Prüfer:	Prof. Dr. rer. nat. habil. Paul Levi
Betreuer:	Priv.-Doz. Dr. Andreas Zell

begonnen am:	15. Mai 1995
beendet am:	15. November 1995

CR-Klassifikation:	C.1.2, C.2.4, D.1.3, I.2.8

Diplomarbeit Nr. 1300

Verteilte Evolutionsstrategien auf dem Supercomputer Intel Paragon

Jürgen Wakunda

Fakultät Informatik
Institut für Parallele und
Verteilte Höchstleistungsrechner
Universität Stuttgart
Breitwiesenstraße 20–22
D–70565 Stuttgart

Erklärung

Hiermit versichere ich, diese Arbeit
selbständig verfaßt und nur die
angegebenen Quellen benutzt zu haben.

(Jürgen Wakunda)

Kurzfassung

In dieser Diplomarbeit wurde ein Programm zum parallelen Ablauf von Evolutionsstrategien entworfen und auf dem MIMD–Parallelrechner Intel Paragon implementiert. Das Programm kann über ein einfaches textuelles Menü bedient und durch die Skript–Sprache Tcl programmiert werden. Als Anwendungen wurden 24 Standard–Testfunktionen implementiert.

Evolutionsstrategien sind allgemeine, stochastische Optimierungsverfahren für mathematische und ingenieurstechnische Probleme. Sie wurden an der Technischen Universität Berlin entwickelt und versuchen ähnlich den Genetischen Algorithmen, die Prinzipien der natürlichen Evolution in der Biologie auf technische Systeme umzusetzen. Damit werden dann Lösungen für Optimierungsprobleme gesucht, für die bisher noch kein besseres, mathematisches Verfahren entwickelt wurde.

Die Idee, die hinter den Evolutionsstrategien steckt, ist folgende: statt eine analytische Lösung für ein Problem zu suchen, wird eine Menge von konkreten Lösungen solange verbessert, bis eine gute Lösung gefunden wurde. Dabei werden Mechanismen, die die Natur benutzt, um Lebewesen möglichst gut ihrer Umwelt anzupassen, in die Technik übertragen, um die Lösungen an das zu berechnende Problem anzupassen.

Eine konkrete Lösung wird dabei durch ein Individuum repräsentiert. Dieses besteht im wesentlichen aus den Objektvariablen, in welchen die Lösung gespeichert ist. Weiterhin besitzt jedes Individuum eine Bewertung, die Qualität oder auch Fitness genannt, welche seine Güte im Vergleich mit den anderen Individuen angibt. Aus der Menge von Individuen, der Population, werden gute Individuen ausgewählt, leicht abgeändert und erneut bewertet. Schlechte Individuen werden verworfen. Durch Iteration dieses Prozesses kann eine stetige Verbesserung der vorhandenen Individuen, und damit der Lösungen für das Problem, erreicht werden.

In dieser Diplomarbeit wird gezeigt, daß sich durch die Parallelisierung von Evolutionsstrategien Vorteile im Hinblick auf die grundsätzliche Konvergenzfähigkeit, die Konvergenzgeschwindigkeit und die Qualität der erreichbaren Lösungen ergeben.

Inhaltsverzeichnis

Kapitel 1

Aufgaben und Ziele

1.1 Einführung

Evolutionsstrategien sind stochastische Optimierungsverfahren, die sich sehr gut auf Probleme anwenden lassen, welche noch wenig untersucht sind. Weiterhin sind sie auch für komplexe Probleme im Hochvariablenraum geeignet, bei denen andere Verfahren versagen. Die einzige Eigenschaft des Problemraumes, auf der sie aufbauen, ist die starke Kausalität (siehe Abschnitt 2.4.2) oder Stetigkeit. Deshalb gehören Evolutionsstrategien zu den allgemeinsten Optimierungsverfahren und eignen sich somit für ein sehr breites Spektrum von Anwendungen.

