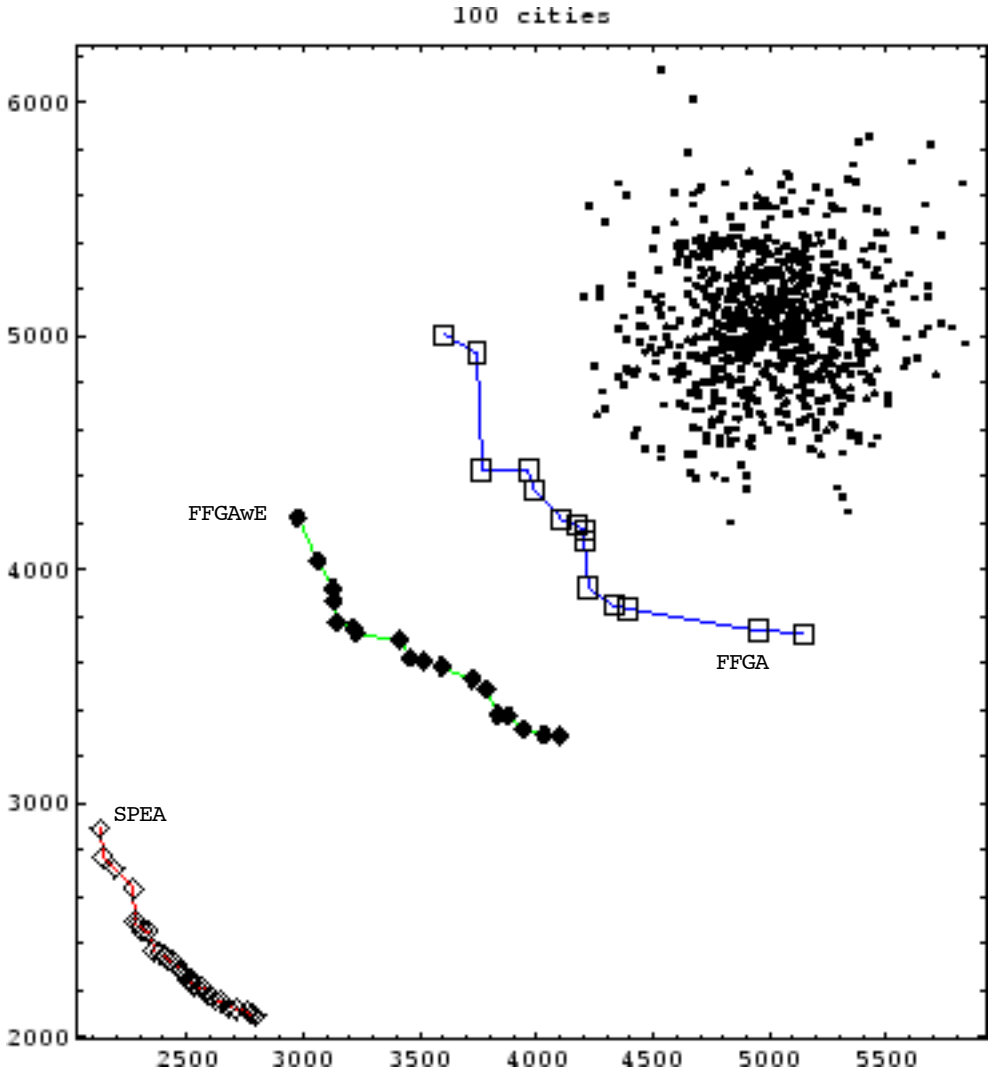
In Listing 5-5 sind alle soeben besprochenen Punkte umgesetzt.

```
int main( int argc, char* argv[] )
{
      const size_t      nrOfGenerations = 500;
      const size_t      nrOfTowns = 250;
      const size_t      popSize = 200;

      RandomNr          rn;
      SPEA              moga( randomNr, popSize, popSize / 4 );
      VecPopulation*    population = new VecPopulation( rn );
      VecPopulation*    nondomSet = new VecPopulation( rn );

      // initial population
      for ( size_t s = 0; s < popSize; ++s )
      {
            population->pushBack(new TSPInd( rn, nrOfTowns ));
      }
      population->initRandom();

      // evolve
      population->updateParetoSet( nondomSet );
      for ( size_t s = 0; s < nrOfGenerations; ++s )
      {
            VecPopulation     matingPool( rn );

            // fill the mating pool
            moga.select( population, &matingPool );
            // get rid of the old generation…
            delete population;
            // …and create the next one
            population = new VecPopulation( rn );
            matingPool.mate2( population );
            population->mutate();

            population->updateParetoSet( nondomSet );
      }

      // now do something with the nondominated front
      ...

      // clean up
      delete nondomSet;
      delete population;
}
```

