Ressourcenverteilung und Informationsaustausch konkurrierender Unterpopulationen in Erweiterten Evolutionären Algorithmen

Arne Griep

Technische Universität Ilmenau

Matrikel: M96

Matrikel-Nr.: 25195

Betreuer: Univ.-Prof. Dr.-Ing. habil. Horst Puta, TU Ilmenau

Betreuer: Dr.-Ing. Hartmut Pohlheim, DaimlerChrysler AG, Berlin

Inhaltsverzeichnis

INHALTSVERZEICHNIS ..... **I**

ABKÜRZUNGEN ..... **II**

1 EINFÜHRUNG **1**

2 KONKURRENZ **5**

2.1 VERÖFFENTLICHUNGEN ZUM THEMA KONKURRENZ 5
2.2 ABLAUF DER KONKURRENZFUNKTION 7
2.2.1 Erfolgsbewertung *7*
2.2.2 Ressourcenzuteilung (resource division) *8*
2.2.3 Ressourcenumverteilung (resource distribution) *10*

3 ERGEBNISSE **15**

3.1 UMVERTEILUNGSGESCHWINDIGKEIT 15
3.1.1 Konkurrenzintervall *von 5 Generationen - verschiedene Konkurrenzraten* *15*
3.1.2 Konkurrenzintervall *von 10 Generationen - verschiedene Konkurrenzraten* *17*
3.2 VERTEILUNGSDRUCK 18
3.3 KOOPERATION 19

4 IMPLEMENTIERUNG **25**

4.1 PROGRAMMABLAUF DER POSITIONSERMITTLUNG UND RESSOURCENZU- /-UMVERTEILUNG 26
4.2 EXTREME PROGRAMMING 27
4.3 ANWENDUNG DES EXTREME PROGRAMMING BEI DER IMPLEMENTIERUNG 28

5 ZUSAMMENFASSUNG **31**

6 ANHANG **33**

6.1 QUELLTEXTE 33
6.1.1 Quelltextauszug `compete.m` *33*
6.1.2 Quelltextauszug `compdiv.m` *37*
6.1.3 Quelltext `testcompete.m` *39*
6.2 LITERATURVERZEICHNIS 41
6.3 ABBILDUNGSVERZEICHNIS 42
6.4 TABELLENVERZEICHNIS 43

Abkürzungen

Abb.	Abbildung
DP	Division Pressure
EA	Evolutionäre Algorithmen
GA	Genetische Algorithmen
Subpop	Unterpopulation
Tab.	Tabelle
XP	Extreme Programming
ZF-Werte	Zielfunktionswerte

1 Einführung

In unserem Alltag begegnet uns oft das Adjektiv <optimal>. LANGENSCHEIDTS FREMDWÖRTERBUCH gibt dazu folgende Auskunft:

- **optimal:** bestmöglich, das Beste verkörpernd

Ein Gegenstand zum Beispiel wird als <optimal> bezeichnet, wenn seine Eigenschaften derart sind, daß sie das bestmögliche Ergebnis darstellen. Den Vorgang, wie etwas optimal gestaltet wird, nennt man <Optimierung>. Optimierung bedeutet, die Einflußgrößen des Problems zu finden, so daß sich hinsichtlich eines Zielkriteriums (oder mehrerer Zielkriterien) das bestmögliche, das optimale Resultat einstellt. Wenn also ein zeitlicher Ablauf optimal gestaltet werden soll, muß das Kriterium <Zeit> meistens möglichst klein gehalten werden. Daher müssen die Parameter dieses Ablaufes dermaßen angepaßt werden, daß dieses Ziel erreicht wird.

Ein Techniker möchte ein System so erschaffen, daß es optimal wird. Er sucht also nach den Einstellungen für ein System, für die es an eine Aufgabenstellung bestmöglich angepaßt ist. Die Angepaßtheit des Systems wird durch den Wert des Zielkriteriums verkörpert. Die Abhängigkeit des Zielkriteriums von den Variablen des Systems wird <Zielfunktion> genannt. Um nun dieses System hinsichtlich dieses Kriteriums zu optimieren, wird die Zielfunktion maximiert oder minimiert. Stellt das Zielkriterium zum Beispiel einen Ertrag dar, findet sich das Optimum dieses Systems im Maximum der Zielfunktion. Handelt es sich beim Zielkriterium um eine Zeit, ist häufig das Minimum ausschlaggebend. Eine Maximierung kann immer in eine Minimierung überführt werden, indem nach dem Minimum der negierten Zielfunktion gesucht wird. Das läßt für beide Varianten eine einheitliche Betrachtungsweise zu. Daher wird im folgenden immer von Minimierung ausgegangen.

Eine Klasse von Verfahren zur Optimierung sind die Evolutionären Algorithmen. Sie gehören zur Gruppe der stochastischen Suchverfahren, da sie (mehr oder weniger) zufallsbehaftet arbeiten. Sie entstanden in Anlehnung an Vorgänge in der Natur. Diese zeigt nämlich, wie durch Einfluß des Zufalls, durch Selektion, Variation, Migration und Konkurrenz im Laufe der Generationen Lebewesen entstanden, die gut an ihre Umgebung angepaßt sind. So finden sich denn genau diese biologischen Vorgänge und ihre Begriffe in den Evolutionären Algorithmen wieder. Ein Lebewesen (Individuum) wird durch die Werte eines Satzes von Variablen beschrieben, sein Lebensraum durch die Zielfunktion und seine Überlebenswahrscheinlichkeit durch seinen Zielfunktionswert im Verhältnis zu den Zielfunktionswerten der anderen Individuen (Fitneß). Da hier generell von Minimierung ausgegangen wird, bedeutet ein kleinerer Zielfunktionswert eine höhere Überlebenswahrscheinlichkeit. Eine oder mehrere Populationen von Individuen durchlaufen nun obengenannte Vorgänge und erzeugen mit steigender Anzahl der Generationen immer besser angepaßte Individuen und damit immer bessere Einstellungen des Systems hinsichtlich des Zielkriteriums. Es findet Evolution statt.

Abb. 1-1 Ablauf eines Evolutionären Algorithmus

Es muß aber nicht, wie bereits angedeutet, bei einer Population bleiben. Es hat gewisse Vorteile, sich von diesem Ansatz zu lösen. SCHWEHM unterscheidet drei Populationsmodelle in [Swm96]:

- Globales Populationsmodell
- Regionales Populationsmodell
- Lokales Populationsmodell

Für die vorliegende Arbeit ist das regionale Populationsmodell von Bedeutung, da es die Unterteilung einer Gesamtpopulation in Unterpopulationen erlaubt. Ursprünglich ist es aus der Idee entstanden, die EA (Evolutionäre Algorithmen) auf Parallelrechner umzusetzen. Neben der Möglichkeit der Parallelisierung ergaben sich weitere Vorteile, die den Einsatz dieses Modells auch auf Nichtparallelrechner lohnend erschienen ließen. Ein Vorteil besteht darin, daß die Unterpopulationen unterschiedliche Strategien nutzen und darüber hinaus in Wettbewerb um Ressourcen stehen können.

Dieser Vorgang „Konkurrenz“ soll in den nachfolgenden Kapiteln besprochen werden. Folgende Fragen stehen dabei im Vordergrund:

1. Wie wird die Güte einer Population bewertet?
2. Wie soll die Güte der einzelnen Populationen verglichen werden?
3. Wie werden die vorhanden Ressourcen auf die Populationen aufgeteilt?
4. Kommt Kooperation zwischen den Populationen zustande oder tritt nur Konkurrenz auf?

Kapitel 2 beschäftigt sich mit den Ablaufschritten, die ein Konkurrenzmodul durchlaufen muß und geht dabei auf die Fragen 1.-3. ein. Kapitel 3 zeigt experimentelle Untersuchungen zu Effek-

ten der Konkurrenz und erörtert Frage 4. Kapitel 4 befaßt sich mit der Implementierung des Konkurrenzmoduls und einigen Erläuterungen zum Extreme Programming. In Kapitel 5 werden schließlich die Ergebnisse dieser Arbeit zusammengefaßt und es wird ein Ausblick auf weitere Arbeiten gegeben. Der Anhang enthält neben Abbildungsverzeichnis, Tabellenverzeichnis, Literaturverzeichnis und Abkürzungsverzeichnis den kommentierten Quellcode der implementierten Routinen.

2 Konkurrenz

Die Prinzipien, nach denen die Evolutionären Algorithmen arbeiten, sind der Natur entlehnt. Dazu gehört auch der Vorgang der Konkurrenz. Konkurrenz tritt zwischen den Individuen und auf höherer Ebene zwischen den Arten auf. Sinn und Zweck dieses Vorganges ist eine „gerechte“ Zuteilung von Ressourcen. In der Natur bestehen diese Ressourcen hauptsächlich aus Lebensraum und Futter. Gerechte Zuteilung heißt: jede Art kann nur Ressourcen entsprechend ihres Anpassungsgrades erreichen. Durch Konkurrenz werden folglich die Arten belohnt, die besser sind. Durch den größeren Ressourcenanteil können sie ihren Fortbestand sichern und sich stärker vermehren.

Eine Art ist durch bestimmte gemeinsame Eigenschaften ihrer Individuen gekennzeichnet. Die Gesamtheit der Eigenschaften ergibt eine Strategie. Es gibt demnach Arten mit verschiedenen Strategien, die miteinander in Konkurrenz um Ressourcen stehen. Der biologische Begriff „Art“ findet seine Entsprechung in den Evolutionären Algorithmen im Begriff „Unterpopulation“. Ressourcen bedeuten in den EA Rechenzeit. Die Rechenzeit wird durch den Anteil an Individuen, die einer Unterpopulation angehören, ausgedrückt. Im Ergebnis erhalten nur Strategien Rechenzeit, die gute Resultate erzeugen und damit die Optimierung voranbringen.

Eine grobe Struktur einer Konkurrenzfunktion sieht folgendermaßen aus:

1. Der Erfolg der Unterpopulationen muß bewertet werden.
2. Die Zuteilung der Ressourcen muß entsprechend ihres Erfolges berechnet werden (*ressource division*).
3. Die Ressourcen müssen anhand der berechneten Aufteilung umverteilt werden (*ressouce distribution*).

Der nachfolgende Abschnitt 2.1 beschäftigt sich mit bisherigen Arbeiten zur Konkurrenz. Danach folgt eine ausführliche Erläuterung der Konkurrenzfunktion im Abschnitt 2.2. Hier wird im einzelnen auf die genannten Ablaufschritte eingegangen.

2.1 Veröffentlichungen zum Thema Konkurrenz

Bisher gibt es nur wenige Untersuchungen, die sich mit dem Thema konkurrierender Unterpopulationen innerhalb der Evolutionären Algorithmen auseinandersetzen. Hier sind die Arbeiten von D. SCHLIERKAMP-VOOSEN und H. MÜHLENBEIN [SVM94] und [SVM96] zu nennen sowie auch die Arbeiten von H. POHLHEIM [Poh98]. In [CPR96] werden konkurrierende Strategien angewendet. Ein Ansatz, der Konkurrenz zwischen Strategien auf anderer Ebene nutzt, findet sich in [IK2001].

In [SVM94] wird von einer konstanten Gesamtzahl von Individuen ausgegangen; die Größe der Unterpopulationen variiert. Das Qualitätsmerkmal der Unterpopulationen basiert auf der Fitneß des jeweils besten Individuums. Außerdem wird die Qualität der letzten 10 Wettbewerbe mit ein-

bezogen, um ineffizientes Oszillieren der Populationsgrößen zu vermeiden. Die Unterpopulation mit dem besten Individuum bekommt Ressourcen in Form von Individuen zugeteilt, die alle anderen Unterpopulationen abgeben müssen. Die Anzahl der abzugebenden Individuen einer Gruppe wird dabei auf 1/8 ihrer derzeitigen Populationsgröße festgelegt. Letztere darf ein vorgegebenes Limit nicht unterschreiten, um sicherzustellen, daß alle Strategien überleben.

Eine Erweiterung dieses Ansatzes findet sich in [SVM96] *(extended competition model)*. Hier bleibt die Größe der gesamten Population nicht mehr konstant, sondern wird verändert. Die Natur zeigt, daß sich die Populationsgrößen durch das Nebeneinanderexistieren der Arten und Interaktion zwischen ihnen ständig anpassen. Da alle Arten unterschiedliche Überlebensstrategien nutzen, benötigen sie teilweise unterschiedliche Populationsgrößen, um effizient zu sein. Dies wird durch die Einführung einer Wachstumsrate für jede einzelne Gruppe umgesetzt. Außerdem wird ein sogenannter Verbrauchsfaktor *(consumption factor)* eingeführt. Aus Sicht der Natur ist er der Verbrauch einer begrenzt vorhandenen Ressourcenmenge durch ein Mitglied einer Art. Je höher er ist, um so weniger Individuen können durch diese vorgegebene Ressourcenmenge unterstützt werden. Das wird durch Implementierung von normalisierten Populationsgrößen umgesetzt. Die Summe dieser bleibt konstant, da sie durch die limitierte Ressource (Rechenzeit) begrenzt wird. Die Summe der realen Populationsgrößen variiert aber in Abhängigkeit des jeweiligen Verbrauchsfaktors. Der Nutzen dieser Methode wurde am Beispiel zweier konkurrierender Unterpopulationen gezeigt. Während die eine Gruppe nur Rekombination (geringer Verbrauchsfaktor – viele Individuen) benutzt, arbeitet die andere nur mit Mutation (hoher Verbrauchsfaktor – wenige Individuen). Die erste kam am Anfang zum Zuge (Breitensuche), die zweite im weiteren Verlauf (Feinsuche). Während der Optimierung sank also die Gesamtzahl von vielen Individuen am Anfang auf wenige Individuen am Ende ab.

H. POHLHEIM [Poh98] macht die Erfolsbewertung der Unterpopulationen von allen Individuen abhängig. D. h., der Erfolg einer Unterpopulation berechnet sich aus dem gemittelten Fitneßwert ihrer Individuen. Die Ordnung dieser Werte ergibt den Rang der Unterpopulation, wobei ein kleinerer Rang eine bessere Unterpopulation bedeutet. Das ermöglicht eine sehr kritische Beurteilung der Qualität der Unterpopulationen gegenüber [SVM94]. Für die Ressourcenaufteilung wird ein Verfahren benutzt, das die Ressourcen gewichtet nach dem Rang der Unterpopulationen auf letztere verteilt. Abhängig von einem Parameter *Verteilungsdruck* kann die Stärke der Bevorzugung der besseren Gruppen eingestellt werden. Diese Methode hat den Vorteil, daß nicht alle Unterpopulationen zu Gunsten der besten benachteiligt werden; sie räumt diesen Strategien auch noch gewisse Chancen ein. Auch hier darf ein *Unterpopulationsminimum* nicht unterschritten werden.

Wie die Methode konkurrierender Unterpopulationen in einer realen Anwendung aussieht, zeigen F. CORNO, P. PRINETTO, M. REBAUDENGO, M.S. REORDA in [CPR96]. Hierbei geht es um das Testen digitaler Schaltkreise. Dieses nimmt heutzutage einen großen Anteil der Entwicklungskosten und –zeit ein. Aus diesem Grund wurde in den letzten Jahren viel Forschungsaufwand betrieben, um effizientere Algorithmen für die automatische Erstellung von Testsequenzen für digitale Schaltkreise zu finden. Nachteile bisheriger Algorithmen lagen in ihrer Unfähigkeit, schwer testbare Fehler zu finden und in der Notwendigkeit sorgfältiger Einstellung ihrer Parameter. Ein zweckmäßiger Ansatz ist die Nutzung Evolutionärer Algorithmen mit konkurrierenden Unterpopulationen. Zu jeder dieser Populationen gehören mehrere Prozessoren eines Parallelrechners, die alle unterschiedliche GA-Suchverfahren bearbeiten. Die Prozessoren sind in Unterpopulationen aufgeteilt, deren Parametersätze verschieden sind. Im Ergebnis dessen werden auch verschiedene

Unterräume des Suchraumes abgedeckt. Periodisch werden die Resultate aller Gruppen ermittelt, verglichen, und Prozessoren werden von den schlechteren Gruppen zu den besseren „verschoben“. Sie übernehmen folglich die Parametersätze, die im Moment am günstigsten erscheinen. Die Existenz jeder Gruppe wird sichergestellt, d. h., mindestens ein Prozessor pro Gruppe bleibt übrig.