Die Idee der Evolutionsstrategien ist der Natur abgeschaut. Die natürliche Evolution als Vorbild hat gezeigt, daß sie fähig ist, aus einfachen Objekten (belebt oder unbelebt), sehr komplexe Objekte zu bilden. Diese sind hervorragend an ihren Lebensraum und ihre Umweltbedingungen angepaßt. Die Prinzipien der Evolution wurden analysiert, auf das Wesentliche vereinfacht und modelliert. Als Hauptprinzipien haben sich dabei die Vererbung, die Mutation und die Selektion herausgestellt. Es hat sich gezeigt, daß auch das vereinfachte Modell der Evolutionsstrategien fähig ist, die Objekte (Lösungen eines bestimmten Problems) sehr gut an ihre Umgebung (die Problemstellung selbst) anzupassen.

Da die Natur selbst ein hochgradig paralleles System ist, bietet es sich an, die Evolutionsstrategien zu parallelisieren. Dadurch kann nicht nur ein Zeitgewinn erzielt werden, sondern durch die Parallelität kommen noch weitere Aspekte hinzu, die einer besseren und schnelleren Lösung dienen können.

1.2 Aufgabenstellung

Aufgabe dieser Diplomarbeit war es, ein Programm zu erstellen, welches verteilte Evolutionsstrategien (VEES) auf dem MIMD[1]–Parallelrechner Intel Paragon implementiert. Das Programm sollte sowohl interaktiv bedient, als auch durch Batchdateien gesteuert werden können. Weiterhin sollte untersucht werden, inwieweit sich die Parallelität vorteilhaft auf das Konvergenzverhalten und die Qualität der erreichbaren Lösungen auswirkt. Dies sollte anhand von

[1] Klassifikation nach Flynn, s. Abschnitt 3.1

ebenfalls zu implementierenden Standard–Benchmark–Funktionen festgestellt werden. Einige dieser Funktionen sollten zum Vergleich mit anderen Implementierungen von Evolutionsstrategien und Genetischen Algorithmen herangezogen werden. Insbesondere sollte ein Vergleich stattfinden, einerseits mit einer zeitgleich entstandenen Implementierung von massiv parallelen Evolutionsstrategien und andererseits mit einer Implementierung von verteilten Genetischen Algorithmen, die in einer vorangegangenen Arbeit entstanden ist.

1.3 Einbettung in das Gesamtprojekt

Das Programm VEES ist im Rahmen des Projektes *Evolutionäre Algorithmen* (EA) am Lehrstuhl Bildverstehen am Institut für Parallele und Verteilte Höchstleistungsrechner der Universität Stuttgart entstanden. Ziel dieses Projektes ist die Untersuchung der Auswirkungen von Parallelität auf Genetische Algorithmen und Evolutionsstrategien. Zu diesem Zweck wurden einige Programme auf verschiedenen Parallelarchitekturen implementiert und Messungen durchgeführt.

Dabei sind außer VEES bisher folgende Arbeiten entstanden:

- MPES, ein Programm für *M*assiv *P*arallele *E*volutions–*S*trategien von Steffen Görzig, das auf dem SIMD–Rechner MasPar MP-1 läuft [Görzig 95].
- VEGA, ein Programm für *VE*rteilte *G*enetische *A*lgorithmen von Roland Baumann, das genauso wie VEES auf dem MIMD–Rechner Intel Paragon, sowie auf Workstations lauffähig ist [Baumann 95].
- MPGA, ein Programm für *M*assiv *P*arallele *G*enetische *A*lgorithmen von Alex Hummler, das auf dem SIMD–Rechner MasPar MP-1 läuft [Hummler 95].
- CNGA, ein Programm für *G*enetische *A*lgorithmen auf dem Neurocomputer *CN*APS von Marc Ebner [Ebner 95]. Dies ist ebenfalls ein Rechner der SIMD–Klasse, allerdings besitzt er weniger Prozessor–Elemente als die MasPar.
- Eine graphische Benutzeroberfläche, mit der alle fünf Programme einheitlich zu bedienen sind von Alexander Hasel [Hasel 95].