Listing 5-5: Optimierungsroutine

6. Die Mehrzieloptimierungsverfahren im Vergleich

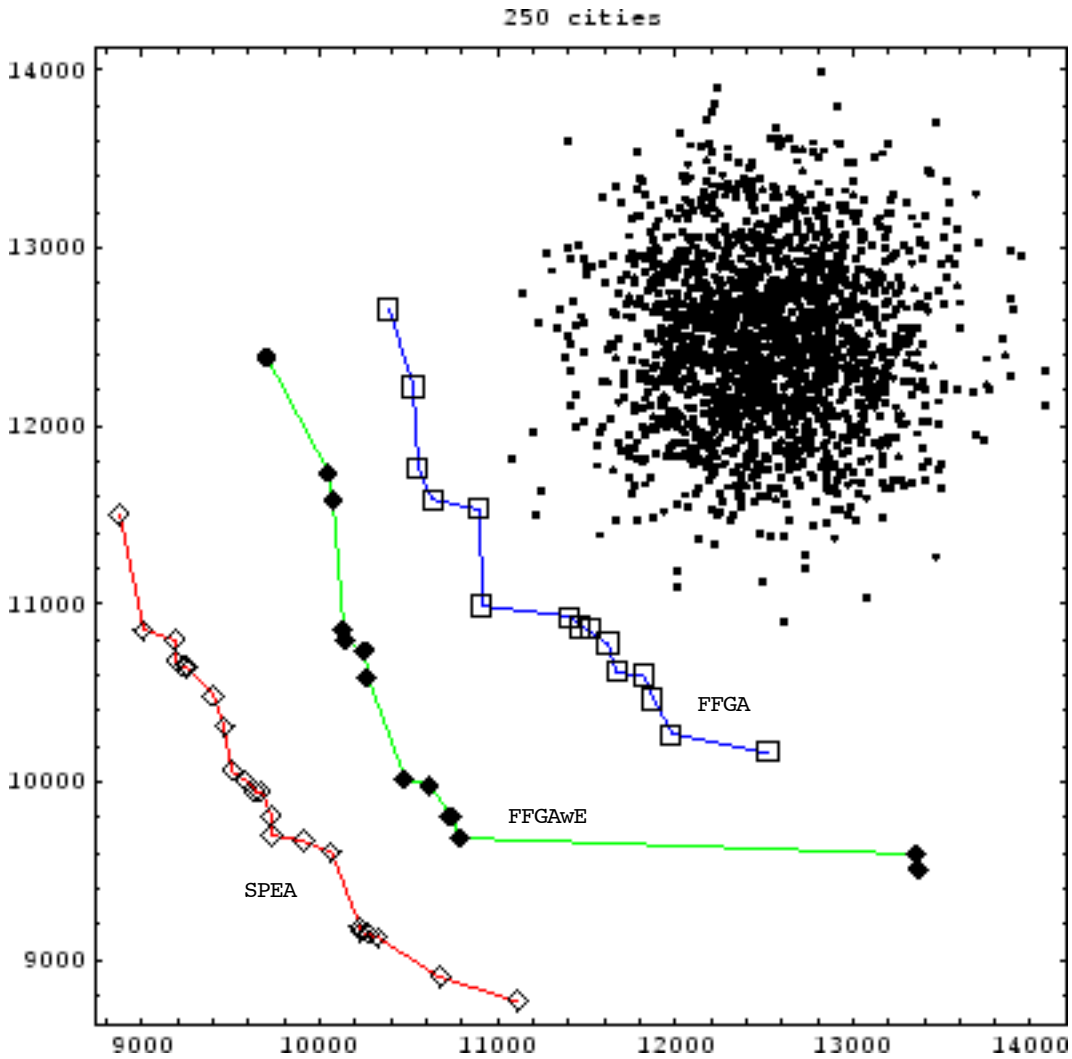
Gemeinsam ermöglichen FEMO und TSP einen Vergleich von FFGA und SPEA. Um den Vergleich fair zu gestalten, wurde FFGA noch so modifiziert, dass Elitismus unterstützt wird ($FFGAwE$): in Anlehnung an SPEA wird eine in der Grösse beschränkte Menge nichtdominierter Individuen zu Beginn jeder Selektion der aktuellen Population hinzugefügt und gleich anschliessend durch die Menge der nun nichtdominierten Populationsindividuen ersetzt. Die restlichen Schritte erfolgen unverändert.



Figur 6-1: Nichtdominierte Fronten bei 100 Städten

Es galt drei Probleme zu lösen, nämlich das Finden von Pareto-optimalen Rundreiserouten für 100, 250 und 500 Städte bei vorgegebenen Reisezeiten und Reisekosten

Jedes Problem wurde von einem Mehrziel-GA in 10 Läufen bearbeitet, wobei jeder Lauf mit einer anderen Ausgangspopulation[5] startete. Ein Lauf bestand aus 500 Generationen, und alle nichtdominierten Individuen jeder Generation wurden pro Lauf in einer speziellen Population gesammelt, welche keinen direkten oder indirekten Einfluss auf die Folgepopulationen hatte. Die Lösungsmenge eines bestimmten Verfahrens für eine bestimmte Anzahl Städte bestand schliesslich aus allen nichtdominierten Rundreiserouten der Vereinigung der 10 "Sammel-populationen".