Konkurrenz muß nicht auf der Ebene der Unterpopulationen stattfinden. Der Einsatz von Strategien kann anstatt über die Größe von Unterpopulationen z. B. über sich anpassende Wahrscheinlichkeiten gesteuert werden. Dieser Ansatz wird von C. IGEL und M. KREUTZ in [IK2001] verfolgt. Die Qualität einer Strategie wird über einen Vergleich aller Individuen, die ein Operator (Operator steht hier gleichbedeutend für Strategie) erzeugt hat, zum besten Individuum gemessen. Alternativ dazu kann der Erfolg durch einen Vergleich mit den jeweiligen Eltern ermittelt werden. Das Qualitätsmerkmal beinhaltet auch die Qualität des letzten Wettbewerbs (und damit rekursiv auch die aller vorangegangenen). In jedem Wettbewerb werden ausgehend von den Qualitätsmerkmalen der Operatoren Wahrscheinlichkeiten berechnet, mit denen die Operatoren in den folgenden Generationen zum Einsatz kommen. Wettbewerbe finden zu definierten Zeitpunkten innerhalb der Optimierung statt. Mit Hilfe dieser Methode optimierten die Autoren Strukturen von Neuronalen Netzen. Dabei stellten sie fest, daß dieser Ansatz signifikante Vorteile zur Strukturoptimierung ohne Operatoradaption bringt.

2.2 Ablauf der Konkurrenzfunktion

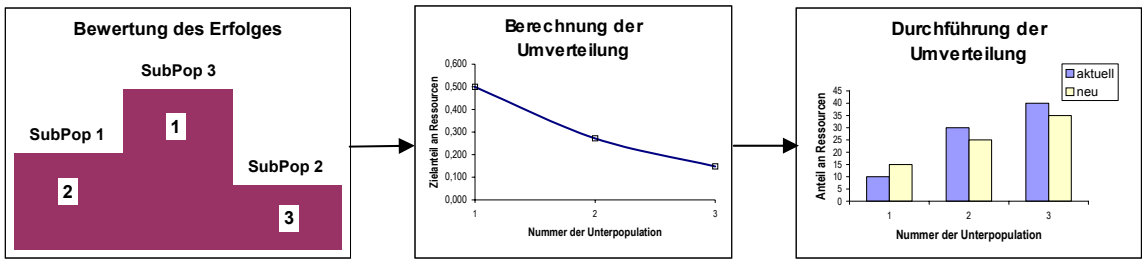

Abb. 2-1 Die drei Schritte der Konkurrenzfunktion

2.2.1 Erfolgsbewertung

Den Erfolg zu bewerten, ist keine triviale Sache. Riskiert man einen Blick in die Natur, so läßt sich erkennen, daß die Art erfolgreich ist, die es versteht zu überleben. Doch wovon ist ihr Erfolg abhängig? An erster Stelle natürlich von ihrer Überlebensfähigkeit und damit der Anpassung an ihre Umgebung. Wie gut eine Art an die Umgebung angepaßt ist, hängt von mehreren Faktoren ab. Ein Faktor ist die Nutzung von Ressourcen. Mehr Ressourcen bedeuten stärkere Fortpflanzung. Ein zweiter Faktor ist das Durchsetzungsvermögen gegenüber Konkurrenten oder Schutzmechanismen gegenüber Feinden. Ein weiterer Faktor ist, wie sie vielleicht Nutzen aus anderen Arten zieht (der auch gegenseitig sein kann). Um so besser eine Art in diesen Faktoren ist, um so größer ist auch ihr Erfolg. Folglich wird nach einer Größe gesucht, welche die Anpassung einer Unterpopulation kennzeichnet, einer Größe, die abhängig von der Qualität ihrer Individuen ist.

Für Evolutionäre Algorithmen gibt es nun mehrere Ansätze, Erfolg zu bewerten.

Ein relativ einfacher Ansatz kommt von D. SCHLIERKAMP-VOOSEN und H. MÜHLENBEIN [SVM94], siehe Abschnitt 2.1. Für die Betrachtung des Erfolges wird jeweils nur das beste Individuum der Gesamtpopulation herangezogen. Nur die Unterpopulation, die das beste Individuum enthält wird als gut bewertet, alle anderen als schlecht.

Ein weiterer Ansatz ist, daß ein Ranking zwischen den jeweils besten Individuen aller Unterpopulationen stattfindet. Die Ordnung dieser besten Individuen gibt auch die Ordnung der Unterpopulationen wieder.

Diese beiden genannten Varianten beurteilen die Qualität der Unterpopulationen nur unzureichend.

Ein Ansatz, der den Erfolg einer Unterpopulation von allen Individuen abhängig macht und daher auch einen genaueren Aufschluß über deren Qualität gibt, wurde in [Poh98] vorgestellt, siehe Abschnitt 2.1. Alle Individuen der Gesamtpopulation werden einem Ranking unterzogen, das für alle Individuen Fitneßwerte errechnet. Der aktuelle Rang einer jeweiligen Unterpopulation wird nun aus dem gemittelten Fitneßwert ihrer Individuen in der Gesamtpopulation berechnet. Der Vorteil liegt darin, daß z. B. eine Gruppe mit vielen guten Individuen besser bewertet wird als eine Gruppe mit einem sehr guten Individuum, aber vielen schlechten.

Da die Rangfolge der Unterpopulationen, z. B. aufgrund ähnlich guter Unterpopulationen stark schwanken kann, wird ein Filter eingeführt. Dieses Filter, welches die vorangegangene Rangverteilung berücksichtigt, verhindert diesen Flimmereffekt:

$$\begin{aligned} &Positionswert(n) = 0{,}9 \cdot Positionswert(n-1) + 0{,}1 \cdot Rangwert(n) \\ &n : Generation \end{aligned} \tag{2-1}$$

Der letztendliche gefilterte Rang ergibt sich nun aus der Reihenfolge der Positionswerte.

2.2.2 Ressourcenzuteilung (resource division)

Auch für die Ressourcenzuteilung sollte man einen Blick in die Natur wagen. Jede Art geht anders mit den ihr zur Verfügung stehenden Ressourcen um. Man kann also mit einer bestimmten Menge einer Ressource wesentlich mehr Individuen einer Art am Leben erhalten als Individuen einer anderen Art. Für die Optimierung bedeutet das die Einführung eines Verbrauchsfaktors und normierter Populationsgrößen, siehe Abschnitt 2.1

Für die konkrete Zuteilung der Ressourcen existieren bisher nur sehr einfache Verfahren. Das einfachste ist sicherlich, nur der besten Population Ressourcen auf Kosten der schlechteren Populationen zuzuteilen, siehe Abschnitt 2.1.

Etwas differenzierter ist der Ansatz, daß ein Teil der Unterpopulationen als erfolgreich angesehen wird, der Rest als erfolglos. Die erfolglosen geben einen bestimmten Prozentsatz ihrer Ressourcen an die erfolgreichen ab. So haben auch Populationen eine Chance, mehr Ressourcen zu erlangen, die nur wenig schlechter als die führende Population sind. Der Nachteil dieser Lösung ist, daß innerhalb der erfolgreichen Unterpopulationen alle gleichgestellt sind, wie auch innerhalb der erfolglosen. Dieses Verfahren schließt das vorher genannte Verfahren ein, indem der Prozentsatz der erfolgreichen Unterpopulationen entsprechend so gewählt wird, daß nur die beste bevorzugt wird.

Um eine gerechtere Zuteilung zu erreichen, schlägt H. POHLHEIM [Poh98] eine Verteilungsfunktion abhängig vom Rang der Unterpopulation vor. Über einen Parameter *Verteilungsdruck* (*DP*) kann die Bevorzugung der besseren Populationen gegenüber den schlechteren eingestellt werden. Je größer er gewählt wird, um so stärker werden ihre Anteile angehoben. Zum Beispiel bekommt die beste Unterpopulation bei einer Gesamtzahl von 8 Unterpopulationen und einem Verteilungsdruck von 4 einen Ressourcenanteil von 50%. Die restlichen 50% werden unter den anderen Unterpopulationen aufgeteilt, zu den Anteilen 25,1%, 12,6%, 6,33% usw., siehe Abbildung 2-2.

Lineare Verteilung erlaubt nur einen maximalen Verteilungsdruck von 2. Daher wird, wenn ein größerer Verteilungsdruck gewünscht wird, die nichtlineare Verteilung benutzt. Diese läßt einen maximalen Verteilungsdruck von *Anzahl der Unterpopulationen - 2* zu. Für obiges Beispiel ist der maximale Verteilungsdruck also 6. Ein weiterer Unterschied dieser beiden Methoden ist, daß mit linearer Verteilung und einem *Verteilungsdruck* von 2 theoretisch ein Anteil von 0 für die schlechteste Unterpopulation eingestellt werden kann (bei nichtlinearer Verteilung nicht möglich). Damit würde ein Aussterben dieser Unterpopulation ermöglicht werden.

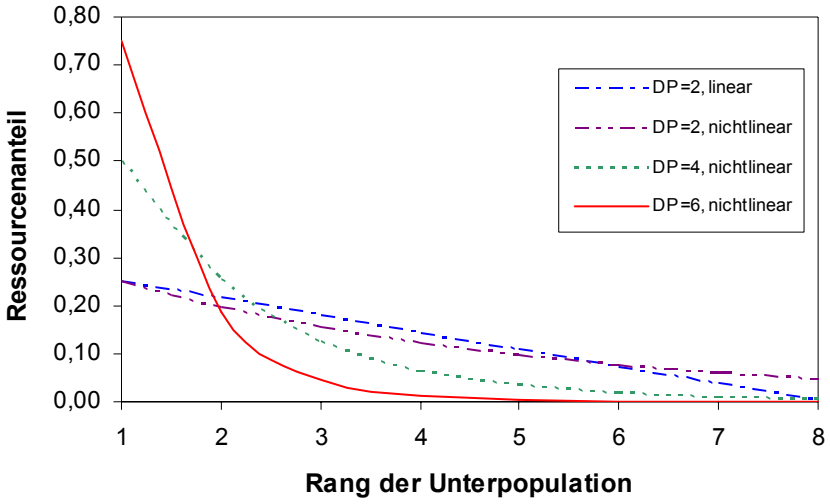

Abb. 2-2 Verteilungsfunktionen bei verschiedenen Verteilungsdrücken für 8 Unterpopulationen; die Verteilungsfunktion bestimmt den Anteil der Unterpopulationen an den Ressourcen abhängig von ihrem Rang. (Deutlich erkennbar ist der geringe Unterschied zwischen linearer und nichtlinearer Verteilung bei $DP = 2$)

Die beiden Varianten der Verteilungsfunktion lassen sich wie folgt berechnen:

Lineare Verteilung:

$$Anteil(Rang) = 2 - DP + 2 \cdot (DP - 1) \cdot \frac{NSubPop - Rang}{NSubPop} \qquad (2\text{-}2)$$

$$DP \in [1{,}0;2{,}0]$$

Nichtlineare Verteilung:

$$Anteil(Rang) = \frac{NSubPop \cdot X^{NSubPop - Rang}}{\sum_{i=1}^{NSubPop} X^{i-1}} \qquad (2\text{-}3)$$

X kann bei Angabe des Verteilungsdruckes *DP* und der Anzahl der Unterpopulationen *NSubPop* als Lösung des folgenden Polynoms berechnet werden:

$$\begin{aligned} 0 &= (DP - NSubPop) \cdot X^{NSubPop-1} + DP \cdot X^{NSubPop-2} + \ldots + DP \cdot X + DP \\ DP &\in [1{,}0; NSubPop - 2] \end{aligned} \tag{2-4}$$

2.2.3 Ressourcenumverteilung (resource distribution)

Die Ressourcenumverteilung ist die Durchführung der Ressourcenzuteilung. Zwei Parameter, die für die Ressourcenumverteilung von Bedeutung sind, heißen *Konkurrenzrate* und *Unterpopulationsminimum*.

Die *Konkurrenzrate* sorgt dafür, daß sich die berechnete Ressourcenzuteilung gebremst einstellt. Der Sinn liegt darin, daß sich bei kurzzeitiger sprunghafter Verbesserung einer Gruppe die Größe dieser Gruppe nicht zu schnell ändern soll. Die *Konkurrenzrate* kann daher als Filter gedeutet werden. Im Zusammenhang mit der Filterung der Rangverteilung stellt sie eine neuerliche Filterung dar. Während die Filterung der Rangverteilung die Änderung der Ränge bremst, beeinflußt die *Konkurrenzrate*, wie schnell die Unterpopulationsgrößen der veränderten Rangverteilung folgen. Beide Filterungen erzielen einen ähnlichen Effekt, aber im Zusammenspiel läßt sich eine bessere Steuerung erreichen.

Das *Unterpopulationsminimum* sorgt dafür, daß erfolglose Gruppen und damit ihre Strategie nicht komplett verschwinden können. Wenn eine Unterpopulation diesen Wert erreicht hat, gibt sie keine weiteren Individuen mehr ab.

Diese beiden Parameter, *Konkurrenzrate* und *Unterpopulationsminimum,* beschränken zusammen mit der Verteilungsfunktion den Anteil der Individuen, welche die überbesetzten Unterpopulationen abzugeben haben. Dieser Anteil darf maximal so groß sein wie die Differenz aus aktueller Verteilung und der über den *Verteilungsdruck* berechneten Verteilung. Andererseits darf er den von der *Konkurrenzrate* bestimmten Prozentsatz nicht überschreiten. Letztlich darf das *Unterpopulationsminimum* nicht unterschritten werden.

Es sollte erwähnt werden, daß auch gute Populationen Individuen abgeben müssen, wenn ihr bisheriger Ressourcenanteil höher als der neu berechnete liegt.

Die Anzahl an Individuen, die Gruppen mit Bedarf hinzugefügt wird, berechnet sich anteilsmäßig aus der Anzahl der abgegebenen Individuen. Der Anteil bestimmt sich aus der Anzahl der Individuen, die laut Verteilungsfunktion von den unterbesetzten Gruppen aufgenommen werden sollen. Wenn sich der Rang der Unterpopulationen über einen längeren Zeitraum nicht ändert, nähert sich die Ressourcenverteilung der vorgegebenen Verteilung an.

Die Funktionsweise des Hinzufügens und Abgebens von Individuen soll in einem Beispiel näher erläutert werden. Das Beispiel ist einem reellen Optimierungslauf entnommen und stellt die erste Generation dar, während derer die Unterschiede zwischen aktueller Verteilung und Zielverteilung groß sind. Im weiteren Verlauf einer Optimierung sind diese Unterschiede dann natürlich nicht mehr so groß. In Abbildung 2-3 werden aktuelle Verteilung und Zielverteilung dargestellt. Ist die Differenz zwischen Zielverteilung und aktueller Verteilung einer Unterpopulation negativ, so muß diese Individuen abgeben. Bei positiver Differenz verhält es sich umgekehrt. Das heißt also, daß Unterpopulation 1 Individuum aufnimmt, Unterpopulation 3 Individuen abgibt usw.