In Abbildung 1.1 sind alle Implementierungen von Evolutionären Algorithmen auf den verschiedenen Rechnertypen dargestellt. Eine Umsetzung von Evolutionsstrategien auf den Neurocomputer CNAPS ist nicht vorgesehen, da dieser keine Hardware–Fließkommaeinheit besitzt, Evolutionsstrategien jedoch auf der Fließkommarechnung basieren. Dadurch läßt sich keine große Effizienz bei der Ausführung erwarten.

In der Abbildung kann man deutlich die vier resultierenden Arten von Implementierungen erkennen. Sie ergeben sich aus den Kombinationen der zwei Arten von Evolutionären Algorithmen – Genetische Algorithmen und Evolutionsstrategien – mit den zwei grundlegenden Arten von Parallelrechnern – SIMD und MIMD (s. Kapitel 3.1), hier als massiv parallele und verteilte Architektur aufgeführt.

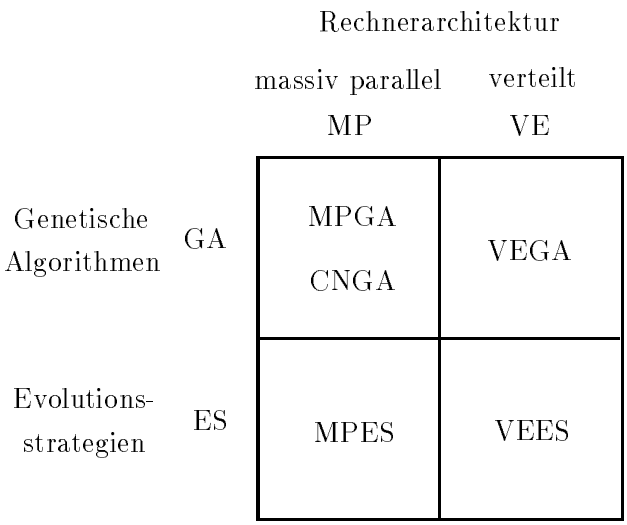


Abbildung 1.1: Übersicht der entstandenen Programme im EA-Projekt

1.4 Rahmenbedingungen

In diesem Abschnitt sind alle Rahmenbedingungen aufgelistet, unter denen das Programm VEES entwickelt wurde.

- **Hardware:** Entwickelt wurde das Programm sowohl auf SUN–Workstations als auch auf dem Parallelrechner Intel Paragon XP/S-5.
- **Betriebssystem:** Auf den Workstations wurde *SunOS Version 1 Release 4.1.3* verwendet. Die Paragon lief unter *Paragon OSF/1 paragon 1.0.4 R1.2.5*.
- **Compiler:** Es wurden die Compiler gcc Version 2.5.8 und Version 2.6.2, sowie icc Paragon Version R5.0.1 verwendet. Der icc–Compiler der Paragon hält sich sehr eng an den ANSI-C–Standard, erweitert diesen lediglich um eine Kommunikationsbibliothek mit Namen *nx*.
- **Vorhandener Code:** Das Programm sollte auf dem Code von VEGA aufbauen. Dazu wurde VEGA zuerst auf einheitliche Namenskonventionen gebracht, die auch in VEES, sowie in allen anderen Teilen des EA–Projektes verwendet wurden.
- **Weitere Software:**
  - **CVS:** Die Software *Concurrent Versions System* wurde zur Verwaltung der Quelldateien eingesetzt, damit auf einfache Weise mehrere Leute an demselben Programm Änderungen durchführen konnten. Dies war z. B. für die graphische Oberfläche nötig.
  - **Tcl:** Die Kommando–Sprache *Tcl* [Ousterhout 95] ist integraler Bestandteil der Oberfläche der Programme MPES, VEES und CNGA. In die Programme VEGA

und MPGA wurde sie nachträglich integriert. Sie dient sowohl zur interaktiven Benutzung, als auch zur Batcheingabe dieser Programme. Die Oberflächen lassen sich dadurch mit dem vollen Umfang der in Tcl verfügbaren Skript–Kommandos programmieren. Tcl ist ein frei verfügbares Softwarepaket.