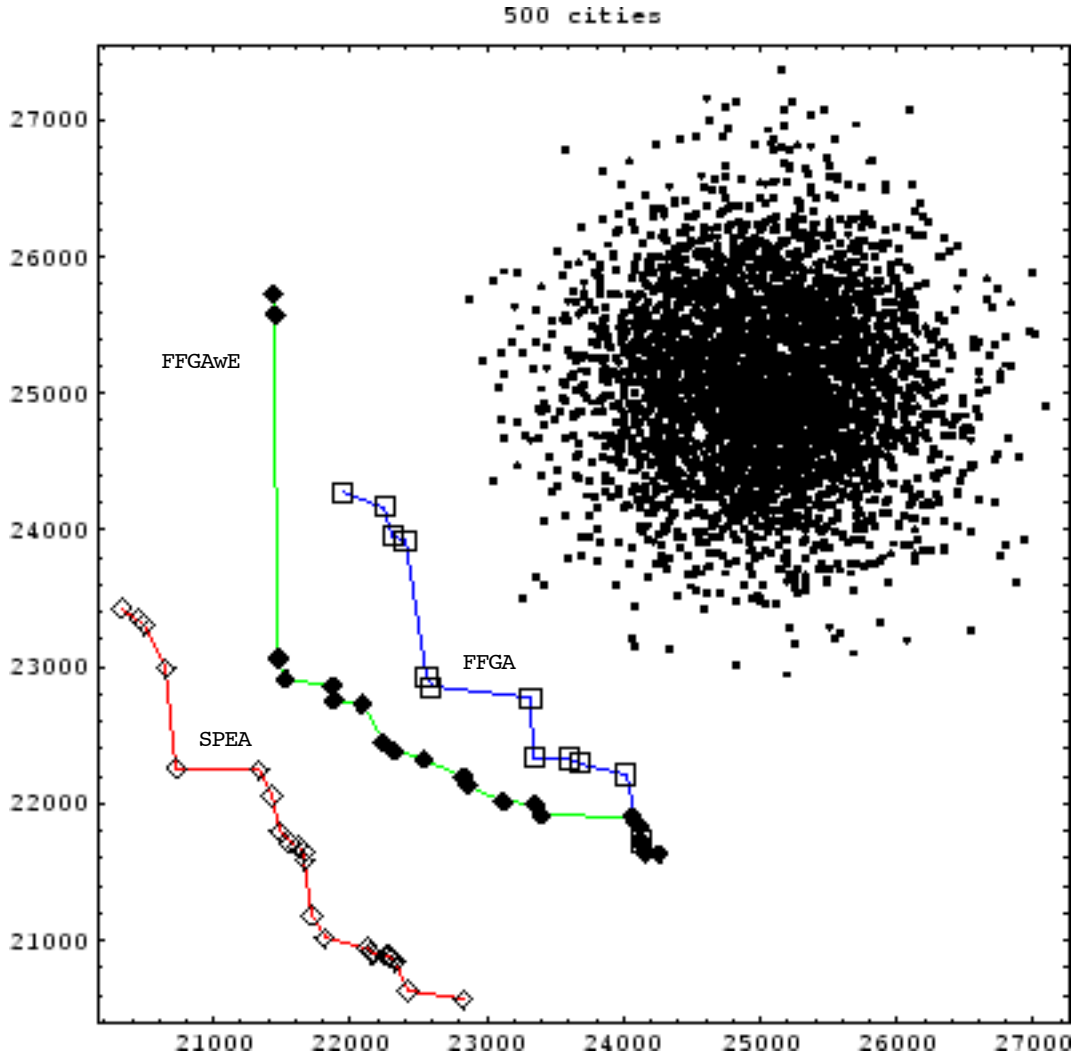
Figur 6-2: Nichtdominierte Fronten bei 250 Städten

Die Parameter für FFGA waren Matingpool-Grösse gleich Anzahl Städte; maximaler Erwartungswert für das Exponential-Ranking $E_{max} = 1.1$ [Nissen 1997, S. 70]; Nischenradius $T_{share} = 0.4886$[6] und Exponent ß = 2 für das Fitness-Sharing [Nissen 1997, S. 77].

[5] Die Ausgangspopulation von Problem x und Lauf y war für alle drei Verfahren jeweils dieselbe, mit der Ausnahme, dass FFGAs Populationen 25% grösser waren als diejenigen von FFGAwE und SPEA.
[6] Mündliche Angabe des Betreuers basierend auf einer Empfehlung von *Deb* und *Goldberg*.

Die Parameter für SPEA waren Matingpool-Grösse gleich vier Fünftel der Anzahl Städte, Grösse der Paretomenge gleich ein Fünftel der Anzahl Städte.

Die Parameter von FFGAwE entsprachen für Matingpool und Paretomenge denjenigen von SPEA, die restlichen denjenigen von FFGA.



Figur 6-3: Nichtdominierte Fronten bei 500 Städten

Figur 6-1 zeigt die Auswertung für 100 Städte, Figur 6-2 für 250 Städte und Figur 6-3 für 500 Städte. Wie zu erwarten, schnitt FFGAwE besser ab als FFGA. SPEA jedoch arbeitete deutlich bessere Fronten heraus. Diese Resultate sind statistich nicht repräsentativ, aber sie erhärten diejenigen in [Zitzler und Thiele 1998], welche mit einem anderen Mehrzieloptimierungsproblem bestimmt wurden.

7. Diskussion

FEMO ist das wesentliche Ergebnis der Semesterarbeit *Ein Framework für die Mehrzieloptimierung mit Genetischen Algorithmen*. Rück- resp. ausblickend stellen sich zwei Fragen: Wurden die Designvorgaben erreicht? Und in welcher Weise kann FEMO weiterentwickelt werden?