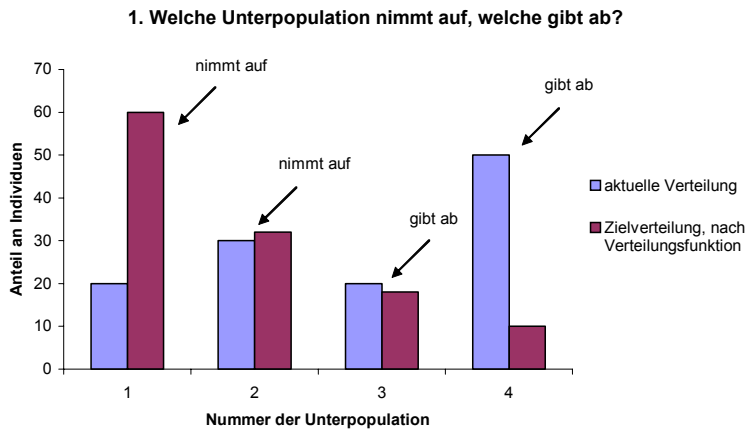

Abb. 2-3 Beispiel der Funktionsweise des Hinzufügens und Abgebens von Individuen:
Die Differenz zwischen aktueller Verteilung und Zielverteilung ist Maß dafür, ob Ressourcen abgegeben oder aufgenommen werden

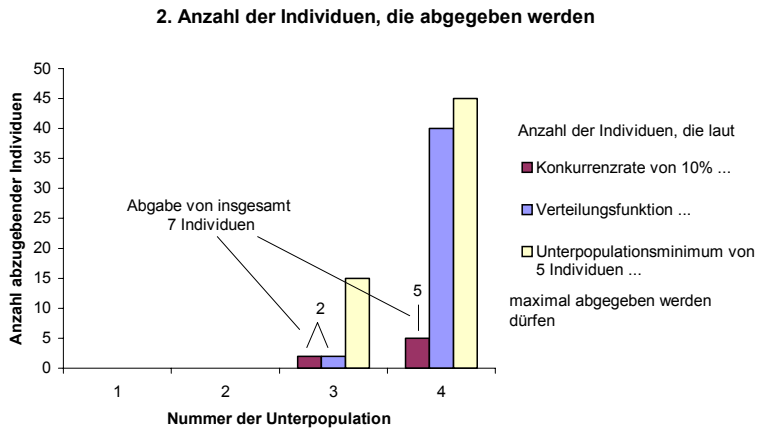

Abb. 2-4 Beispiel der Funktionsweise des Hinzufügens und Abgebens von Individuen:
Stellt obere Begrenzungen für die Anzahl abzugebender Individuen dar; das Minimum dieser wird verwendet

Abbildung 2-4 zeigt die Begrenzungen, nach denen die maximal abzugebende Zahl der Individuen für die betreffenden Unterpopulationen ermittelt wird. Drei Begrenzungen sind ausschlaggebend:

- Die durch die *Konkurrenzrate* bestimmte Begrenzung. Hier beträgt die *Konkurrenzrate* 10%, das bedeutet für Unterpopulation 3 eine Zahl von 2 Individuen und für Unterpopulation 4 eine Zahl von 5 Individuen.
- Die Differenz aus aktueller Verteilung und Zielverteilung. Das heißt für Unterpopulation 3 wieder 2 Individuen und für Unterpopulation 4 eine Zahl von 40 abzugebenden Individuen.
- Die maximale Zahl abzugebender Individuen laut *Unterpopulationsminimum*. Da mindestens 5 Individuen übrig bleiben sollen, kann Unterpopulation 3 also 15 Individuen abgeben und Unterpopulation 4 kann 45 Individuen abgeben.

Das Minimum dieser Maximalzahlen wird hier für beide betreffende Unterpopulationen durch die *Konkurrenzrate* bestimmt (für Unterpopulation 3 auch durch die Verteilungsfunktion). Insgesamt werden also 7 Individuen abgegeben.

Aus Abbildung 2-5 wird ersichtlich, daß von Unterpopulation 1 und 2 insgesamt 42 Individuen aufgenommen werden sollen (SP1 – 40, SP2 – 2 Individuen). Da aber nur 7 Individuen verteilt werden können, muß das verhältnismäßig geschehen. Diese werden also im Verhältnis 40:2 aufgeteilt. Für Unterpopulation 1 bedeutet das 6 Individuen, und für Unterpopulation 2 bedeutet das 1 Individuum. Eigentlich hätte Unterpopulation 2 bei dieser Verteilung nur den Bruchteil eines Individuums bekommen. Es wird aber sichergestellt, daß wenigstens 1 Individuum aufgenommen wird.

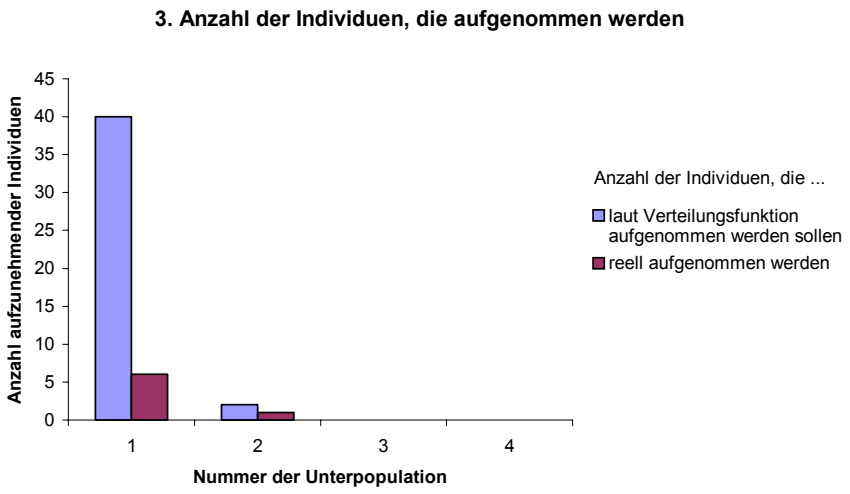

Abb. 2-5 Beispiel der Funktionsweise des Hinzufügens und Abgebens von Individuen;
42 Individuen sollen insgesamt aufgenommen werden, aber 7 stehen nur zur Verfügung → die 7 Individuen werden verhältnismäßig verteilt

Die Selektion der abzugebenden Individuen kann auf verschiedene Weise erfolgen:

- anhand der besten Individuen,
- anhand zufällig ausgewählter Individuen,
- anhand der schlechtesten Individuen.

Die besten Individuen zu wählen, würde eine über die Abgabe von Individuen hinaus weitere Benachteiligung der ohnehin schon schlechten Unterpopulationen bedeuten. Die zufällige Selektion nutzt keine Wertung der Individuen, kann also gute wie auch schlechte Individuen ersetzen. Bei der Auswahl der schlechtesten Individuen wäre die Benachteiligung gering. Das ist daher die Methode der Wahl.

Eine weitere wichtige Größe ist das *Konkurrenzintervall*. Es gibt an, wie oft innerhalb eines Optimierungslaufes ein Wettbewerb stattfindet, also wie lange sich Unterpopulationen ohne Veränderung der ihnen zustehenden Ressourcen entwickeln. Alternativ dazu kann ein Wettbewerb aber auch eingeleitet werden, wenn in einer Population ein deutlicher Fortschritt erkennbar ist.

Im Zusammenhang mit der *Konkurrenzrate* ergibt sich die Umverteilungsgeschwindigkeit:

$$Umverteilungsgeschwindigkeit = \frac{Konkurrenzrate}{Konkurrenzintervall} \tag{2-5}$$

Es handelt sich hierbei allerdings nur um die maximal mögliche Umverteilungsgeschwindigkeit, da der Anteil der umverteilten Individuen natürlich auch durch die Verteilungsfunktion begrenzt sein kann.

Nachdem in diesem Kapitel der Vorgang der Konkurrenz theoretisch erläutert wurde (in welche Schritte sie unterteilt ist, und von welchen Parametern sie beeinflußt wird) werden in Kapitel 3 Ergebnisse von Experimenten gezeigt, die sich mit den Effekten dieser Parameter und der Konkurrenz beschäftigen.

3 Ergebnisse

Dieses Kapitel beschäftigt sich mit einigen in Kapitel 2 eingeführten Größen und den Effekten der Konkurrenz in experimentellen Untersuchungen. Dabei werden die Umverteilungsgeschwindigkeit und der Verteilungsdruck untersucht (Abschnitt 3.1). Weiterhin wird der Frage nachgegangen, ob trotz Konkurrenz zwischen den Unterpopulationen Kooperation auftritt (Abschnitt 3.2).

3.1 Umverteilungsgeschwindigkeit

Zur Untersuchung der Umverteilungsgeschwindigkeit wurden verschiedene Optimierungsläufe mit der Genetic and Evolutionary Algorithm Toolbox for use with MATLAB®, kurz [GEATbx] durchgeführt. Alle Läufe arbeiteten mit 4 Unterpopulationen, Startverteilung jeweils 100 Individuen pro Unterpopulation, linearem Verteilungsdruck von 2. Die laut Verteilungsfunktion erreichbaren Unterpopulationsgrößen sind von Rang 1 nach Rang 4: 201, 108, 59 und 32 Individuen (kleinerer Rang bedeutet höheren Erfolg).

In den folgenden Diagrammen sind die Größe der Unterpopulationen und ihre Ordnung über die Generationen dargestellt. Die Ordnungen der Unterpopulationen sind hier zwar als kontinuierliche Größen dargestellt, entscheidend für die Ressourcenverteilung ist aber nicht der Abstand der Linien sondern ihre Reihenfolge. An diesen Diagrammen läßt sich gut veranschaulichen, mit welcher Geschwindigkeit die Verteilung einem Rangwechsel folgt.

3.1.1 *Konkurrenzintervall* von 5 Generationen - verschiedene Konkurrenzraten

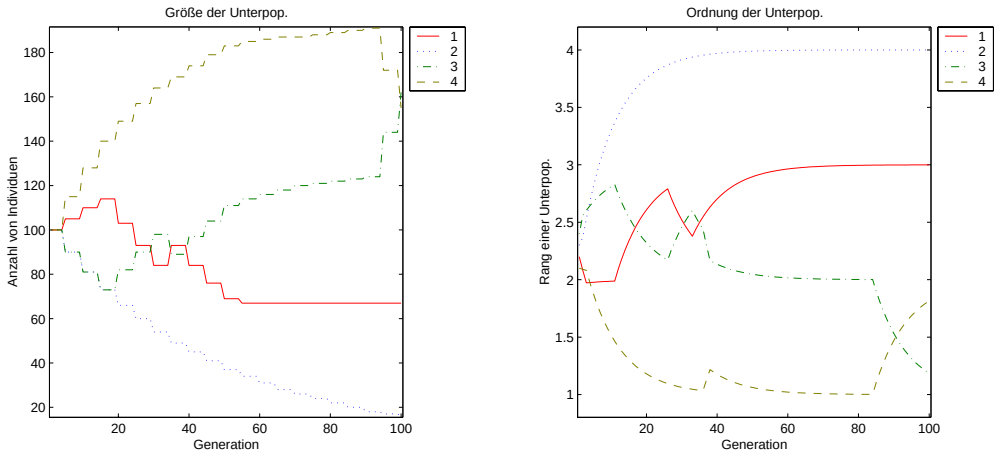

Abb. 3-1 *Konkurrenzintervall*: 5 Generationen, *Konkurrenzrate*: 10%

Eine Einstellung der *Konkurrenzrate* von 10% besagt, daß eine Unterpopulation maximal 10% der derzeitigen Anzahl ihrer Individuen an andere Unterpopulationen abgeben darf. Das bedeutet ein relativ langsames Nachführen der Verteilung auf eine Änderung in der Ordnung der Unterpopulationen. Deutlich zu erkennen ist das *Konkurrenzintervall*; alle 5 Generationen treten sichtbare Sprünge in den Größen auf (Abb. 3-1).

Der Anteil, den eine Unterpopulation abgeben muß, wird dabei von der Verteilungsfunktion und dem *Unterpopulationsminimum* eingeschränkt, was zum Beispiel den sehr kleinen Sprung der Größenlinie 1 von Generation 50 zu Generation 55 (der letzte Sprung) erklärt. Ein Sprung von 10% wäre merklich größer (Abb. 3-1).

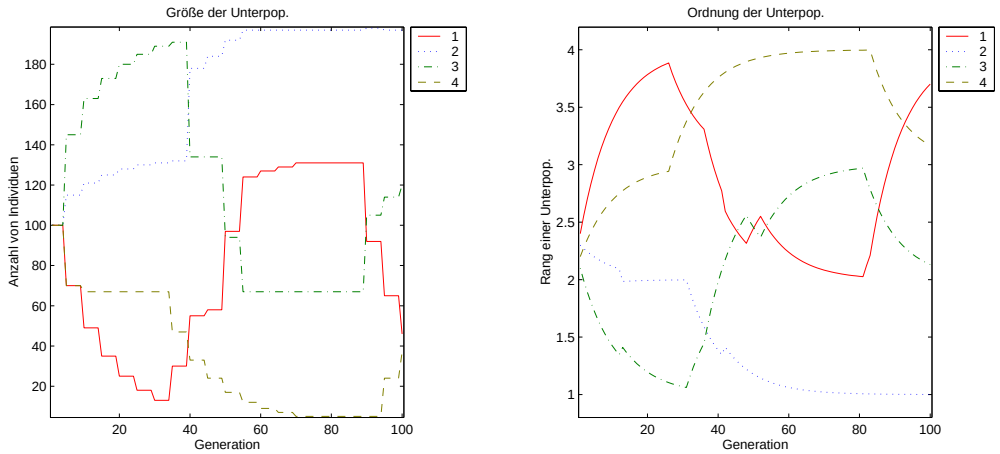

Abb. 3-2 *Konkurrenzintervall*: 5 Generationen, *Konkurrenzrate*: 30%

Im Vergleich zur Einstellung einer *Konkurrenzrate* von 10% wird durch eine Rate von 30% eine erkennbar schnellere Umverteilung erreicht. Die maximalen Sprünge pro Generation fallen in der Größe höher aus (Abb. 3-2).

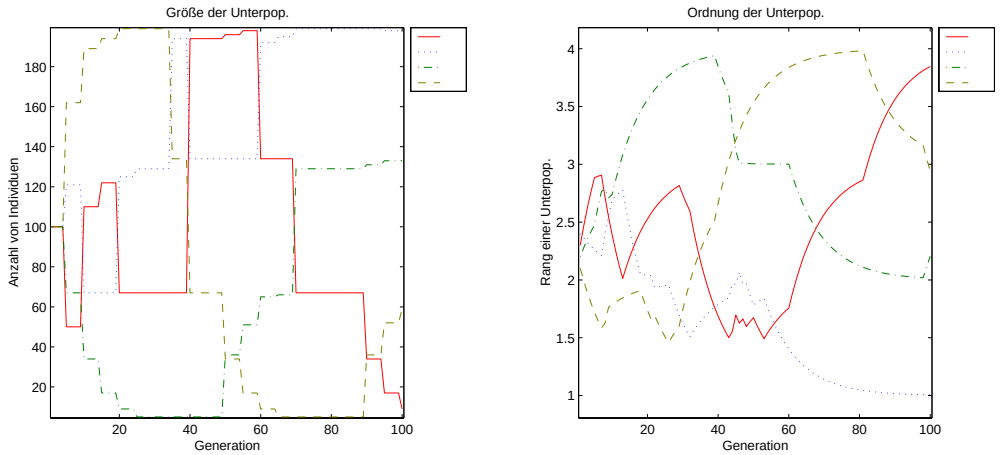

Abb. 3-3 *Konkurrenzintervall*: 5 Generationen, *Konkurrenzrate*: 50%

In Abbildung 3-3 ist gut der Zusammenhang der beiden dargestellten Diagramme zu sehen. Beispielsweise läßt sich in der rechten Grafik ein Rangwechsel der ersten und der vierten Unterpopulation kurz vor Generation 70 erkennen. Da nur alle 5 Generationen ein Wettbewerb stattfindet, kann erst in Generation 70 eine Reaktion erfolgen, die auch sehr heftig erfolgt (linke Grafik). Das

ist auf die *Konkurrenzrate* von 50% zurückzuführen, die gegenüber einer Rate von 10% eine wesentlich höhere Umverteilungsgeschwindigkeit nach sich zieht.