- **Gnuplot:** Das Programm *Gnuplot* zur Darstellung von Funktionsgraphen wird in VEES dazu benutzt, um den Verlauf statistisch erfaßter Größen zu veranschaulichen. Gnuplot ist eine frei erhältliche und kopierbare Software der GNU[2] Free Software Foundation.

Da das Programm MPES ungefähr zur gleichen Zeit wie VEES entstanden ist, wurden einige interne Schnittstellen und Ausgabe–Dateiformate beider Programme aufeinander abgestimmt und vereinheitlicht. Die textuelle Oberfläche der beiden Programme und die Schnittstelle dazu wurden von Steffen Görzig und Alexander Hasel entwickelt [Görzig 95, Hasel 95]. Die darauf aufsetzende Menüstruktur basiert auf demselben Code wie diejenige von MPES. Dadurch ist die Bedienung von MPES und VEES sehr ähnlich und eine einheitliche Benutzung möglich.

[2] *G*nu's *n*ot *U*nix

Kapitel 2

Evolutionäre Algorithmen

In diesem Kapitel wird eine Einführung in Evolutionäre Algorithmen gegeben, welche sich untergliedern lassen in die Genetischen Algorithmen und die Evolutionsstrategien. Der grundlegende Unterschied zwischen beiden ist die Art der Repräsentation von Individuen und die Methoden zum Fortschreiten im Lösungsraum. Bei Genetischen Algorithmen sind die Individuen einer Population weiträumiger über den Lösungsraum verteilt, wohingegen Evolutionsstrategien eher lokal im Lösungsraum dem Weg des steilsten Anstiegs folgen.

Zum allgemeinen Verständnis folgt zuerst ein kleiner Exkurs in die biologischen Grundlagen und eine Begriffsbildung. Dann wird die Vorgehensweise von Genetischen Algorithmen beschrieben und anschließend ausführlich auf Evolutionsstrategien eingegangen.

2.1 Biologischer Hintergrund

Die biologische Evolution ist ein Optimierungsprozeß mit dem die Natur Lebewesen ständig an die Umwelt anpaßt, in der sie leben. Über einen Zeitraum von ca. 4 Milliarden Jahren hat sie es geschafft, aus den ersten Bausteinen des Lebens hochkomplexe Lebewesen, wie z. B. den Menschen, zu schaffen. Dies wurde durch stetige Veränderung und Verbesserung vorhandener Lebewesen erreicht, wodurch sich neue Arten bildeten. Diese Erkenntnis ist jedoch nicht selbstverständlich und wurde lange Zeit angezweifelt.

Bis ins 18. Jahrhundert hinein hielt man die im Tier– und Pflanzenreich auftretenden verschiedenen Arten für die unveränderlichen Einheiten des Lebens und nahm an, daß es so viele Arten gebe wie am Anfang der Welt erschaffen wurden. Der erste, der davon sprach, daß die gegenwärtig lebenden Arten von früheren ausgestorbenen Formen abstammen, war J. B. Lamarck (1744–1829). Allgemein bekannt wurde die Abstammungslehre aber erst durch Charles Darwin (1809–1882). Er legte zugleich eine Erklärung für die Ursachen der Evolution vor. Es ist dies das Prinzip der natürlichen Auslese durch Mutation und Selektion. Er stellte die These auf, daß gut an ihre Umwelt angepaßte Lebewesen sich im Konkurrenzkampf um Nahrung und Paarungspartner besser durchsetzen können. Dadurch kann sich ihr Erbgut und die damit einhergehenden Eigenschaften besser verbreiten.

Ein weiterer bedeutender Wissenschaftler auf dem Gebiet der Genetik war Johann Gregor Mendel. Er erforschte die Vererbungsregeln und entdeckte dabei die dominant–rezessive und

intermediäre Vererbung. Sehr bekannt sind die Mendelschen Vererbungsregeln, welche Aussagen machen über die Häufigkeit des Auftretens von Erbmerkmalen in den Folgegenerationen nach einer Kreuzung von reinerbigen Individuen.