Drei Punkte in der Problemstellung (siehe Anhang A) beziehen sich auf das Framework: objektorientiertes Design, zwei Mehrzielverfahren und Erstellung einer Dokumentation. Der erste Punkt wurde konsequent umgesetzt; wo immer Varianten von Datentypen vorgesehen sind, sind diese durch Erbschaftsbeziehungen –das Prinzip des *information hiding* [Mössenböck 1998] respektierend– miteinander verknüpft, nie über (fehleranfälligere) Funktionszeiger resp. Prozedurvariablen. Der zweite Punkt wurde dank FFGAwE sogar leicht übertroffen, aber dieser Zusatz war nach der Implementation von SPEA und FFGA wohl nützlich, aber nicht wirklich aufwendig. Die Dokumentation besteht aus zwei wesentlichen Teilen: dem Tutorial (§5.3.) und der Referenz (Anhang C). Zeitlich bedingt ist das Tutorial eher knapp ausgefallen, so dass Programmierer mit wenig Erfahrung Mühe bekunden könnten. Dafür wurde umso mehr Energie in die Referenz investiert, die bei einer allfälligen Veröffentlichung des Frameworks auch nicht mehr ins Englische übersetzt werden muss.

Die allgemeinen Designziele verlangten Einfachheit, Spezialisierung, Flexibilität, Erweiterbarkeit, Kompaktheit. Wie mit TSP (Anhang B) gezeigt wurde, bedarf es relativ wenig Code, um ein Problem zu modellieren und optimieren. Ob die teilweise Spezialisierung auf Mehrzieloptimierung FEMOs Erweiterbarkeit hinsichtlich anderer EAs negativ beeinflusst, wird sich noch zeigen müssen (*Individual*s Schnittstelle könnte da ein Problem darstellen). Kompakt ist FEMO sicher, besteht es doch gerade mal aus 6 Gruppen von Klassen, von denen einige grosszügig formatiert auf einer A4-Seite problemlos Platz finden.

Die im Anforderungskatalog erwähnte hierarchische Struktur ist ein wesentliches Merkmal von FEMO: Es gibt Klassen von Genen, Chromosomen, Individuen und Populationen, wobei eine Abhängigkeit –wenn überhaupt– nur in eine Richtung besteht. Dann sollten je ein Kreuzungs- und Mutationsverfahren implementiert werden, FEMO bietet sogar drei Kreuzungs- und (je nach Sichtweise) wenigstens

fünf Mutationsoperatoren. Individuen wurden so entworfen, dass Zielwerte (der noch nicht existenten Unterklassen) nur dann berechnet werden müssen, wenn sie noch nie berechnet wurden – eine immense Laufzeitersparnis. Populationen unterstützen sowohl *mating restrictions* (nur für Elternpaare) als auch *clustering*, das Vereinen von zwei Populationen muss der Benutzer jedoch selbst in die Hand nehmen. Allgemeine Ein-/Ausgaberoutinen fehlen in FEMO gänzlich.

Erweitern sollte man FEMO wohl zuerst um weitere Mehrziel-GAs, um Benutzern eine grössere Auswahl zu ermöglichen. Ob die Trennung von Populationen in Funktionalität und Container sinnvoll war, muss abgeklärt werden; besteht kein Bedarf nach weiteren *Population*-Unterklassen, wäre es aus Gründen der Effizienz und Übersichtlichkeit wünschenswert, aus *Population* und *VecPopulation* eine konkrete (finale) Klasse zu machen. Mit Binär- und Permutationschromosomen und ihren Operatoren sind zwei wichtige Kodierungsformen bereits realisiert. Als nützlich könnten sich auch baumartige Chromosomen erweisen, vor allem für andere EAs. Ein-/Ausgaberoutinen drängen sich derzeit nicht auf, da mit C++-Streams einfache Routinen schnell implementiert werden können.

Am allerwichtigsten ist jedoch, dass FEMO von verschiedensten Interessenten breit eingesetzt und dann gemäss deren Feedback oder besser noch mit deren Hilfe weiterentwickelt wird.

Anhang A

Aufgabenstellung

Anhang B

Beispielprogramme

Anhang C

Klassenbeschreibungen

Class Descriptions

At the time of this writing, FEMO consists of the following classes (those printed in italic are abstract):

- TIKEAFException
    - LimitsException
    - NilException
    - ProbabilityException
- RandomNr
- *Gene*
    - BitGene
    - RealValGene
- *Chromosome*
    - *LinearChromosome*
        - BitStringChromosome
        - GeneVecChromosome
    - PermutationChromosome
- *Individual*
- *Population*
    - VecPopulation
- *MOEA*
    - FFGA
    - FFGAwE
    - SPEA

In the remainder of this appendix, the public and protected interface of these classes is described. Maintainers of the framework are additionally referred to the source code.

TIKEAFException[7]

Common base class of all exceptions that might be thrown from within the framework. In order to prevent the framework from checking for exceptions, `#define NOTIKEAFEXCEPTIONS` in header file `WithOutExceptions.h`.

Public Field:

`message`

Variable of type `string` that can be used for error messages. The framework stores in it the origin of the respective exception (i.e., "class::method").

Constructors:

`TIKEAFException()`

Default constructor; assigns the empty string (i.e., "") to `message`.

`TIKEAFException( string s )`

Assigns `s` to `message`.

LimitsException

Subclass of `TIKEAFException` that is thrown when array bounds are exceeded, interval limits violated, etc.

Public Field:

See `TIKEAFException`.

Constructors:

`LimitsException()`

Default constructor; calls `TIKEAFException()`.

`LimitsException( string s )`

Calls `TIKEAFException( s )`.

NilException

Subclass of `TIKEAFException` that is thrown when a pointer argument is 0 (`nil`, `null`) but mustn't be.

Public Field:

See `TIKEAFException`.