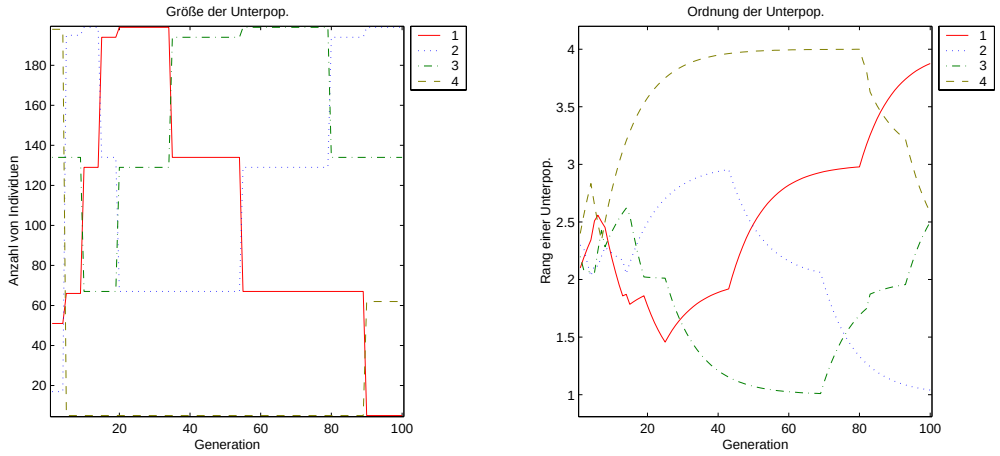

Abb. 3-4 *Konkurrenzintervall*: 5 Generationen, *Konkurrenzrate*: 100%

Eine *Konkurrenzrate* von 100% ist die maximale Einstellung; das bedeutet ein sofortiges Nachführen (d. h. zur nächsten durch das *Konkurrenzintervall* teilbaren Generation) der laut Verteilungsfunktion berechneten Umverteilung (Abb. 3-4). Es findet keinerlei Filterung durch die *Konkurrenzrate* statt.

3.1.2 *Konkurrenzintervall* von 10 Generationen - verschiedene Konkurrenzraten

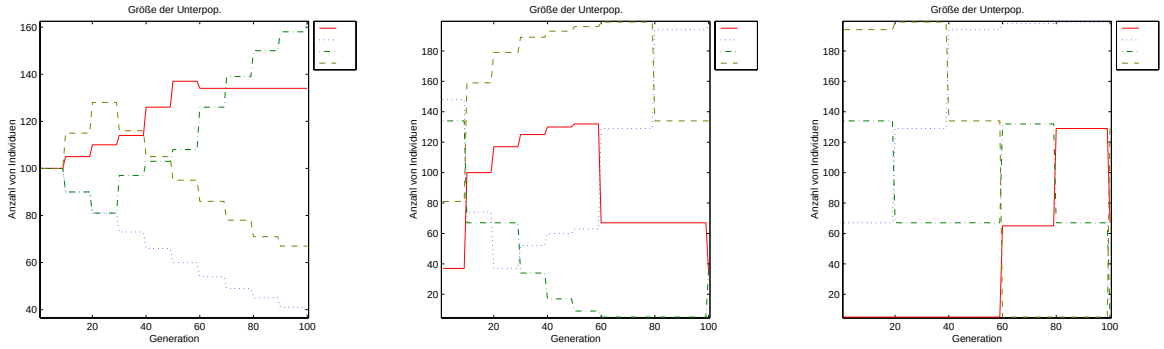

Abb. 3-5 *Konkurrenzintervall*: 10 Generationen, Konkurrenzraten von links nach rechts: 10%, 50%, 100%

Durch das höhere *Konkurrenzintervall* von 10 ergeben sich im Vergleich zu Abschnitt 3.1.1 trägere Umverteilungen (halbe Geschwindigkeit). Das Intervall ist auch hier gut ablesbar, die Sprünge finden alle 10 Generationen statt. Für die Auswirkung der *Konkurrenzrate* treffen hier die in Abschnitt 3.1.1 gemachten Aussagen zu.

Die maximale Umverteilungsgeschwindigkeit ist in den in Abbildung 3-6 gegenübergestellten Grafiken gleich groß, der Quotient aus *Konkurrenzrate* und *Konkurrenzintervall* beträgt jeweils 0,02. Das drückt sich auch in ähnlichen Anstiegen der Größenlinien der Unterpopulationen aus, gut sichtbar in den Linien 2 (linke Grafik) und 4 (rechte Grafik). Wofür die Größenlinie in der linken Grafik zwei Schritte macht, benötigt die Größenlinie in der rechten Grafik nur einen

Schritt. Natürlich sind die Abstufungen in der linken Grafik aufgrund des geringeren Konkurrenzintervalls feiner.

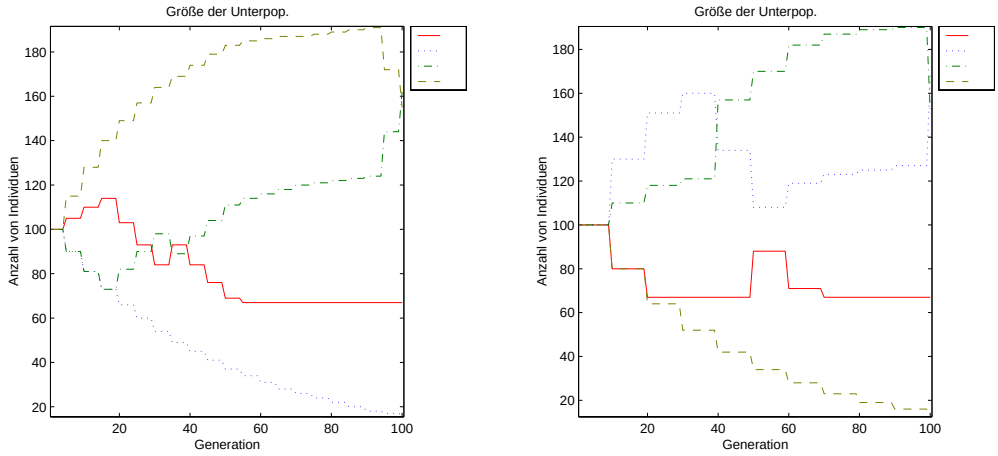

Abb. 3-6 Vergleich: verschiedene Parameter aber gleiche maximale Umverteilungsgeschwindigkeiten links: *Konkurrenzintervall* 5, *Konkurrenzrate* 10%; rechts: *Konkurrenzintervall* 10, *Konkurrenzrate* 20%

3.2 Verteilungsdruck

Zur Verdeutlichung des Einflusses des Verteilungsdruckes wurden drei Optimierungsläufe durchgeführt. Sie liefen mit acht Unterpopulationen und Verteilungsdrücken von 2, 4 und 6. Letzterer ist die maximal mögliche Einstellung bei nichtlinearer Verteilung und acht Unterpopulationen, siehe Gleichung (2-3). Die [GEATbx] nutzt bei einem Verteilungsdruck von 2 automatisch lineare Verteilung; die Unterschiede zu nichtlinearer Verteilung bei dieser Einstellung sind aber so gering, daß sie anstatt dieser zu einem Vergleich herangezogen werden kann, siehe Abbildung 2-2.

Ein höherer Verteilungsdruck bedeutet eine stärkere Bevorzugung der besseren Unterpopulationen. Das wird in Abbildung 2-2 deutlich. Während $DP = 4$ einen Anteil von 50% bzw. $DP = 6$ einen Anteil von 75% für die beste Unterpopulation erzielt, bringt $DP = 2$ nur einen Anteil von 25%.

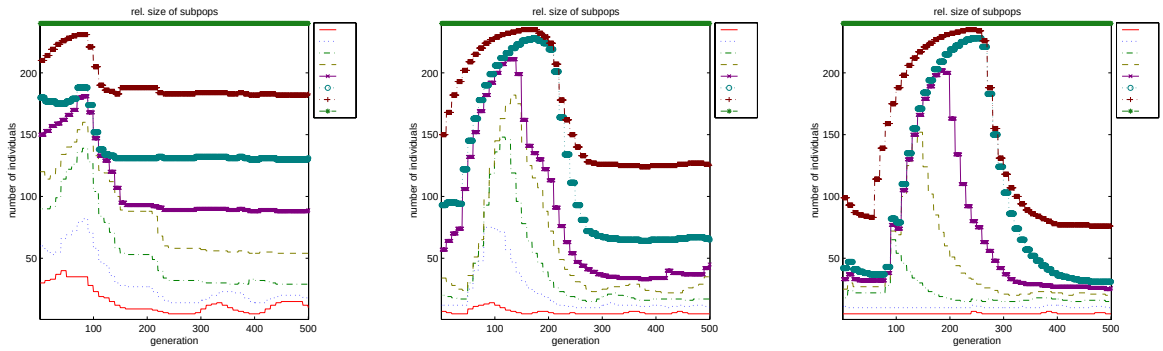

Abb. 3-7 Vergleich verschiedener Verteilungsdrücke: links: $DP = 2$; Mitte: $DP = 4$; rechts: $DP = 6$

Abbildung 3-7 zeigt die relative Größe der Unterpopulationen über den Optimierungslauf bei den genannten Einstellungen. D. h., daß die Größe der Unterpopulationen gleichbedeutend mit den Abständen zwischen den Kurven ist. Auch hier ist der Einfluß des Parameters gut erkennbar. So treten in den beiden rechten Grafiken deutlich größere Abstände zwischen Größenlinien auf als in der linken Grafik, während sich gleichzeitig die restlichen Größenlinien dichter drängen.

3.3 Kooperation

Eine Aufgabenstellung dieser Arbeit ist es, zu untersuchen, ob neben der Konkurrenz zwischen den Unterpopulationen auch Kooperation unter diesen auftritt. Um das zu überprüfen, wurden drei verschiedene Strategien ausgewählt, die sich in ihren Mutationsbereichen signifikant unterscheiden. Als zu optimierende Funktion wird einerseits die DE JONG Funktion 1 [DeJ75] verwendet, andererseits die RASTRIGIN-Funktion. Die DE JONG Funktion 1 besitzt nur ein Minimum, das im Koordinatenursprung liegt. Die RASTRIGIN-Funktion basiert auf der DE JONG Funktion 1. Diese ist mit einer Cosinusfunktion moduliert. Daher besitzt sie im Gegensatz dazu sehr viele lokale Minima, die dicht beieinander liegen. Das absolute Minimum liegt ebenfalls im Koordinatenursprung. Beide Funktionen werden mit 30 Dimensionen berechnet.

Es ist zu erwarten, daß Strategien mit sehr kleinen Mutationsbereichen in den vielen lokalen Minima der RASTRIGIN-Funktion steckenbleiben werden, während Strategien mit großen Schrittweiten, nicht genügend kleine Schritte machen können, um die Umgebung eines dieser lokalen Minima genauer zu untersuchen.

Im Verbund dagegen sollten die Strategien, wenn Kooperation auftritt, genauere Werte finden (diese Erwartung gilt auch für die DE JONG Funktion 1) oder überhaupt erst in die Umgebung des Minimums kommen.

Ein Verbund verschiedener Strategien wird hier auf zweierlei Art realisiert:

- Erstens über den Einsatz von Migration.
- Zweitens über den Einsatz von Konkurrenz zusätzlich zur Migration.

Migration bedeutet regelmäßigen Informationsaustausch zwischen den Strategien und birgt damit die Erwartung von Kooperation zwischen den Strategien. Migration in Kombination mit Konkurrenz wirft die Frage auf, ob die Kooperation durch Konkurrenz zunichte gemacht wird, oder ob eine Art nützliches Wechselspiel zwischen Kooperation und Konkurrenz zustande kommt. Die Hoffnung ist, daß sich dadurch die Werte weiter verbessern. Wenn im folgenden von Migration gesprochen wird, ist Migration allein gemeint. Wenn von Konkurrenz gesprochen wird, schließt das immer Migration mit ein.

Die Strategien müssen sich anfangs einzeln an den Optimierungsproblemen versuchen, dann in den oben genannten Verbundmöglichkeiten. Die Mutationsbereiche der drei Strategien betragen 0.1, 0.002 und 10^{-5}. Die Mutationspräzision wird auf jeweils 4 festgelegt. Zur Bedeutung dieser Parameter sei auf [GEATbx] verwiesen. Die Anzahl der Individuen beträgt für jeden Lauf insgesamt 90 Individuen. Für die Läufe mit Migration und Konkurrenz werden sie zu jeweils 30 Individuen auf die 3 Unterpopulationen verteilt. Bei der Einstellung mit Konkurrenz stellt das natürlich nur die Anfangsverteilung dar.

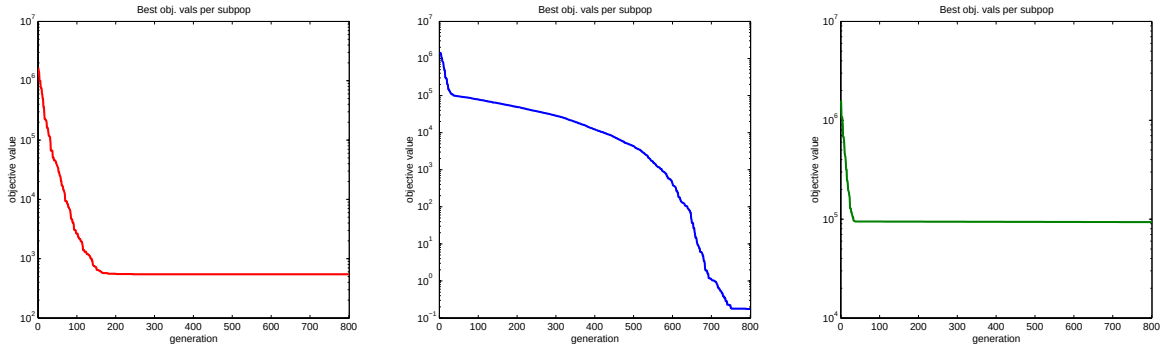

Abb. 3-8 Optimierung der DE JONG Funktion 1, die drei Strategien im Einzellauf, Mutationsbereiche von links nach rechts: 0.1, 0.002, 10^{-5}

Wie in Abbildung 3-8 zu erkennen ist, schafft es nur die Strategie mit dem mittleren Mutationsbereich von 0.002 (mittlere Grafik, Abb. 3-8) in die nähere Umgebung des Minimums. Nach einem großen Sprung am Anfang ist der Fortschritt dieser Strategie fast linear bis zum Ende. Das läßt sich allerdings in dieser Grafik aufgrund der logarithmischen Skalierung schwer erkennen.

Die Strategie mit dem großen Mutationsbereich (linke Grafik, Abb. 3-8) kommt anfangs gut vorwärts, bleibt aber etwa nach der 250. Generation bei einem Zielfunktionswert von ungefähr 550 stehen. Ihre Schrittweite ist zu groß, um in die nähere Umgebung des Maximums zu kommen.