Die Erbsubstanz von Lebewesen ist in den *Chromosomen* gespeichert, welche sich im Zellkern jeder Zelle befinden. Der Stoff, aus dem die Chromosomen bestehen, ist die *Desoxyribonukleinsäure* (DNA). Sie hat die Struktur einer gewundenen Doppelhelix. Das Molekül besteht aus zwei Strängen, in denen sich Zucker- und Phosphatmoleküle abwechseln. An den Zuckermolekülen sind die beiden Stränge durch Basenpaare verbunden, ähnlich den Sprossen einer Leiter. Es gibt dabei 4 verschiedene Arten von *Basen*: *Adenin (A), Cytosin (C), Guanin (G) und Thymin (T)*. Es stehen sich dabei immer die komplementären Basen C und G, bzw. A und T gegenüber.

Lebewesen bestehen hauptsächlich aus *Proteinen* und aus mit Hilfe von Proteinen erzeugten Stoffen. Proteine wiederum sind aus 20 verschiedenen *Aminosäuren* zusammengesetzt. Eine Sequenz von jeweils drei aufeinanderfolgenden Basenpaaren der DNA codiert dabei eine Aminosäure. Nach dem Plan der DNA werden Aminosäuren produziert und zu Proteinen zusammengesetzt. Ein Abschnitt in der DNA, welcher für die Synthese eines kompletten Proteins zuständig ist, heißt Gen.

Das durch seine in den Erbanlagen gespeicherte Gen–Informationen charakterisierte Lebewesen wird als *Genotyp* bezeichnet. Das äußere Erscheinungsbild des Lebewesens, das vom Genotyp festgelegt ist, aber beim Entstehen des konkreten Individuums durch äußere Einflüsse noch leicht abgewandelt wird, heißt *Phänotyp*. Die phänotypische Vielfalt ist dadurch größer als die genotypische.

Die DNA wird durch Keimzellen während der sexuellen Fortpflanzung an Nachkommen weitergegeben. In den Keimzellen ist nur eine Hälfte des normalerweise doppelt vorhandenen Chromosomensatzes enthalten. Bei der Verbindung zweier Keimzellen während der Fortpflanzung wird der doppelte Chromosomensatz wieder hergestellt. Während sich zwei Chromosomen miteinander verbinden, kann es vorkommen, daß die beiden Molekülketten an derselben Stelle aufbrechen und vertauscht werden (siehe Abbildung 2.1). Dieser Vorgang wird *Crossover* genannt und führt dazu, daß die entstehenden Chromosomen sich von den elterlichen Chromosomen unterscheiden. Je nach Anzahl der Bruchstellen, handelt es sich um *n-Punkt–Crossover*.

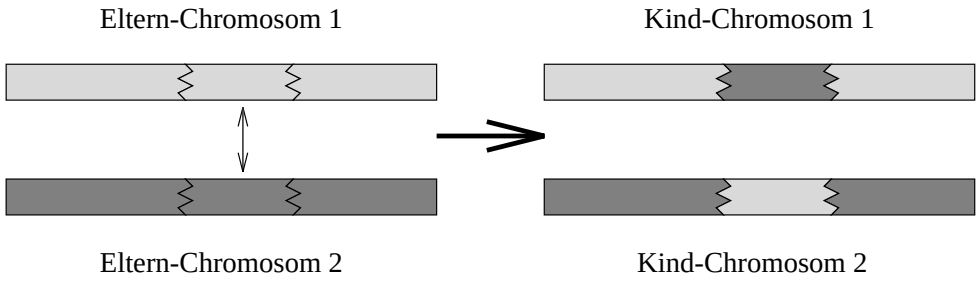


Abbildung 2.1: 2-Punkt–Crossover zweier Chromosomen

Weitere Veränderungen der Chromosomen können durch andere Einflüsse herbeigeführt werden, z. B. durch radioaktive Strahlung. Sie führt dazu, daß die Molekülketten der Chromoso-

men auseinanderbrechen oder sich durch die Energiezufuhr chemisch verändern. Radioaktive Strahlung ist auf der Erde in Form der natürlichen Hintergrundstrahlung praktisch überall in kleinem Maße vorhanden.