Constructors:

`NilException()`

Default constructor; calls `TIKEAFException()`.

`NilException( string s )`

[7] While in development, FEMO was code-named TIKEAF. In order not to introduce last-minute bugs, the name has not been changed in source code files.

Calls `TIKEAFException( s )`.

ProbabilityException

Subclass of `TIKEAFException` that is thrown when a probability parameter is not in [0; 1].

Public Field:

See `TIKEAFException`.

Constructors:

`ProbabilityException()`

Default constructor; calls `TIKEAFException()`.

`ProbabilityException( string s )`

Calls `TIKEAFException( s )`.

RandomNr

Instances of this base class generate uniformly distributed pseudo-random numbers. Generally, one only instance per optimization problem is necessary, and all classes that make use of random numbers keep a reference to that instance.

Protected Field:

`z`

Variable of type `long` that represents the current state of the generator.

Constructors:

`RandomNr()`

Default constructor; initializes the generator with the current time.

`RandomNr( long seed )`

Initializes the generator with `seed`.

Public Methods:

`setSeed( long seed )`

Reinitializes the generator with `seed`.

`uniform01(): double`

Returns a uniformly distributed random real in]0; 1[of type `double`.

`uniform0Max( size_t max ): size_t`

Returns a uniformly distributed random integer in [0; `max`[of type `size_t`.

`uniformMinMax( int min, int max ): int`

Returns a uniformly distributed random integer in [`min`; `max`[of type `int`.

`flipP( double p ): bool`

Returns with probability `p` `true` (vs. `false`).

Gene

Abstract base class which defines the interface of gene classes that act as building blocks for heterogeneous chromosomes (see `GeneVecChromosome` for an example).

Protected Fields:

`randomNr`

Reference of type `RandomNr`. `Gene` subclasses use this `RandomNr` instance to generate pseudo-random numbers.

`pMutation`

Variable of type `double` that stores the gene's mutation probability.

Constructor:

`Gene( RandomNr& rn, double pm )`

Makes `randomNr` refer to `rn` and initializes `pMutation` with `pm`.

PublicMethods:

`setPMutation( double pm )`

Reinitializes the mutation probability `pMutation` with `pm`.

`initRandom()`

Abstract method that randomly sets the value of the gene (the allele) when overridden in subclasses.

`mutate()`

Abstract method that mutates the value of the gene (the allele) with probability `pMutation` when overridden in subclasses.

`clone(): Gene*`

Abstract method that returns a pointer of type `Gene*` with same dynamic type as the `Gene` subclass it was invoked on.

BitGene

Subclass of `Gene` that has two possible alleles: `true` and `false`.

Protected Fields:

See `Gene`.

Constructor:

`BitGene( RandomNr& rn, double pm )`

Calls `Gene( rn, pm )`.

PublicMethods:

See also `Gene`.

`initRandom()`

Sets the allele with equal probability to either `true` or `false`.

`mutate()`

Inverts the allele with probability `pMutation`.

`clone(): Gene*`

Returns a pointer to a copy of the `BitGene`.

`setAllele( bool a )`

Sets the allele to `a`.

`getAllele(): bool`

Returns the allele (a `bool`).

RealValGene

Subclass of `Gene` that manages a real-valued allele in the interval [`min`; `max`] (see Protected Fields).

Protected Fields:

See also `Gene`.

`min`

Constant of type `double` denoting the smallest value the allele can assume.

`max`

Constant of type `double` denoting the biggest value the allele can assume.

`allele`

Variable of type `double` that represents the `RealValGene`'s current state.

Constructor:

`RealValGene( RandomNr& rn, double pm, double minA, double maxA )`

Calls `Gene( rn, pm )` and defines the allele's interval limits.

PublicMethods:

See also `Gene`.

`initRandom()`

Sets `allele` to a uniformly distributed random variable in [`min`; `max`].

`mutate()`

Calls `initRandom()`.

`clone(): Gene*`

Returns a pointer to a copy of the `RealValGene`.

`setAllele( double a )`

Sets `allele` to `a`.

`getAllele(): double`

Returns `allele` (a `double`).

Chromosome

Abstract base class that defines the interface of any kind of chromosome class (lists, trees, …).

Protected Field:

`randomNr`

Reference of type `RandomNr`. `Chromosome` subclasses use this `RandomNr` instance to generate pseudo-random numbers.

Constructor:

`Chromosome( RandomNr& rn )`

Makes `randomNr` refer to `rn`.

PublicMethods:

`initRandom()`

Abstract method that randomly sets the genotype when overridden in subclasses.

`mutate()`

Abstract method that mutates the genotype when overridden in subclasses (exception: `PermutationChromosome`).

`clone(): Chromosome*`

Abstract method that returns a pointer of type `Chromosome*` with same dynamic type as the `Chromosome` subclass it was invoked on.

`size(): size_t`

Abstract method that returns the number of genes in the respective `Chromosome` subclass instance.

LinearChromosome

Subclass of `Chromosome` and abstract base class of linearly structured chromosomes. It's main purpose is to provide crossover operators.

Protected Fields:

See also `Chromosome`.

`length`

Constant of type `size_t` denoting the chromosome's length (i.e., its size).

Constructor:

`LinearChromosome( RandomNr& rn, size_t l )`

Calls `Chromosome( rn )` and initializes `length` with `l`.

PublicMethods:

See also `Chromosome`.

`size(): size_t`

See `Chromosome`.