Die Strategie mit kleinem Mutationsbereich (rechte Grafik, Abb. 3-8) kommt so gut wie gar nicht voran. Auch hier tritt innerhalb der ersten Generationen ein großer Sprung auf, ähnlich dem bei Einstellung mit mittlerer Schrittweite. Er läßt sich damit erklären, daß die Individuen am Anfang noch sehr zufällig verteilt sind. Durch Rekombination dieser sehr unterschiedlichen Individuen werden noch große Fortschritte erreicht, die dann aber ab etwa Generation 30 nahezu vollständig nachlassen. Die Individuen werden sich aufgrund des geringen Mutationsbereiches zunehmend ähnlicher, so daß ab Generation 30 auch durch Rekombination keine großen Veränderungen der Individuen mehr stattfinden.

Abbildung 3-9 zeigt, wie die Strategien zusammenarbeiten. In diesem Fall kommt nur Migration zum Einsatz. Das Migrationsintervall von 40 Generationen läßt sich an den Sprüngen in den Zielfunktionswerten gut ablesen. Erkennbar ist, daß es zu einer Ablösung zwischen den mit verschiedenen Strategien arbeitenden Unterpopulationen kommt. Jede der Strategien „grast“ unterschiedliche Suchräume der zu optimierenden Funktion ab, und zwar nacheinander. D. h., jede Strategie wird von ihrem Vorgänger in die richtige Position gesetzt, um wirksam sein zu können. Strategie 1 ist für das „Grobe“ zuständig, während Strategie 2 im Nahbereich genauer zu Werke geht und Strategie 3 die Feinarbeit leistet. Das wird deutlich, wenn man die beiden Diagramme in Abbildung 3-9 in Kombination betrachtet. Anfangs hat Strategie 1 den größten Erfolg (und daher den kleinsten Rang!). Bis ungefähr zur 150. Generation bleibt sie in dieser Position und hat sich bis zum Zielfunktionswert von 1000 vorgearbeitet. Dann wird sie von Strategie 2 abgelöst. Im rechten Diagramm wird der Rangwechsel angezeigt, während im linken Diagramm (Zielfunktionswerte) Strategie 1 nun langsam zu Sprüngen (immer zum Migrationszeitpunkt) neigt. Dies ist ein Zeichen dafür, daß sie selbst keine weiteren Verbesserungen finden kann. Der Übergang erfolgt fließend. Strategie 2 besitzt bis etwa Generation 380 den besten Rang. Der Zielfunktionswert, den sie erreicht, liegt um 1. Danach kommt Strategie 3 mit dem kleinsten Mutationsbereich zum Zu-

ge. Insgesamt erreicht die Optimierung einen Zielfunktionswert von ca. 10^{-4}, der weit unter dem liegt, was die einzelnen Strategien einzeln erreicht hatten.

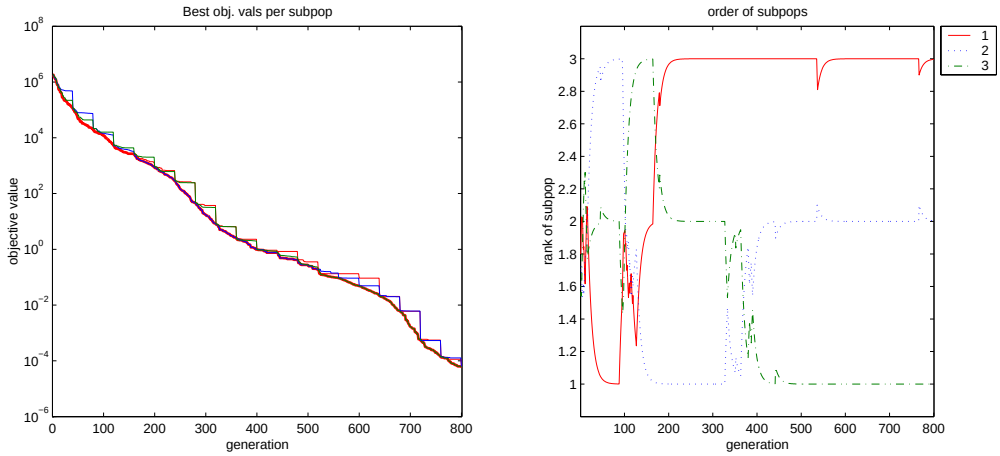

Abb. 3-9 Optimierung der DE JONG Funktion 1 – nur Migration: links: Konvergenz; rechts: Ordnung der Unterpopulationen

Ein ähnliches Bild ergibt sich für die Optimierung mit eingeschalteter Konkurrenz (Abbildung 3-10). Auch hier ist eine Ablösung zwischen den Strategien gut erkennbar. Auffällig sind die teilweise wesentlich größeren Sprünge der Zielfunktionswerte der erfolglosen Unterpopulationen im Konvergenzdiagramm. Diese sind darauf zurückzuführen, daß sie durch die Konkurrenz Individuen abgeben mußten und daher noch schlechter werden. Durch ihre geringe Größe ist die Wahrscheinlichkeit klein, daß Individuen dieser Populationen ersetzt werden. Daher treten die Sprünge seltener auf.

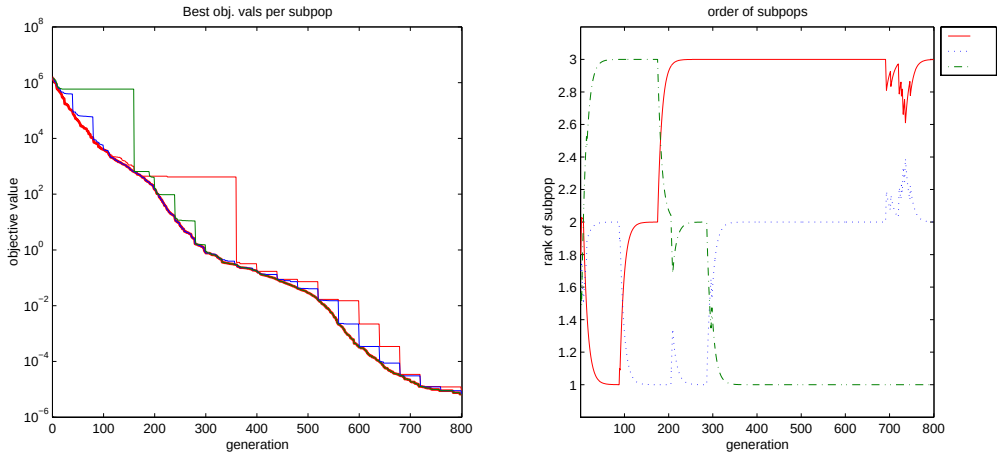

Abb. 3-10 Optimierung der DE JONG Funktion 1 – Konkurrenz und Migration: links: Konvergenz; rechts: Ordnung der Unterpopulationen

Für die Optimierung der RASTRIGIN-Funktion (Abbildung 3-11) ergibt sich insgesamt ein ähnliches Bild, verglichen mit der Optimierung der DE JONG Funktion 1. Strategie 1 und Strategie 3 erreichen im Einzellauf in etwa die gleichen Werte wie bei der Optimierung der DE JONG Funktion 1. Strategie 2 ist jedoch etwas schlechter im Vergleich. Hier erfüllt sich die Erwartung, die am Anfang dieses Kapitels gemacht wurde. Diese Strategie ist nicht in der Lage, die Umgebung des

globalen Minimums genauer zu ertasten. Strategie 1 und 3 bleiben schon viel weiter vorn stehen, wenn auch aus unterschiedlichen Gründen. Die Erklärungen, die für die Optimierung der DE JONG Funktion 1 gemacht wurden, gelten auch hier.

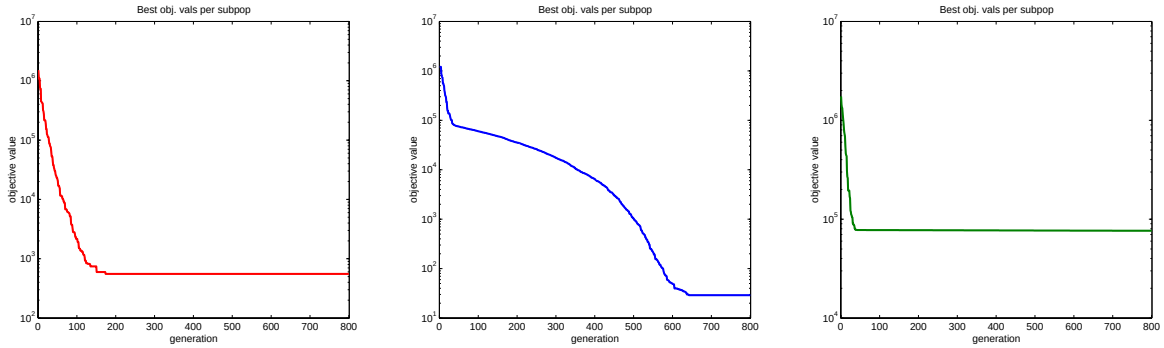

Abb. 3-11 Optimierung der RASTRIGIN-Funktion, die drei Strategien im Einzellauf, Mutationsbereiche von links nach rechts: 0.1, 0.002, 10^{-5}

Wie bei der Optimierung der DE JONG Funktion 1 kommt es in Abbildung 3-12 zu einer Kooperation zwischen den Unterpopulationen, so daß der Optimierungslauf dieser Unterpopulationen im Verbund ein wesentlich besseres Ergebnis erzielt, als diese einzeln erreichen könnten.

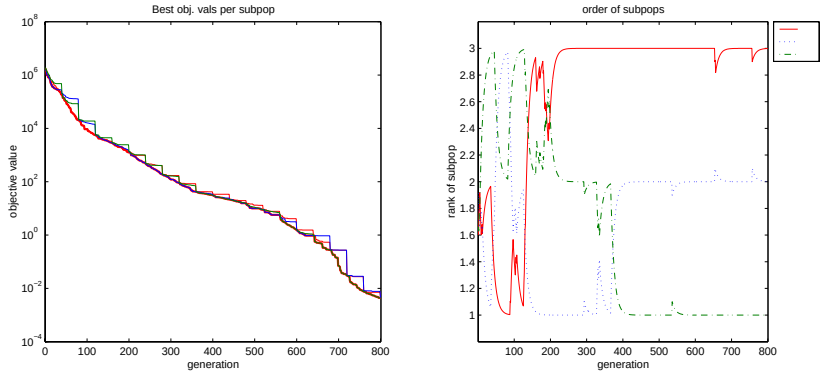

Abb. 3-12 Optimierung RASTRIGIN-Funktion – nur Migration: links: Konvergenz; rechts: Ordnung der Unterpopulationen

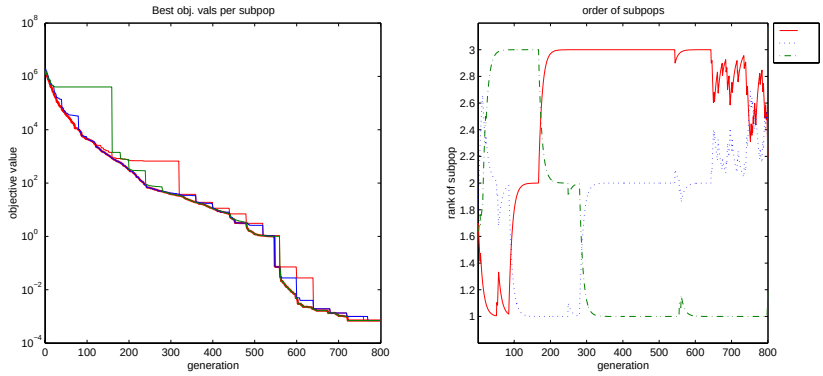

Abb. 3-13 Optimierung der RASTRIGIN-Funktion – Konkurrenz und Migration: links: Konvergenz; rechts: Ordnung der Unterpopulationen

Für Abbildung 3-13 treffen die zum Optimierungslauf der DE JONG Funktion 1 mit Konkurrenz getroffenen Aussagen (Abbildung 3-10) in gleicher Weise zu.

Um einen Vergleich zwischen Optimierung mit Migration und Optimierung mit Konkurrenz zu machen, werden Optimierungen mit obigen Einstellungen 40 mal durchgeführt und die Daten statistisch ausgewertet. Das wird für beide Zielfunktionen realisiert. Ermittelt werden die besten Zielfunktionswerte für jeden Optimierungslauf. Davon werden die Werte angegeben, die maximal 30 % vom kleinsten dieser Zielfunktionswerte abweichen. Ähnliches wird für die schlechtesten Zielfunktionswerte durchgeführt, nur daß hier diejenigen Zielfunktionswerte aufgeführt werden, die maximal 10% vom größten Zielfunktionswert abweichen. Des weiteren wird der Durchschnitt aller besten Zielfunktionswerte und deren Standardabweichung berechnet. Zuletzt wird ermittelt, in welcher Generation im Durchschnitt das beste Individuum gefunden wurde.

Die Ergebnisse fallen zugunsten der Optimierung mit Konkurrenz aus. Mit eingestellter Konkurrenz wurden absolut sowie auch im Mittel bessere Zielfunktionswerte erreicht als bei Optimierung mit Migration. Weiterhin war die Standardabweichung der Zielfunktionswerte durchweg kleiner, und der beste Zielfunktionswert wurde im Mittel schneller gefunden.

Nachfolgend sind die ermittelten Werte aufgelistet:

Tab. 3-1 Statistischer Vergleich: Migration – Konkurrenz

Optimierung mit:	**Migration**		**Konkurrenz**	
Zielfunktion:	RASTRIGIN	DE JONG1	RASTRIGIN	DE JONG1
kleinste beste ZF-Werte (maximale Abweichung vom kleinsten: 30%)	9,162E-04 1,091E-03	1,055E-05	4,599E-04 5,567E-04	3,163E-06 3,339E-06 3,349E-06 3,559E-06 3,743E-06 3,788E-06 3,946E-06 3,972E-06
größte schlechteste ZF-Werte (maximale Abweichung vom größten: 10%)	9,994E-01 9,977E-01 9,977E-01	1,004E-03	9,957E-01 9,956E-01	1,130E-04
Mittelwert der besten ZF-Werte:	8,455E-02	9,703E-05	5,090E-02	8,607E-06
Standardabweichung der besten ZF-Werte:	2,671E-01	1,654E-04	2,195E-01	1,711E-05
Im Mittel wurde das beste Individuum gefunden (Generation)	795	798	772	787

4 Implementierung

Die Konkurrenzfunktion `compete.m` ist eine Unterfunktion der [GEATbx], die in MATLAB® umgesetzt ist. In der Konkurrenzfunktion wurden die in Kapitel 2 erläuterten Ansätze implementiert. Die Steuerung des Ablaufes des Evolutionären Algorithmus innerhalb der [GEATbx] übernimmt die Funktion `geamain2.m`. Da bereits eine einfache Konkurrenzfunktion vorhanden war, brauchten keine weiteren Aufrufe in der `geamain2.m` ergänzt werden. Die hauptsächliche Arbeit bestand also in einer Neuimplementierung der `compete.m`. Die für diese Funktion wichtigen Aufrufe und ihre Ein- und Ausgabewerte zeigt Abbildung 4-1.