Durch die geschilderten Mechanismen wird das Erbgut und dadurch der Genotyp von Lebewesen verändert. Dadurch verändert sich auch der Phänotyp und somit die Eigenschaften und Fähigkeiten des Individuums. Die Veränderungen des neuen Lebewesens in Bezug auf seine Eltern können sowohl positiver, als auch negativer Natur sein. Positive Veränderungen können z. B. dazu führen, daß das Individuum besser als andere in Lage ist, Nahrung zu finden. Es überlebt länger und somit sind seine Chancen größer, sich fortzupflanzen und sein Erbgut mitsamt den positiven Veränderungen an seine Nachkommen weiterzugeben. Dadurch verbreiten sich positive Veränderungen innerhalb einer Art.

2.2 Begriffe

In den Evolutionären Algorithmen wird versucht, den Prozeß der biologischen Evolution zu modellieren. Dabei werden einige Begriffe gebraucht, die aus der Biologie stammen. Diese werden hier in dem Sinn erklärt, in dem sie bei den Evolutionären Algorithmen verwendet werden.

Genotyp
Der Genotyp ist die codierte Erbinformation eines Individuums und entspricht den Chromosomen in der Natur. Er besteht aus kleineren Einheiten, den Genen. Im Genotyp ist eine Lösung des Optimierungsproblemes vollständig beschrieben. Die Gene werden in Genetischen Algorithmen durch Bits[1] repräsentiert, der Genotyp ist somit eine Kette von Bits, auch Bitstring genannt. In Evolutionsstrategien sind die Gene reellwertige Variablen.

Phänotyp
Der Phänotyp ist die konkrete Realisierung einer Lösung des Optimierungsproblems. Er ist vollständig vom Genotyp bestimmt. In Evolutionären Algorithmen ist die Abbildung vom Genotyp zum Phänotyp eindeutig. Es kann normalerweise nicht vorkommen, daß aus demselben Genotyp verschiedene Phänotypen hervorgehen.

Individuum
Ein Individuum stellt eine einzelne Lösung für die zu optimierende Funktion dar. Die Lösung wird dabei durch seine Gene, deren Gesamtheit der Genotyp ist, codiert. Ein Individuum besitzt sowohl die Genotyp- als auch die Phänotyp–Ausprägung.

Population
Eine Population ist eine Menge von Individuen, die sich denselben Lebensraum teilen und somit untereinander konkurrieren.

Generation
Eine Generation ist ein Schritt eines Evolutionären Algorithmus, in dem aus der vorhandenen Population neue Individuen gebildet werden und alte Individuen verworfen werden, d. h. naturanalog, daß sie sterben.

Rekombination
Dies ist der Prozeß, wenn Erbinformationen verschiedener Individuen — den Eltern — ge-

[1] **Bi**nary digi**ts**, kleinste Informationseinheit in Computern

mischt werden, um ein neues Individuum — das Kindindividuum — zu bilden. In Genetischen Algorithmen wird dieser Vorgang auch **Crossover** genannt. In Evolutionären Algorithmen ist es ohne weiteres möglich, daß mehr als zwei Individuen an einer Rekombination beteiligt sind, dies wird dann *Multirekombination* genannt. In der Natur hat ein Individuum in den allermeisten Fällen nur zwei Eltern. Ausnahmen hiervon gibt es nur bei Viren.

Mutation
Mutation ist der Vorgang der zufälligen Veränderung von Genen. Da in Genetischen Algorithmen die Gene durch Bits repräsentiert sind, ist hier die Mutation das Umschalten eines Bits in den komplementären Zustand. Bei Evolutionsstrategien bedeutet Mutation eine Verschiebung eines Individuums im Vektorraum der Lösungen um einen Vektor von zufälligem Betrag (innerhalb gewisser Grenzen) und zufälliger Richtung.

Qualitätsfunktion
Die Qualitätsfunktion — auch Fitnessfunktion — dient der Berechnung der Qualität oder Fitness eines Individuums. Die Qualität ist ein direktes Maß für die Tauglichkeit, sie bestimmt die Güte der Lösung im Verhältnis zu anderen Lösungen.