`setPMutation( size_t index, double pm )`

Abstract method that sets the mutation probability of the gene at position `index` to `pm`.

`copy( LinearChromosome* lc, size_t from, size_t to )`

Abstract method that copies `lc`'s alleles within position `from` (inclusive) and position `to` (exclusive). The dynamic types of `lc` and the chromosome this method was invoked on must be equal.

`onePointCrossover( RandomNr& rn, LinearChromosome* mother, LinearChromosome* father, LinearChromosome* daughter, LinearChromosome* son )`

Static method that features one-point crossover by making use of `copy()`. The children must be preallocated!

`twoPointCrossover( RandomNr& rn, LinearChromosome* mother, LinearChromosome* father, LinearChromosome* daughter, LinearChromosome* son )`

Static method that features two-point crossover by making use of `copy()`. The children must be preallocated!

BitStringChromosome

Subclass of `LinearChromosome` for binary-coded linear chromosomes.

Protected Fields:

See `LinearChromosome`.

Constructors:

`BitStringChromosome( RandomNr& rn, size_t l )`

Calls `LinearChromosome( rn, l )` and assigns each gene a mutation probability of $\texttt{length}^{-1}$.

`BitStringChromosome( RandomNr& rn, size_t l, double pm )`

Calls `LinearChromosome( rn, l )` and assigns each gene a mutation probability of `pm`.

Public Methods:

See also `LinearChromosome`.

`initRandom()`

Sets a bit with equal probability to either 0 or 1.

`mutate()`

Inverts a bit with its respective probability.

`clone(): Chromosome*`

Returns a pointer to a copy of the `BitStringChromosome`.

`setPMutation( size_t index, double pm )`

See `LinearChromosome`.

`copy( LinearChromosome* lc, size_t from, size_t to )`

See `LinearChromosome`.

`set( size_t index, int a )`

Sets the bit at position `index` to 0 if `a` equals 0 and to 1 if it doesn't.

`get( size_t index ): int`

Returns the bit at position `index` (an `int`).

GeneVecChromosome

Subclass of `LinearChromosome` for management of `Gene`s.

Protected Fields:

See `LinearChromosome`.

Constructor:

`GeneVecChromosome( RandomNr& rn, vector< Gene* >& vg )`

Calls `LinearChromosome( rn, vg.size() )` and takes over the management of the `Gene`s in `vg`.

Public Methods:

See also `LinearChromosome`.

`initRandom()`

Calls each `Gene`'s `initRandom()` method.

`mutate()`

Calls each `Gene`'s `mutate()` method.

`clone(): Chromosome*`

Returns a pointer to a deep copy of the `GeneVecChromosome`.

`setPMutation( size_t index, double pm )`

Calls `setPMutation( pm )` of the `Gene` specified by `index`.

`copy( LinearChromosome* lc, size_t from, size_t to )`

See `LinearChromosome`.

`at( size_t index ): Gene*`

Returns a pointer to the `Gene` at position `index`.

PermutationChromosome

Subclass of `Chromosome` for use with permutation problems. A permutation is represented by a certain arrangement of the positive integers from 1 up to the chromosome's length.

Protected Field:

See `Chromosome`.

Constructor:

`PermutationChromosome( RandomNr& rn, size_t l )`

Calls `Chromosome( rn )` and makes the chromosome represent a `l`-permutation.

Public Methods:

`initRandom()`

Permutes the integers.

`mutate()`

Does nothing!

`clone(): Chromosome*`

Returns a pointer to a copy of the `PermutationChromosome`.

`size(): size_t`

Returns the length of the permutation.

`get( size_t index ): int`

Returns the integer at position `index` (an `int`).

`inversionMutation()`

Mutates the chromosome by inverting an integer subsequence.

`scrambleSublistMutation()`

Mutates the chromosome by permuting an integer subsequence.

`moveSequenceMutation()`

Mutates the chromosome by relocating an integer subsequence.

`swapMutation()`

Mutates the chromosome by exchanging the positions of two integers.

`uniformOrderBasedCrossover( RandomNr& rn, PermutationChromosome* mother, PermutationChromosome* father, PermutationChromosome* daughter, PermutationChromosome* son )`

Static method that features uniform-order-based crossover. The children must be preallocated!

`edgeRecombinationCrossover( RandomNr& rn, PermutationChromosome* mother, PermutationChromosome* father, PermutationChromosome* child )`

Static method that features edge-recombination crossover. The child must be preallocated!

Individual

Abstract base class that defines the interface of individuals used with multiobjective optimization EAs. Once an objective has been calculated, the value is stored in order to avoid further (expensive) calculations.

Public Type:

`enum Distance { geno, phenoPar, phenoObj }`

Enumeration defining three types of distance measures.

Public Field:

`fitness`

Variable of type `double` that clients of `Individual` can use as they see fit.

Protected Fields:

`nrOfObjectives`

Constant of type `size_t` denoting the number of objectives.

`randomNr`

Reference of type `RandomNr` that `Individual` subclasses use to generate pseudo-random numbers.

Constructor:

`Individual( RandomNr& rn, size_t noo )`

Makes `randomNr` reference `rn` and initializes `nrOfObjectives` with `noo`.

Protected Methods:

`covers( bool bigger, Individual* ind )`

Returns `true` if `this` covers `ind`. `bigger` must be `true` for maximization and `false` for minimization problems.