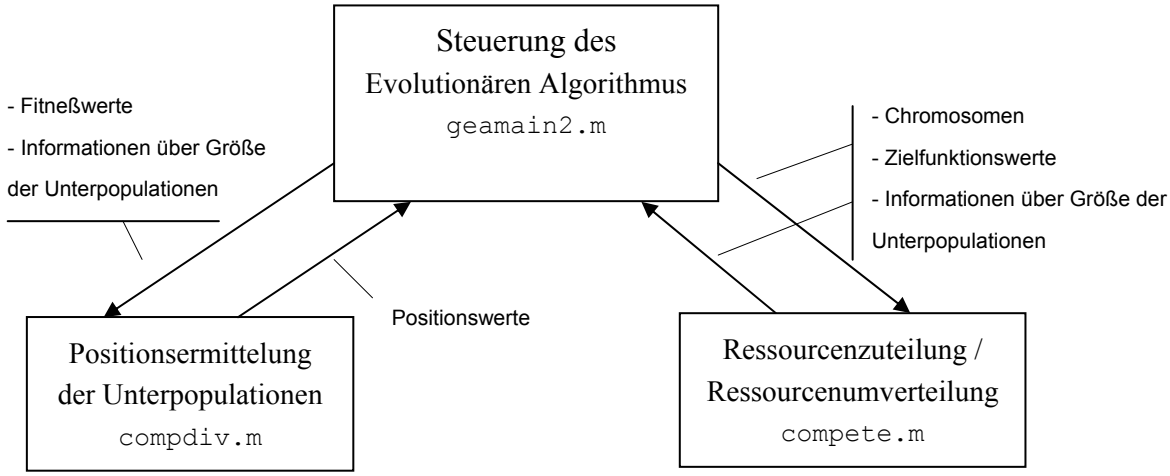

Abb. 4-1 Aufrufe und Parameterübergabe des für die `compete.m` relevanten Teils der [GEATbx]

Die Positionsermittlung der Unterpopulationen erfolgt in der Unterfunktion `compdiv.m` nach dem in Kapitel 2, Abschnitt 2.2.1 vorgestellten Verfahren von Hartmut Pohlheim. Neben Informationen darüber, wieviele und welche Individuen zu den einzelnen Unterpopulationen gehören, benötigt diese Funktion die zugehörigen Fitneßwerte. Als Ergebnis liefert die Funktion also Positionswerte, deren Reihenfolge Aussage über die Ränge der Unterpopulationen macht. Die Auswertung der Reihenfolge und die Ressourcenzu- und -umverteilung anhand dieser Informationen ist Aufgabe der Funktion `compete.m`.

Der Abschnitt 4.1 geht genauer auf die Funktionsweise der `compete.m` ein, während sich Abschnitt 4.2 mit der Vorgehensweise des Extreme Programming und Abschnitt 4.3 mit dessen Anwendung bei der Umsetzung der Konkurrenzfunktion beschäftigt.

4.1 Programmablauf der Positionsermittlung und Ressourcenzu- /-umverteilung

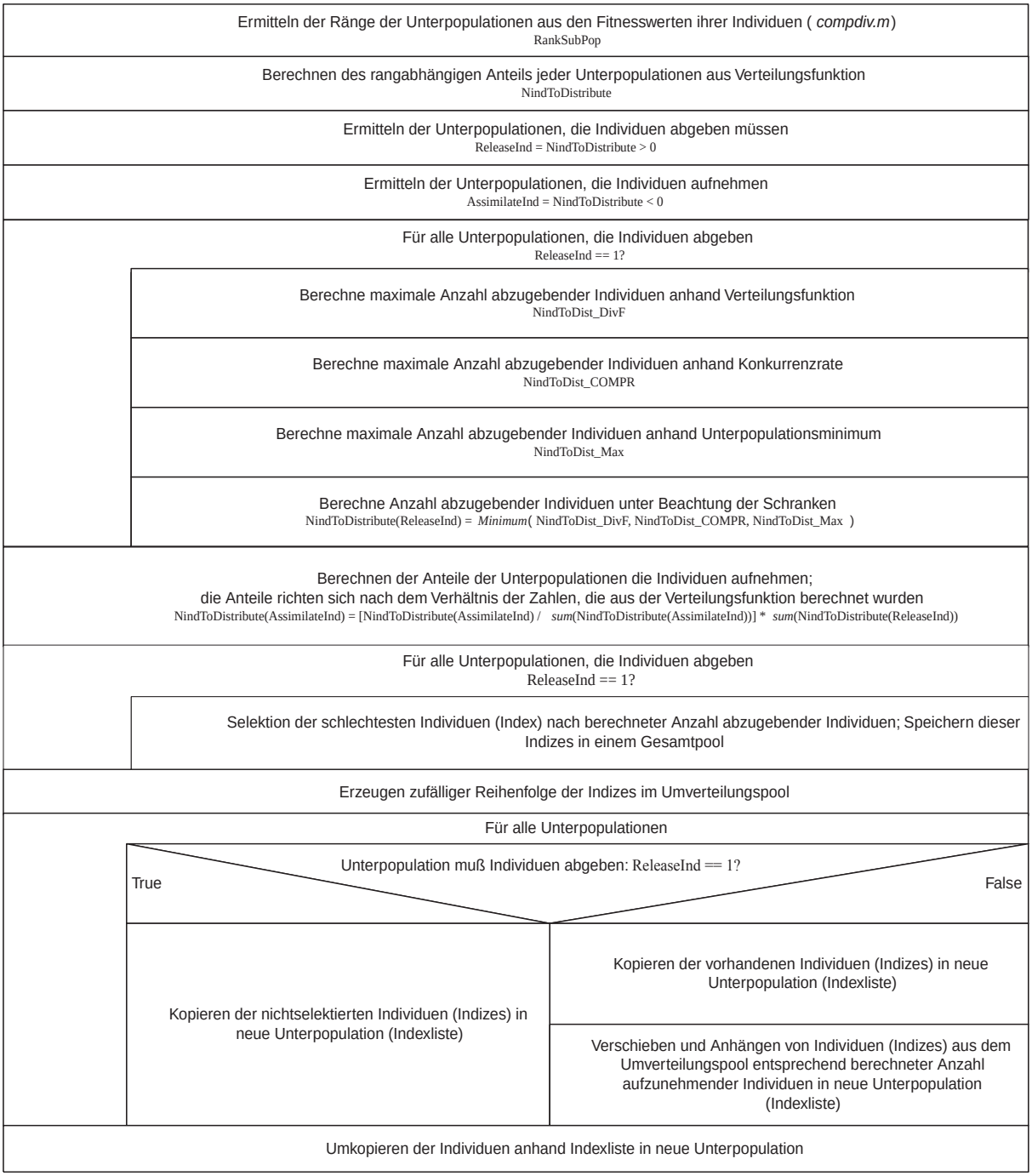

Abb. 4-2 Programmablaufplan `compete.m`; der kommentierte Quelltext kann dem Anhang, Abschnitt 6.1 entnommen werden

4.2 Extreme Programming

Für die Entwicklung und Implementierung der Konkurrenzfunktion wurden einige Elemente des Extreme Programming entlehnt. Hier soll ein kurzer Einblick in das XP (Extreme Programming) folgen, der jedoch bei weitem nicht alle Aspekte und Details beleuchtet.

Extreme Programming ist ein von K. BECK [Bec99] entworfenes Vorgehensmodell für die Softwareentwicklung. Ziel des XP ist es, im Unterschied zu anderen Vorgehensmodellen, daß schnell Resultate vorliegen. Das Produkt soll schon in den ersten Versionen seinen Verwendungszweck, wenigstens zum Teil, erfüllen.

Extreme Programming stützt sich auf eine Reihe von Praktiken, wobei hier nur einige aufgezählt werden sollen:

The Planning Game

Beim Planspiel legen Auftraggeber und Entwickler den Umfang der nächsten Softwareversion fest, also welche Funktionen (User Stories) als nächstes implementiert werden. Es werden Prioritäten vergeben, in welcher Reihenfolge dies geschieht. Freigabetermine werden abgesprochen.

Der Auftraggeber formuliert keine Anforderungen, sondern Probleme, sogenannte „User Stories“, die das Produkt lösen soll. Jede dieser User Stories beschreibt nur einen Gesichtspunkt der Anwendung. Die Sammlung aller User Stories ergibt den kompletten Ablauf. Wenn der Auftraggeber später noch weitere Einfälle hat, kommt einfach eine weitere Story dazu.

Small Releases

Jede freigegebene Softwareversion soll so klein wie möglich gehalten werden, so daß sie einen Nutzwert für den Auftraggeber besitzt.

Testing

Tests sind eine wesentliche Praktik des XP. Vor dem Implementieren müssen Entwickler und Auftraggeber die Testfälle festlegen. Vor dem Hinzufügen einer neuen Funktion steht immer die Erstellung eines Testmoduls. Weiterhin müssen nach ihrer Implementierung alle bereits vorhanden Tests durchgeführt werden, um die Funktionsfähigkeit der Software zu gewährleisten. Ändern sich Funktionen, müssen auch die Tests geändert werden. So entwickeln sich die Tests parallel zum Produkt über die gesamte Projektzeit weiter. Die Tests werden automatisiert abgearbeitet.

Simple Design

Einfaches Design heißt, daß die Software alle vorgesehenen Testfälle bestehen muß. Sie darf keine doppelten Abläufe besitzen und jeder beabsichtigte Zweck muß wichtig für die Software sein. Die Software darf nur benötigte Fähigkeiten besitzen.

Pair Programming

Programmiert wird paarweise. D. h., zwei Programmierer bilden eine Partnerschaft und teilen sich einen Rechner, einen Monitor und eine Tastatur. Die Partnerschaften sind dynamisch, anders gesagt, die Programmierer können nach Bedarf und Verantwortlichkeit die Partnerschaften wechseln. Auf diese Weise arbeiten sie an verschiedenen Komponenten der Software mit.

Collective Ownership

Der Quelltext gehört dem ganzen Team und jeder hat das Recht, an allen Teilen des Quelltextes Veränderungen vornehmen zu dürfen. Jeder ist verantwortlich für das gesamte System. Die Voraussetzung dafür wurde durch das Pair Programming geschaffen, da jeder etwas über jeden Teil des Systems weiß.

Continuous Integration

Der Quellcode wird alle paar Stunden integriert und getestet. Die Tests müssen hundertprozentig erfolgreich abgelaufen sein, bevor neuer Code integriert werden darf. Damit steht fest, welches Team im Falle eines Fehlers für diesen verantwortlich ist (nämlich das Team, das gerade integriert).

4.3 Anwendung des Extreme Programming bei der Implementierung

Bei der Konkurrenzfunktion handelt es sich nur um ein kleines Projekt, das zugleich auch nur einer geringen Koordination bzw. Abstimmung zwischen Auftraggeber und Entwickler bedarf. Aus diesem Grunde war nur ein Programmierer (der Autor dieses Textes) direkt mit der Implementierung beschäftigt. Ein zweiter Programmierer (der Betreuer) stand für die Überprüfung des Programms bzw. für Fragen diesbezüglich zur Verfügung.

Infolgedessen konnten natürlich auch nicht alle Praktiken des XP umgesetzt werden. So wurden hier die Methoden der *Small Releases*, des *Testing*, des *Simple Design* und der *Continuous Integration* benutzt.

Für das *Testing* wurde eine Testroutine entwickelt. Dazu wurden Referenzvariablen generiert, die das gewünschte Ergebnis des Algorithmus bei genau bekannten Eingangsvariablen darstellen. Die Testroutine vergleicht die Ausgangsvariablen des Programms mit den Referenzvariablen auf Gleichheit.

Auf diese Weise wurden mehrere Testszenarien erstellt, um das Programm auf alle Anforderungen hin zu überprüfen. Nach jeder Änderung der Konkurrenzfunktion wird die Testroutine ausgeführt und auf Fehlerberichte kontrolliert.

Grundfunktionalität

Der Test kontrolliert zunächst anhand eines einmalig zufällig generierten Beispiels (im Gegensatz zu den konstruierten Tests der folgenden Szenarien) die Grundfunktionalität. D. h., werden die nicht selektierten Individuen richtig in die neue Population eingefügt. Des weiteren wird geprüft, ob die richtigen Individuen für die Umverteilung ausgewählt wurden, wie auch deren richtige Anzahl und ob auch sie an die richtigen Stellen in den Ausgangsdaten des Programms zu finden sind.

Test auf Einhalten der minimalen Populationsgröße

Das nächste Szenario besteht aus Daten, die einen Test auf Einhaltung einer minimalen Populationsgröße ermöglichen.

Keine Umverteilung unter entsprechenden Bedingungen

Es wird geprüft, ob bei Nichtänderung des Ranges und der den Unterpopulationen zugeordneten Ranking-Werte keine Umverteilung von Ressourcen stattfindet.

Gleiche Anzahl und Gleichheit der Individuen

In allen Testszenarien wird indirekt geprüft, daß die Individuen der Gesamtpopulation vor und nach der Umverteilung nach Anzahl und Eigenschaften dieselben sind.

5 Zusammenfassung

In der vorliegenden Arbeit wurden mehrere Aspekte Evolutionärer Algorithmen in Zusammenhang mit konkurrierenden Unterpopulationen untersucht.

Um die Güte einer Unterpopulation zu ermitteln, wurde der Ansatz verfolgt, die Fitneßwerte aller Individuen in die Bewertung einzubeziehen. Das verhindert eine Überbewertung sehr guter Individuen. Es kommen auch Unterpopulationen mit nur geringfügig schlechteren Individuen zum Zuge, die dafür aber in einer größeren Zahl vorhanden sind. Weiterhin gehen in die Gütebewertung auch die Gütewerte der vorangegangen Generationen ein, was zu schnelle Rangwechsel zwischen den Unterpopulationen verhindert. Die Ränge der Unterpopulationen ergeben sich aus der Ordnung der Gütewerte der Unterpopulationen.

Die Ressourcenaufteilung geschieht anhand einer Verteilungsfunktion, die über einen Parameter Verteilungsdruck gestaucht oder gestreckt werden kann. Darüber läßt sich einstellen, wie stark die besseren Unterpopulationen bevorzugt werden. Der Vorgang der Umverteilung ist ein durch Parameter gesteuerter Vorgang, der die Verteilungsfunktion als Zielverteilung betrachtet. Über diese Parameter kann die Geschwindigkeit der Umverteilung beeinflußt werden und damit die Verzögerung der Aufteilung auf die Rangordnung der Unterpopulationen. Nur wenn sich die Rangfolge lange Zeit nicht ändert, stellt sich die Zielverteilung ein. Diese Vorgehensweise erlaubt eine gerechtere Verteilung der Ressourcen zwischen den Unterpopulationen, als es bisherige Ansätze taten.

Welche Vorteile bringt der Einsatz konkurrierender Unterpopulationen? Konkurrenz wird immer im Zusammenspiel mit Migration verwendet, durch deren Nutzung Kooperation zwischen Strategien entstehen kann. Kooperation ist hier im Sinne einer Ablösestrategie gemeint, in der eine Kombination verschiedener Suchstrategien (die jeweils einer Unterpopulation zugeordnet sind) die Optimierungsaufgabe besser lösen kann, als alle Suchstrategien für sich jeweils allein. Dadurch kommen zu verschiedenen Zeitpunkten der Optimierung unterschiedliche Strategien zum Einsatz, die von der jeweils vorangegangenen besten Suchstrategie in eine gute Startposition versetzt werden. Wenn nun Konkurrenz zwischen den Unterpopulationen geschaffen wird, können nochmals bessere Ergebnisse erzielt werden.

Ein anderer Vorteil konkurrierender Unterpopulationen ist die Untersuchung von verschiedenen Suchstrategien im Hinblick auf die Auswahl der für ein bestimmten Problemkreis am besten geeigneten Strategie. Es können verschiedene Strategien (Unterpopulationen mit unterschiedlichen Parametersätzen) ins „Rennen“ geschickt werden, wobei durch die Nutzung von Konkurrenz den schlechter geeigneten Strategien Ressourcen entzogen werden. Diese laufen nur mit, und verbrauchen kaum Rechenzeit. Daher fällt die Auswahl der am besten geeigneten Strategien (die mit den vorgestellten Mittel gut verglichen werden können) leichter, als dies bei separat ablaufenden Optimierungen der Fall wäre.