Selektion
Die Selektion ist eine Auswahlprozeß unter Individuen. Die Selektion bestimmt, welche Individuen einer Population überleben und in die nächste Generation übernommen werden. Dazu gibt es verschiedene Verfahren. Mit Selektion kann aber auch der Auswahlprozeß gemeint sein, der Eltern bestimmt, die zur Bildung von Kindindividuen herangezogen werden. Dann wird aber meist von Elternauswahl gesprochen.

2.3 Genetische Algorithmen

Genetische Algorithmen wurden erstmals von Holland vorgestellt [Holland 75, Holland 93]. Später wurden sie von Goldberg [Goldberg 89] verfeinert. Genetische Algorithmen versuchen insbesondere den Aspekt der Chromosomen aus der Natur zu modellieren. Ein Individuum besteht aus einem Chromosom, dessen Gene durch Bits repräsentiert sind. Durch diese Bits ist die Lösung des Optimierungsproblems codiert. Soll die Lösung bewertet oder ausgegeben werden, so muß zuerst die Bitfolge decodiert werden. Dies ist der Vorgang der Umwandlung vom Genotyp in den Phänotyp. Dabei stehen mehrere Möglichkeiten der Codierung zur Verfügung. Die beiden meist verwendeten sind die Binärcodierung, und die Codierung nach Gray [Gray 53].

Der Grundalgorithmus ist in Abbildung 2.2 dargestellt. Zuerst wird eine Startpopulation initialisiert. Dabei werden die Chromosomen der Individuen mit zufälligen Bitmustern vorbelegt. Dadurch sind die Individuen einigermaßen gleichmäßig über den Lösungsraum verteilt. Diese Individuen werden dann mit der Qualitätsfunktion bewertet.

Nun folgt die Hauptschleife des Algorithmus. Es werden bestimmte Individuen ausgewählt, die zur Fortpflanzung herangezogen werden. Auswahlkriterium ist in den meisten Fällen die Qualität. Es stehen verschiedene Auswahlverfahren zur Verfügung, z. B. *Roulette Wheel*, *Deterministic Sampling*, *Stochastic Remainder* oder *Ranking*. Das Roulette Wheel Verfahren und Ranking werden im Abschnitt 2.4.5 näher erläutert.

Die Funktionsweise der Fortpflanzung ist eng an das biologische Vorbild angelehnt und besteht aus dem *Crossover* (s. Abb. 2.1) der Bitketten der Elternindividuen. Anschließend werden die