`dominates( bool bigger, Individual* ind )`

Returns `true` if `this` dominates `ind`. `bigger` must be `true` for maximization and `false` for minimization problems.

`subInitRandom()`

Abstract method that randomly sets the chromosomes.

`subMutate(): bool`

Abstract method that returns `true` if it mutated the chromosomes.

`subGetObjective( size_t index ): double`

Abstract method that returns the objective specified by `index`.

Public Methods:

`initRandom()`

Calls `subInitRandom()` and forgets any previous calculated objectives.

`mutate()`

Calls `subMutate()` and forgets any previous calculated objectives if `subMutate()` returns `true`.

`getObjective( size_t index ): double`

Returns the objective specified by `index`. Calls `subGetObjective( index )` only if the respective objective hasn't been calculated yet.

`clone(): Individual*`

Abstract method that returns a pointer of type `Individual*` with same dynamic type as the `Individual` subclass it was invoked on.

`distance( Distance type, Individual* ind ): double`

Abstract method that returns the `type` distance between `this` and `ind`.

`covers( Individual* ind ): bool`

Abstract method that returns `true` if `this` covers `ind`.

`dominates( Individual* ind ): bool`

Abstract method that returns `true` if `this` dominates `ind`.

`equals( Individual* ind ): bool`

Returns `true` if `this` equals `ind`.

`mateWith( Individual* ind, vector< Individual* >& children )`

Abstract method that tells the instance it was invoked on to mate with `ind` and put the offspring at the end of `children`.

Population

Abstract container class that manages `Individual`s. Most of the functionality, like mating, clustering,

etc., is already implemented such that subclasses mainly have to provide the actual container only. And they should support efficient random-access (see `at()`).

Protected Field:

`randomNr`

Reference of type `RandomNr` that `Population` and its subclasses use to generate pseudo-random numbers.

Constructor:

`Population( RandomNr& rn )`

Makes `randomNr` reference `rn`.

Protected Methods:

`switchAt( size_t index1, size_t index2 )`

Abstract method that exchanges the positions of the `Individual`s specified by `index1` and `index2`.

`deleteAt( vector< size_t >& indices )`

Abstract method that deletes the `Individual`s the positions of which appear in the increasingly ordered `indices`.

`deleteAll()`

Abstract method that gets rid of all `Individual`s.

`nondominatedPoints( Population* population )`

Clones all nondominated `Individual`s into `population`.

Public Methods:

`size(): size_t`

Abstract method that returns the number of `Individual`s in the population.

`sort( bool increasing )`

Sorts the `Individual`s according to their fitness in increasing (`true`) or decreasing (`false`) order.

`at( size_t index ): Individual*`

Abstract method that returns a pointer to the `Individual` at position `index`.

`pushBack( Individual* ind )`

Abstract method that adds the `Individual` pointed to by `ind` at the end of the population.

`popBack(): Individual*`

Abstract method that removes the `Individual` at the end of the population and returns a pointer to it.

`removeAt( size_t index ): Individual*`

Abstract method that removes the `Individual` specified by `index` and returns a pointer to it. The order of the remaining `Individual`s might change!

`initRandom()`

Calls each `Individual`'s `initRandom()`.

`mutate()`

Calls each `Individual`'s `mutate()`.

*c*lone(): Population*

Returns a pointer of type Population* with same dynamic type as the Population subclass it was invoked on.

deleteCovered()

Deletes all Individuals that are coverd or duplicates of nondominated Individuals.

updateParetoSet(Population* paretoSet)

Calls nondominatedPoints(paretoSet) and then calls paretoSet->deleteCovered().

mate2(Population* population)

Makes pairs of parents from all Individuals, tells them to mate, and puts their offspring into population.

mate2(Individual::Distance disType, double distance, int maxAttempts, Population* population)

Same as mate2(population) with mating restrictions.

aveLinkCluster(Individual::Distance disType, size_t maxPopSize)

Reduces the population size while its bigger than maxPopSize using average linkage clustering and the distance measure specified by disType.

VecPopulation

Subclass of Population that uses a vector<> to manage the Individuals.

Protected Field:

See Population.

Constructors:

VecPopulation(RandomNr& rn)

Calls Population(rn).

VecPopulation(RandomNr& rn, size_t estimate)

Calls Population(rn) and reserves memory for estimate Individuals (might save time).

Protected Methods:

See also Population.

switchAt(size_t index1, size_t index2)

See Population.

deleteAt(vector< size_t >& indices)

See Population.

deleteAll()

See Population.

Public Methods:

See also Population.

size(): size_t

See `Population`.

`at( size_t index ): Individual*`

See `Population`.

`pushBack( Individual* ind )`

See `Population`.

`popBack(): Individual*`

See `Population`.

`removeAt( size_t index ): Individual*`

See `Population`.

`clone(): Population*`

Returns a pointer to a deep copy of the `VecPopulation`.

MOEA

Abstract base class of multiobjective-evolutionary algorithm classes; that is, classes that take care of fitness assignment (ranking) and selection (i.e., filling the mating pool).

Protected Fields:

`randomNr`

Reference of type `RandomNr`. `MOEA` subclasses use this `RandomNr` instance to generate pseudo-random numbers.

`matingPoolSize`

Constant of type `size_t` that `MOEA` subclasses use to look up the mating-pool size.

Constructor:

`MOEA( RandomNr& randomNr, size_t matingPoolSize )`

Makes `randomNr` refer to `rn` and initializes `matingPoolSize` with `mps`.