Der Einsatz von konkurrierenden Strategien ist eine leistungsfähige Erweiterung Evolutionärer Algorithmen. Das hat diese Arbeit anhand von (wenn auch konstruierten) Beispielen gezeigt. Bei

den immer komplexer werdenden realen Optimierungsproblemen werden sie zukünftig ihr Potential zeigen können.

6 Anhang

6.1 Quelltexte

6.1.1 Quelltextauszug `compete.m`

```
% COMPETition between subpopulations
%
% This function performs competition between subpopulations.
% The selection of the individuals for deletion can be chosen
% between:
%  - uniform at random selection
%  - fitness-based selection
% For fitness-based competition (worst individuals are deleted)
% the fitness values/ranking of the population (Rank) is needed.
% If omitted or empty single objective scaling using the first
% column of the objective values (ObjV) is assumed.
% If the objective values of the population (ObjV) are input
% parameter and ObjV is output parameter the objective values
% are copied, according to the flowing of individuals, saving
% the recomputation of the objective values for the whole popu-
% lation.
% The function can handle multiple objective values per individual.
%
% Syntax:  [NewChrom, NewObjV, SUBPOP, CompQuality] = compete(Chrom, SUBPOP, CompQuality,
%                                                              CompOpt, ObjV, FitnV)
%
% Input parameters:
%    Chrom     - Matrix containing the individuals of the current
%                population. Each row corresponds to one individual.
%    SubPop    - Vector/scalar containing number of individuals per
%                subpopulation/number of subpopulations
%                if omitted or NaN, 1 subpopulation is assumed
%    CompQuality -
%                if omitted, NaN or empty, CompQuality is calculated
%                equivalent to the size of each subpopulation
%    CompOpt   - (optional) Vector containing competition paremeters
%                CompOpt(1): CompRate - Rate of individuals to be deleted/
%                            included per subpopulation (% of subpopulation)
%                            if omitted or NaN, 0.2 (20%) is assumed
%                CompOpt(2): Select - number indicating the selection method
%                            of deleting individuals
%                            0 - uniform selection
%                            1 - fitness-based selection (worst individuals are deleted)
%                            if omitted or NaN, 1 is assumed
%                CompOpt(3): CompMin - number indicating the minimal number of
%                            individuals in a subpopulation
%                            if omitted or NaN, 5 is assumed
%                CompOpt(4): DivisionPressure (analog to selective pressure)
%                            maximal 2 für lineare Verteilung
%                            maximal Anzahl der Unterpopulationen minus 2
%                            if omitted or NaN, 2 is assumed
%                CompOpt(5): for test purpose only
%                            1 - lin. division function
%                            2 - nonlin. division function
%    ObjV      - Column vector or matrix containing the objective values
%                of the individuals in the current population,
%                if Rank is omitted, first column is used for
%                fitness-based competition, saves recalculation
%                of objective values for population
```

```
%    FitnV     - (optional) Column vector containing the fitness
%                values of the individuals
%                in the current population, best individual has
%                highest value,
%                if omitted or empty, single objective scaling
%                using first column of ObjV is assumed
%
% Output parameters:
%    NewChrom  - Matrix containing the individuals of the current
%                population after competition.
%    NewObjV   - (optional) Column vector containing the objective
%                values of the individuals of the current generation
%                after competition.
%    SubPop    - (optional) Vector/scalar containing number of indi-
%                viduals per subpopulation/number of subpopulations
%                after competition
%
% See also: geamain2, migrate, compdiv

function [NewChrom, NewObjV, SubPop, CompQuality, TestVars] = ...
         compete(Chrom, SubPop, CompQuality, CompOpt, ObjV, FitnV);

% change this parameter for other selection methods: worst, best, random
selectmeth = 'worst';

% change these parameters for turning on some visualisation
DrawDivision = 1;
DrawDistribution = 1;
truncationdivision = 0;

NumSubPop = length(SubPop);

% Set standard competition parameter
% CompRate = 0.2; Select = 1; CompMin = 5; DivisionPressure = 2
CompOptStandard = [0.2, 1, 5, 2, NaN];
CompOptIntern = CompOptStandard; CompOptIntern(1:length(CompOpt)) = CompOpt;
CompRate = CompOptIntern(1); Select = CompOptIntern(2); CompMin = CompOptIntern(3);
DivPress = CompOptIntern(4);

if isnan(CompRate), CompRate = CompOptStandard(1);
elseif (CompRate < 0 | CompRate > 1),
   error('Parameter for competition rate must be a scalar in [0 1]');
end

if isnan(Select), Select = CompOptStandard(2);
elseif (~any(Select==[0,1])),
   error('Parameter for selection method must be 0 or 1');
end

if isnan(CompMin), CompMin = CompOptStandard(3);
elseif CompMin < 0,
   error('Parameter for competition minimum must be an integer greater 0!');
end

if (Select == 1 & IsFitnV == 0),
   error('FitnV (or ObjV) for fitness-based distribution needed (in competition).');
end

if CompRate == 0,  NewChrom = Chrom; NewObjV = ObjV; return; end

% ranking of subpopulations, subpopulation(n) corresponds to RankSubPop(n)
RankSubPop = ...
   sum(
       repmat(CompQuality, [1, NumSubPop])' > repmat(CompQuality, [1, NumSubPop])  ...
      ) + 1 ;

% ressource ratio
if ~isnan(CompOptIntern(5)),
   % for test purpose
   ratio = ranking(RankSubPop, [DivPress, CompOpt(5),0],[],[]) / length(RankSubPop);
else
   ratio = ranking(RankSubPop', DivPress) / length(RankSubPop);
end;

if truncationdivision == 1,
   ratio(RankSubPop < 2) = 1;
   ratio(RankSubPop > 1) = 0;
```

```
end
% total number of individuals
NindAll = size(ObjV,1);
NewSubPop = [];

% find out which subpops have to release individuals according division function
% number of individuals to rearrange per subpop
NindToDistribute = fix(SubPop - NindAll * ratio);
% populations which have to release individuals
ReleaseInd = find(NindToDistribute > 0);

% find out how many individuals have to be released
% by keeping of: CompMin, max. number of ind. to release defined by CompRate ...
%                and max. number defined by division function
% at least one individual will be released by keeping CompMin
% number of individuals which can be distributed according comprate
NindToDist_CompRate = floor(CompRate * (SubPop(ReleaseInd)));

% number of individuals which should be distributed according division function
NindToDist_DivF = floor(SubPop(ReleaseInd) - ratio(ReleaseInd) * NindAll);

% number of individuals which can be distributed by keeping ...
%  minimal number of individuals per subpop
NindToDist_Max = SubPop(ReleaseInd) - CompMin;

% keeping of maximal number of individuals to release defined by comprate ...
%  and max. number defined by division function
% at least one individual will be released
NindToDistribute(ReleaseInd) = max(min(NindToDist_CompRate, NindToDist_DivF), 1);

% keeping of CompMin
NindToDistribute(ReleaseInd) = min(NindToDist_Max, NindToDistribute(ReleaseInd));

% find out how many individuals have to be added
% populations which assimilate individuals
AssimilateInd = find(NindToDistribute < 0);

% get individuals proportional to the number of individuals to be released, ...
%  according division function
NindToDistribute(AssimilateInd) = - ceil(NindToDistribute(AssimilateInd) ...
                                  / sum(NindToDistribute(AssimilateInd)) ...
                                  * sum(NindToDistribute(ReleaseInd)));

                               % calculate size of new subpops
NewSubPop = SubPop - NindToDistribute;

% remain of rounding difference
DiffNindAll = NindAll - sum(NewSubPop);

% detect best subpop
[dummy, ixbestsubpop] = min(RankSubPop);

% transfer remain to best subpop
NindToDistribute(ixbestsubpop) = NindToDistribute(ixbestsubpop) - DiffNindAll;

% final values
NewSubPop = SubPop - NindToDistribute;

% sort individuals within their subpops and select individuals for distribution
IxToDistribute = [];
IxToDistributeAll = [];
for irun = 1 : NumSubPop,
   % indices of individuals of current subpop
   StartIx = sum(SubPop(1 : irun - 1)) + 1; NindIx = StartIx : sum(SubPop(1 : irun));

   % sort individuals within their subpop, relative indices
   switch selectmeth
   case 'worst', [dummy, IxSortedRel] = sort(-FitnV(NindIx));
   case 'best', [dummy, IxSortedRel] = sort(FitnV(NindIx));
   case 'random', [dummy, IxSortedRel] = sort(rand(SubPop(irun),1));
   end

   % compute absolute indices
   IxSortedAbs(NindIx) = IxSortedRel + StartIx - 1;

   % indices of individuals to distribute per subpop
   IxToDistribute{irun} = [IxSortedAbs(StartIx : StartIx + NindToDistribute(irun) - 1)];
```

```
   if NindToDistribute(irun) > 0,
      % indices of individuals to distribute, overall
      IxToDistributeAll = [IxToDistributeAll; IxToDistribute{irun}'];
   end
end

% reconstruction of the whole population
NewObjV = zeros(size(ObjV));
NewChrom = zeros(size(Chrom));
BackupIxToDistribute=IxToDistribute;
TestVars=[IxToDistribute {NindToDistribute} {IxToDistributeAll}]; % for test purpose
IxNewSUBPOP=[];

% how much individuals have to be added per subpop
AssimilateInd = find(NindToDistribute < 0);
NindToAdd = zeros(size(NindToDistribute));
NindToAdd(AssimilateInd) = -NindToDistribute(AssimilateInd);

% for test purpose
% save testvars\nodistribution\IxToDistributeAll.txt IxToDistributeAll -ascii;

% create random order of distributionpool
[dummy, IxToAdd]=sort(rand(length(IxToDistributeAll),1));

for irun = 1:NumSubPop,
   % indices of individuals of current subpop, old and new
   StartIx = sum(SubPop(1 : irun - 1)) + 1;
   NindIx = StartIx : sum(SubPop(1 : irun));
   NewStartIx = sum(NewSubPop(1 : irun - 1)) + 1;
   NewNindIx = NewStartIx : sum(NewSubPop(1 : irun));

   if NindToDistribute(irun) > 0, % if individuals have to be released
      BackupNindIx = NindIx;

      % remove individuals to delete from sub index
      BackupNindIx(IxToDistribute{irun} - StartIx + 1) = [];

      % new order of individuals
      IxNewSUBPOP = [IxNewSUBPOP; BackupNindIx'];
   else % if individuals will be assimilated
      BackupNewNindIx = NewNindIx;

      % shorten length of BackupNewNindIx to length of NindIx
      BackupNewNindIx(SubPop(irun) + 1 : NewSubPop(irun)) = [];

      % new order of individuals
      IxNewSUBPOP = [IxNewSUBPOP; NindIx'];
      IxNewSUBPOP = ...
        [IxNewSUBPOP; ...
         IxToDistributeAll( ...
            IxToAdd(sum(NindToAdd(1 : irun - 1)) + 1 : sum(NindToAdd( 1 : irun))) ...
         )
        ];
   end
end

% copy old individuals to new population
NewChrom = Chrom(IxNewSUBPOP, :);
NewObjV = ObjV(IxNewSUBPOP, :);

% draw figure of required division and actual division
if DrawDivision ~= 0 | DrawDistribution ~= 0,

   FigureName = ('Division and Distribute per Generation');

   NumRowSubplots = 1;
   NumColSubplots = DrawDistribution + DrawDivision;

   % look for figure, set name, paperposition, position and all standard settings
   FigureTag = 'geatbx_fig_complot';

   VisuOptBase = visuoptset;
   VisuOptBase = visuoptset( VisuOptBase ...
      , 'Plot.FigureName', FigureName, 'Plot.FigureTag', FigureTag ...
      , 'Plot.SubplotGrid', [NumRowSubplots NumColSubplots] ...
      , 'Preproc.CompressMode', [] ...
      , 'Plot.WindowWidth', 0.4 ...
```

```
      );

   if DrawDivision == 1,
      % draw subplot of required division
      VisuOpt = visuoptset( VisuOptBase ...
         , 'Plot.Type', '2dstairs', 'Plot.GridMode', 'y' ...
         , 'Plot.LegendMode', 1, 'Plot.SubplotNum', NumRowSubplots ...
         , 'Plot.Title', 'Division', ...
         , 'Plot.YCaption', 'rank subpop', 'Plot.XCaption', 'share of ressources' ...
         );
      currentratio = NewSubPop / sum(NewSubPop);
      currentratio = -sort(-currentratio);
      % draw subplot of current division
      visubase({ [-sort(-ratio); min(ratio)], [0:NumSubPop] + 0.5; ...
                 [currentratio; min(currentratio)], [0:NumSubPop] + 0.5}, ...
               {'required division'; 'current division'}, VisuOpt);
   end

   % draw figure of distribution
   if DrawDistribution == 1,
      VisuOpt = visuoptset( VisuOptBase ...
         , 'Plot.Type', 'cq', 'Plot.ColorBarMode', '', 'Plot.GridMode', '' ...
         , 'Plot.LegendMode', 0, 'Plot.SubplotNum', NumColSubplots ...
         , 'Plot.Title', 'Distribution' ...
         , 'Plot.YCaption', 'index of individuals', ...
         , 'Plot.XCaption', 'old subp. | distr. subpop | new subp.' ...
         );
      ColVal = NaN * zeros(NindAll, 3); % last column is not used by pcolor
      for irun = 1:length(SubPop),
         % indices of individuals of current subpop
         StartIx = sum(SubPop(1 : irun - 1)) + 1; NindIx = StartIx : sum(SubPop(1 : irun));
         NewStartIx = sum(NewSubPop(1 : irun - 1)) + 1;
         NewNindIx = NewStartIx : sum(NewSubPop(1 : irun));
         % set color values of 1st and 3rd column to number of current subpop
         % 1st column stands for the original subpop, 3rd column stands for the new subpop
         ColVal(NindIx, 1) = irun;
         ColVal(NewNindIx, 3) = irun;
      end
      % 2nd column is set to newsubpop but with colors of the 1st
      ColVal(:, 2) = ColVal(IxNewSUBPOP, 1);
      ColVal1 = [num2cell(ColVal, 2), num2cell(NaN * ones(size(ColVal)), 2)];
      visubase(ColVal1, {''}, VisuOpt);
   end
end
SubPop = NewSubPop';

% end of function
```