Public Method:

`select( Population* population, Population* matingPool )`

Abstract method that has to be overridden in concrete `MOEA` subclasses. The semantics are: A fitness value is assigned to individuals in `population`, `population` is possibly sorted, and `matingPoolSize` times an individual from `population` is cloned into `matingPool` (i.e., appended at the end). If not stated otherwise, no individuals have been removed from or added to `population`, and `population` and `matingPool` can point to the same `Population` instance.

FFGA

Subclass of `MOEA` that features fitness assignment (exponential ranking, fitness sharing) and (stochastic universal sampling) selection of *Fonseca*'s and *Fleming*'s 1993 genetic algorithm.

Protected Fields:

See `MOEA`.

Constructor:

`FFGA( RandomNr& rn, size_t mps, double eMax, Individual::Distance disType, double sigmaShare, double beta )`

Calls `MOEA( rn, mps )` and defines the parameters necessary for exponential ranking and fitness sharing.

Public Method:

`select( Population* population, Population* matingPool )`

See `MOEA`.

FFGAwE

Subclass of `MOEA` that features fitness assignment (exponential ranking, fitness sharing) and (stochastic universal sampling) selection of *Fonseca*'s and *Fleming*'s 1993 genetic algorithm with additional support of elitism.

Protected Fields:

See `MOEA`.

Constructor:

`FFGAwE( RandomNr& rn, size_t mps, size_t paretoSize, double eMax, Individual::Distance disType, double sigmaShare, double beta )`

Calls `MOEA( rn, mps )` and defines the maximum Pareto-set size as well as the parameters necessary for exponential ranking and fitness sharing.

Public Methods:

`select( Population* population, Population* matingPool )`

See also `MOEA`. In general, `population` contains additional individuals after the call to `select()` as all individuals of the internal Pareto set are cloned into it before ranking!

`cloneParetoPoints( Population* population )`

Clones all individuals of the internal Pareto set into `population`.

SPEA

Subclass of `MOEA` that features fitness assignment and (binary tournament) selection of the *Strength Pareto Evolutionary Algorithm*. Every instance manages its own Pareto set.

Protected Fields:

See `MOEA`.

Constructor:

`SPEA( RandomNr& rn, size_t mps, size_t paretoSize )`

Calls `MOEA( rn, mps )` and defines the maximum Pareto-set size.

Public Methods:

`select( Population* population, Population* matingPool )`

See `MOEA`.

`cloneParetoPoints( Population* population )`

Clones all individuals of the internal Pareto set into `population`.

Bibliographie

Genetik:

[Miram und Scharf 1988]
MIRAM, Wolfgang, SCHARF, Karl-Heinz (Hrsg.). – *Biologie heute SII*, Schroedel, Hannover, Neubearbeitung 1988.

Evolutionäre Algorithmen:

[Blickle 1996]
BLICKLE, Tobias. – *Yet Another Genetic Programming Library In C*, Programmdokumentation, Institut für Technische Informatik und Kommunikationsnetze, Eidgenössische Technische Hochschule Zürich, November 1996.

[Fonseca und Fleming 1993]
FONSECA, Carlos M., FLEMING, Peter J. – *Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion and Generalization*, pp. 416-423 in: FORREST, Stephanie (Ed.). – *Proceedings of the Fifth International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo (California), 1993.

[Fonseca und Fleming 1995]
FONSECA, Carlos M., FLEMING, Peter J. – *An Overview of Evolutionary Algorithms in Multiobjective Optimization*, pp. 1-16 in: *Evolutionary Computation*, Volume 3, Number 1, 1995.

[Haupt und Haupt 1998]
HAUPT, Randy L., HAUPT, Sue Ellen. – *Practical Genetic Algorithms*, John Wiley & Sons, New York, 1998.

[Mitchell 1996]
MITCHELL, Melanie. – *An Introduction to Genetic Algortihms*, The MIT Press, Cambridge (Massachusetts), 1996.

[Nissen 1997]
NISSEN, Volker. – *Einführung in Evolutionäre Algorithmen: Optimierung nach dem Vorbild der Natur*, Vieweg, Braunschweig/Wiesbaden, 1997.

[Zitzler und Thiele 1998]
ZITZLER, Eckart, THIELE, Lothar. - *An Evolutionary Algorithm for Multiobjective Optimization: The Strength Pareto Approach*, TIK-Report 43, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology Zurich, May 1998.

OOP und C++:

[Mössenböck 1998]
MÖSSENBÖCK, Hanspeter. – *Objektorientierte Programmierung in Oberon-2*, Springer, Heidelberg, 3. Auflage 1998.

[Stroustrup 1997]
STROUSTRUP, Bjarne. – *The C++ Programming Language*, Addison-Wesley, Reading (Massachusetts), 3rd Edition 1997.

URLs:

[URL 4-1]
http://research.de.uu.net:8080/encore/www/Q20_1.htm

[URL 4-2]
http://www.shef.ac.uk/uni/project/gaipp/gatbx.html

[URL 4-3]
http://www.aridolan.com/ga/gaa/gaa.html

[URL 4-4]
http://lancet.mit.edu/galib-2.4/ClassHierarchy.html

[URL 4-5]
http://lancet.mit.edu/ga/

[URL 4-6]
http://geneuro.ugr.es/~jmerelo/EO.html

[URL 5-1]
http://java.sun.com/products/hotspot/index.html

[URL 5-2]
http://egcs.cygnus.com/