6.1.2 Quelltextauszug `compdiv.m`

```
% COMPute DIVerse things of GEA Toolbox
%
% This function computes diverse special results for the GEA Toolbox
% during computation used at different points of the toolbox.
%
% Syntax:  [OP1, OP2, OP3, OP4, OP5] = ...
%          compdiv(WhatTask, P1, P2, P3, P4, P5, P6, P7, P8, P9, P10)
%
% Input parameters:
%    WhatTask      - String containing the name of the needed computation
%    P1 - P10  - Parameters needed for the specific computations
%
% Output parameter:
%    OP1 - OP5 - Output parameters, specific for every computation
%
%    WhatTask == 'possubpop'
%       Sort/order subpopulations according objective values, return
%          position of every subpopulation
%       P1     - Matrix/vector containing (objective) values
%       P2     - Vector containing number of individuals per subpopulation
%       P3     - Vector containing previous filtered order of subpopulations
```

```
%       OP1    - (row) Vector containing position of every subpopulation
%                  subpopulation with best/minimal objective values gets 1,
%                  worst subpopulation gets length(SUBPOP)
%       OP2    - (row) Vector containing index to best values in every subpopulation,
%                  thus, P1(OP2) = values of best individuals of
%                  every subpopulation
%       OP3    - (row) Vector containing filtered order of subpopulations

function [OP1, OP2, OP3, OP4, OP5] = ...
         compdiv(WhatTask, P1, P2, P3, P4, P5, P6, P7, P8, P9, P10);

% Compute ordering of subpopulations
elseif strcmp(lower(WhatTask), 'possubpop'),
   ObjV = P1; SUBPOP = P2; NumSUBPOP = length(SUBPOP);
   if nargin < 4, PosSubPopFilt = []; else PosSubPopFilt = P3; end

   % Parameter checking
   if size(ObjV,1) ~= sum(SUBPOP),
      error('Size of ObjV (P1) and values of SUBPOP (P2) disagree!');
   end

   % Set internal variables
   ValforMinimum = zeros(1, NumSUBPOP); ixbest = zeros(1, NumSUBPOP);

   % Compute best (minimal) value / weighted sum of ranks per subpopulation/subvector
   % 0: weighted ranking using all values; 1: ranking with best value only
   RankwithOne = 0;
   if RankwithOne == 1,
      % sort values of one subpopulation, look for minimal value in subpopulation,
      % smaller value for minimum in subpopulation is better and determines rank
      % of subpopulation,
      % here only the best/minimal value is used for subpopulation ranking
      for irun = 1:NumSUBPOP,
         % Extract values
         Startix = sum(SUBPOP(1:irun-1))+1;
         SubObjV = ObjV(Startix:sum(SUBPOP(1:irun)),:);
         if size(ObjV, 2) > 1,
            % at the moment use first column for single objective ranking
            SubObjV = SubObjV(:, 1);
         end
         [ValforMinimum(irun), ixbest(irun)] = min(SubObjV);
         ixbest(irun) = ixbest(irun) + Startix - 1;
      end
   else
      % sort by ranking all individuals at once, compute sum of (negative) rank
      % values in subpopulation and weight with number of individuals in
      % subpopulation, smaller value for weighted sum is better and determines
      % rank of subpopulation,
      % here all values are used for weighted ranking of subpopulations
      RankValues = -(ranking(ObjV));
      RankValSubpop = zeros(1, NumSUBPOP);
      for irun = 1: NumSUBPOP,
         StartIx = sum(SUBPOP(1:irun-1))+1; NindIx = StartIx:sum(SUBPOP(1:irun));
         [MinRank(irun), ixbest(irun)] = min(RankValues(NindIx));
         ixbest(irun) = ixbest(irun) + StartIx - 1;
         ValforMinimum(irun) = sum(RankValues(NindIx)) / length(NindIx);
      end
   end

   if RankwithOne == -1,
      [minRank, ixbestall] = min(MinRank);
      ValforMinimum = zeros(1, NumSUBPOP) + minRank + 10;
      ValforMinimum(ixbestall) = minRank;
   end

   % Compute ranking of subpopulations, index of best individual per subpopulation and
   % filtered order of subpopulations
      sum(
       repmat(CompQuality, [1, NumSubPop])' > repmat(CompQuality, [1, NumSubPop])  ...
      ) + 1 ;

   PosSubPopAct = ...
    sum( ...
         repmat(ValforMinimum, [NumSUBPOP, 1])' < repmat(ValforMinimum, [NumSUBPOP, 1]) ...
       ) + 1);
   if RankwithOne == -1, PosSubPopFilt = PosSubPopAct;
   elseif isempty(PosSubPopFilt),
```

```
      PosSubPopFilt = length(PosSubPopAct) / 2 + 0.1 * PosSubPopAct;
   else PosSubPopFilt = 0.9 * PosSubPopFilt + 0.1 * PosSubPopAct; end

   % Assign variables to output variables
   OP1 = PosSubPopAct;
   OP2 = ixbest;
   OP3 = PosSubPopFilt;
```

6.1.3 Quelltext `testcompete.m`

```
% Syntax:  testcompete(number);
%            number - 0: all tests
%                     1: simple test with 4 subpops, 40 individuals
%                     2: check for compliance with defined compmin
%                     3: check for nothing happens if sizes of subpops does not change

function testcompete(number);

switch number
case 0, directories = {'testvars\4subpop\';'testvars\compmin\';'testvars\nodistribution\'};
case 1, directories = {'testvars\4subpop\'};
case 2, directories = {'testvars\compmin\'};
case 3, directories = {'testvars\nodistribution\'};
end;

for i=1:length(directories)

   directoryname=directories{i};

   Chrom = load([directoryname,'chrom.txt']);
   SUBPOP = load([directoryname,'subpop.txt']);
   CompQuality = load([directoryname,'CompQuality.txt']);
   CompOpt = load([directoryname,'CompOpt.txt']);
   ObjV = load([directoryname,'objv.txt']);

   NumSUBPOP = length(SUBPOP);

   [NewChrom, NewObjV, NewSUBPOP, NewCompQuality, TestVars] = ...
      compete(Chrom, SUBPOP, CompQuality, CompOpt, ObjV);

   NewChromRef = load([directoryname, 'newchrom.txt']);
   NewSUBPOPRef = load([directoryname, 'newsubpop.txt']);
   NewObjVRef = load([directoryname, 'newobjv.txt']);
   IxToDistributeAllRef = load([directoryname, 'IxToDistributeAll.txt']);


   [IxToDistribute] = TestVars(1 : NumSUBPOP);
   NindToDistribute = TestVars{NumSUBPOP + 1};
   IxToDistributeAll = TestVars{NumSUBPOP + 2};

   c = find(NindToDistribute < 0);
   NindToAdd = zeros(size(NindToDistribute));
   NindToAdd(c) = NindToDistribute(c);

   % check for right newobjv and newchrom
   ObjVAdded = [];
   ChromAdded = [];
   for irun = 1 : NumSUBPOP,
      StartIx = sum(NewSUBPOPRef(1:irun - 1)) + 1;
      NindIx = StartIx : sum(NewSUBPOPRef(1 : irun)) + NindToAdd(irun);
      msg = ('NewObjV and NewChrom are right');
      if any(NewObjVRef(NindIx, :)~=NewObjV(NindIx,:)), msg =('NewObjV is false'); end;
      if any(NewChromRef(NindIx, :)~=NewChrom(NindIx,:)), msg = ('NewChrom is false'); end;
      if NindToDistribute(irun) < 0,
         % extraction of added individuals and their object values from result of compete
         ObjVAdded = ...
            [ObjVAdded;
            NewObjV(StartIx + length(NindIx) : sum(NewSUBPOP(1 : irun)), :)];
         ChromAdded = ...
            [ChromAdded; ...
            NewChrom(StartIx + length(NindIx) : sum(NewSUBPOP(1 : irun)), :)];
```

```
      end;
   end;
   if (length(IxToDistributeAllRef) ~= size(ObjVAdded,1) ...
     | length(IxToDistributeAllRef) ~= size(ChromAdded, 1)),
         msg = ('number of rearranged individuals does not conform with reference');
   elseif isempty(IxToDistributeAllRef) & isempty(ObjVAdded) & isempty(ChromAdded),
      msg = ('there was no redistribution');
   else
      % comparision of rearranged objectvalues
      if any(sortrows(ObjV(IxToDistributeAllRef, :)) ~= sortrows(ObjVAdded)),
         msg = ('NewObjV is false');
      end         % comparision of rearranged individuals
      if any(sortrows(Chrom(IxToDistributeAllRef, :)) ~= sortrows(ChromAdded)),
         msg = ('NewChrom is false');
      end
   end;
   % ranking of subpopulations
   [dummy, RankSubPop] = sort(CompQuality);
   [dummy, RankSubPop] = sort(RankSubPop);
end;

% end of function
```

6.2 Literaturverzeichnis

[Bec99] *Beck, K.*: extreme Programming explained. Reading, MA: Addison-Wesley, 1999.

[CPR96] *Corno, F., Prinetto, P., Rebaudengo, M. and Reorda, M. S.*: Exploiting Competing Subpopulations for Automatic Generation of Test Sequences for Digital Circuits. in [PPSN4], pp. 792-800, 1996.

[Ble01] *Bleul, A.*: Programmier-Extremisten, c't 3/01, pp. 182-185: Hannover, Verlag Heinz Heise, 2001.

[DeJ75] *De Jong, K.*: An analysis of the behavior of a class of genetic adaptive systems. Doctoral dissertation, University of Michigan, Dissertation Abstracts International, 36(10), 5140B, University Microfilms No. 76-9381, 1975.

[GEATbx] *Pohlheim, H.*: GEATbx: Genetic and Evolutionary Algorithm Toolbox for use with MATLAB®. http://www.geatbx.com/index.html, 1995-2001.

[ICEC96] Proceedings of the Third IEEE Conference on Evolutionary Computation 1996, Piscataway, New Jersey, USA: IEEE Press, 1996.

[IK2001] *Igel, C., Kreutz, M.*: Operator adaptation in evolutionary computation and its application to structure optimization of neural networks. Technical Report IRINI 2001-03, Institut für Neuroinformatik, Ruhr- Universität Bochum, 2001. http://www.neuroinformatik.ruhr-uni-bochum.de/PEOPLE/igel/EP003.ps.gz

[Poh98] *Pohlheim, H.*: Entwicklung und systemtechnische Anwendung Evolutionärer Algorithmen. Aachen, Germany: Shaker Verlag, 1998. (Development and Engineering Application of Evolutionary Algorithms) http://www.pohlheim.com/diss/index.html

[Poh99] *Pohlheim, H.*: Evolutionäre Algorithmen - Verfahren, Operatoren, Hinweise aus der Praxis. Berlin, Heidelberg, New York: Springer-Verlag, 1999. (Evolutionary Algorithms - Methods, Operators, Practical Examples. in german) http://www.pohlheim.com/eavoh/index.html

[Poh00] *Pohlheim, H.*: Konkurrenz und Kooperation in Erweiterten Evolutionären Algorithmen. Proceedings of 45th International Scientific Colloquium Ilmenau, Germany, pp. 179-187, 2000. http://www.pohlheim.com/publications.html

[Poh01] *Pohlheim, H.*: Competition and Cooperation in Extended Evolutionary Algorithms. in Spector, L. (ed.): GECCO'2001 - Proceedings of the Genetic and Evolutionary Computation Conference – Late Breaking Papers. San Francisco, CA: Morgan Kaufmann Publishers, pp. 331-338, 2001. http://www.pohlheim.com/publications.html

[PPSN3] *Davidor, Y., Schwefel, H.-P. and Männer, R.*: Parallel Problem Solving from Nature - PPSN III: International Conference on Evolutionary Computation. volume 866 of Lecture Notes in Computer Science, Berlin, Heidelberg, New York: Springer-Verlag, 1994.

[PPSN4] *Voigt, H.-M., Ebeling, W., Rechenberg, I. and Schwefel, H.-P.*: Parallel Problem Solving from Nature - PPSN IV: International Conference on Evolutionary Computation. volume 1141 of Lecture Notes in Computer Science, Berlin, Heidelberg, New York: Springer-Verlag, 1996.

[SVM94] *Schlierkamp-Voosen, D. and Mühlenbein, H.*: Strategy adaptation by competing subpopulations. in , pp. 199-208, 1994. ftp://borneo.gmd.de/pub/as/ga/gmd_as_ga-94_14.ps

[SVM96] *Schlierkamp-Voosen, D. and Mühlenbein, H.*: Adaptation of Population Sizes by Competing Subpopulations. in [ICEC96], pp. 330-335, 1996. ftp://borneo.gmd.de/pub/as/ga/gmd_as_ga-96_01.ps

[Swm96] *Schwehm, M.*: Globale Optimierung mit massiv parallelen genetischen Algorithmen. Dissertation, Universität Erlangen-Nürnberg, 1996. http://www7.informatik.uni-erlangen.de/~schwehm/MYpapers/Dissertation.html

6.3 Abbildungsverzeichnis

Abb. 1-1 Ablauf eines Evolutionären Algorithmus 2
Abb. 2-1 Die drei Schritte der Konkurrenzfunktion 7
Abb. 2-2 Verteilungsfunktionen bei verschiedenen Verteilungsdrücken für 8 Unterpopulationen; die Verteilungsfunktion bestimmt den Anteil der Unterpopulationen an den Ressourcen abhängig von ihrem Rang. (Deutlich erkennbar ist der geringe Unterschied zwischen linearer und nichtlinearer Verteilung bei *DP* = 2) 9
Abb. 2-3 Beispiel der Funktionsweise des Hinzufügens und Abgebens von Individuen: Die Differenz zwischen aktueller Verteilung und Zielverteilung ist Maß dafür, ob Ressourcen abgegeben oder aufgenommen werden 11
Abb. 2-4 Beispiel der Funktionsweise des Hinzufügens und Abgebens von Individuen: Stellt obere Begrenzungen für die Anzahl abzugebender Individuen dar; das Minimum dieser wird verwendet 11
Abb. 2-5 Beispiel der Funktionsweise des Hinzufügens und Abgebens von Individuen; 42 Individuen sollen insgesamt aufgenommen werden, aber 7 stehen nur zur Verfügung -> die 7 Individuen werden verhältnismäßig verteilt 12
Abb. 3-1 *Konkurrenzintervall*: 5 Generationen, *Konkurrenzrate*: 10% 15
Abb. 3-2 *Konkurrenzintervall*: 5 Generationen, *Konkurrenzrate*: 30% 16
Abb. 3-3 *Konkurrenzintervall*: 5 Generationen, *Konkurrenzrate*: 50% 16
Abb. 3-4 *Konkurrenzintervall*: 5 Generationen, *Konkurrenzrate*: 100% 17
Abb. 3-5 *Konkurrenzintervall*: 10 Generationen, Konkurrenzraten von links nach rechts: 10%, 50%, 100% 17
Abb. 3-6 Vergleich: verschiedene Parameter aber gleiche maximale Umverteilungsgeschwindigkeiten links: *Konkurrenzintervall* 5, *Konkurrenzrate* 10%; rechts: *Konkurrenzintervall* 10, *Konkurrenzrate* 20% 18
Abb. 3-7 Vergleich verschiedener Verteilungsdrücke: links: *DP* = 2; Mitte: *DP* = 4; rechts: *DP* = 6 18
Abb. 3-8 Optimierung der DE JONG Funktion 1, die drei Strategien im Einzellauf, Mutationsbereiche von links nach rechts: 0.1, 0.002, 10^{-5} 20
Abb. 3-9 Optimierung der DE JONG Funktion 1 – nur Migration: links: Konvergenz; rechts: Ordnung der Unterpopulationen 21
Abb. 3-10 Optimierung der DE JONG Funktion 1 – Konkurrenz und Migration: links: Konvergenz; rechts: Ordnung der Unterpopulationen 21
Abb. 3-11 Optimierung der RASTRIGIN-Funktion, die drei Strategien im Einzellauf, Mutationsbereiche von links nach rechts: 0.1, 0.002, 10^{-5} 22
Abb. 3-12 Optimierung RASTRIGIN-Funktion – nur Migration: links: Konvergenz; rechts: Ordnung der Unterpopulationen 22
Abb. 3-13 Optimierung der RASTRIGIN-Funktion – Konkurrenz und Migration: links: Konvergenz; rechts: Ordnung der Unterpopulationen 22
Abb. 4-1 Aufrufe und Parameterübergabe des für die `compete.m` relevanten Teils der [GEATbx] 25
Abb. 4-2 Programmablaufplan `compete.m`, der kommentierte Quelltext kann dem Anhang, Abschnitt 6.1 entnommen werden 26

6.4 Tabellenverzeichnis

Tab. 3-1 Statistischer Vergleich: Migration – Konkurrenz.....23