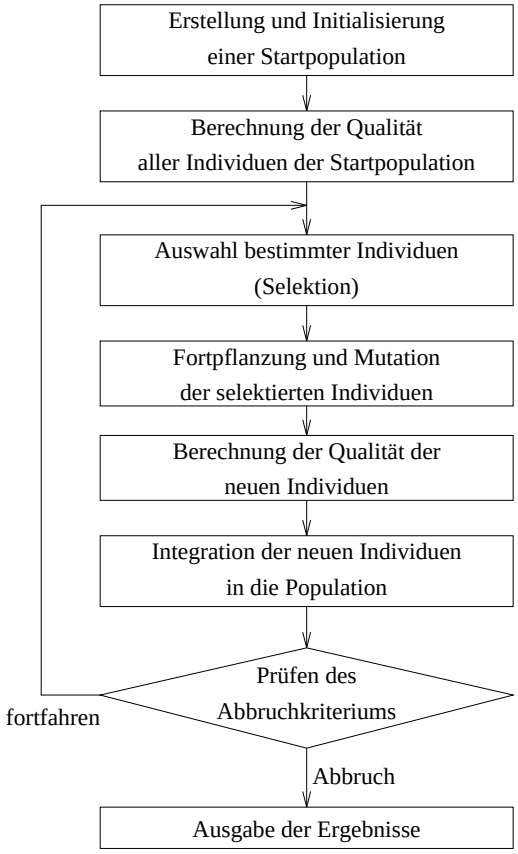


Abbildung 2.2: Ablauf eines Genetischen Algorithmus

neu entstandenen Individuen mutiert, d. h. zufällig ausgewählte Bits werden von 0 nach 1 oder von 1 nach 0 umgedreht. Es existieren aber auch andere Mutationsoperatoren, die auf die spezielle Problemstellung abgestimmt sind. Der Hauptoperator der Genetischen Algorithmen zur Erzielung neuer Lösungen und zum Fortschreiten im Lösungsraum ist allerdings das Crossover. Der Prozentsatz der mutierten Bits ist normalerweise sehr klein (Mutationsrate von ca. 0.1%). Er dient der Verhinderung der frühzeitigen Konvergenz der Lösungen in lokalen Optima durch Erhöhung der Vielfältigkeit der vorkommenden Chromosomen.

Die neu erzeugten Individuen werden dann in die vorhandene Population integriert. Auch hierbei gibt es verschiedene Möglichkeiten:

- Ein zufällig ausgewähltes Eltern–Individuum wird durch eines der neu erzeugten Kinder ersetzt.
- Kinder ersetzen irgendwelche Eltern–Individuen, die schlechter als sie selbst sind *(elitäre Ersetzung)*.
- Es überleben grundsätzlich alle Kinder. Eventuell noch übrige Plätze in der Population werden durch Eltern–Individuen belegt *(Generationenersetzung)*.
- Ein Kind ersetzt einen seiner direkten Eltern, falls es besser als dieser ist *(direkte Elternersetzung)*.

Am Ende der Hauptschleife des Genetischen Algorithmus wird ein Abbruchkriterium geprüft. Ist dieses nicht erfüllt, so wird die nächste Generation berechnet, sonst endet der Algorithmus und die Ergebnisse werden ausgegeben. Das Abbruchkriterium kann das Erreichen einer maximalen Rechenzeit oder einer bestimmten Anzahl von Generationen sein. Alternativ kann auch das Auftreten eines Individuums mit einer Qualität, die besser ist als eine vorgegebene Grenze (Konvergenzgrenze) zur Beendigung des Algorithmus führen.

An dieser Stelle soll noch kurz auf einen Hauptaspekt der Genetischen Algorithmen eingegangen werden, der am Anfang dieses Abschnitts kurz genannt wurde: die Codierung der Gene durch Bits. Die Verwendung unterschiedlicher Codierungsarten (binär oder Gray) ist folgendermaßen motiviert: bei der Binärcodierung kann es vorkommen, daß das Umkippen eines einzelnen Bits, wie es in der simulierten Mutation gemacht wird, dazu führt, daß aus einer codierten Zahl eine andere Zahl wird, die sich um einen sehr großen Betrag von der ursprünglichen unterscheidet. Kippt man z. B. bei der Binärzahl $11000000_2 = 192_{10}$ das höchstwertige Bit um, so wird daraus die Zahl $01000000_2 = 64_{10}$. Dies bedeutet, daß das Gesetz der starken Kausalität (näheres dazu siehe Abschnitt 2.4.2) nicht gilt, d. h. die Größe der Änderung der Gene (1 Bit) steht in keinem Verhältnis zur Größe der Änderung im Phänotypen (Betrag von 128) und damit indirekt der Qualität des Individuums. Dies kann zu erheblichen Problemen bei der Konvergenz führen. Durch Verwendung der Gray–Codierung versucht man dieses Problem zu vermeiden. Bei dieser Codierung bewirkt eine Änderung von einem Bit auch nur eine kleine Änderung der durch die Bitkette codierten Zahl. Solche Probleme treten bei Evolutionsstrategien nicht auf, da hier die Individuen nicht durch Bitketten codiert werden.

2.4 Evolutionsstrategien

Ein weiterer Vertreter der Evolutionären Algorithmen sind die Evolutionsstrategien. Die dahinterstehende Idee ist grundsätzlich die gleiche: Es wird versucht, den Prozeß der biologischen Evolution als allgemeine Optimierungsstrategie auf technischen Systemen zu modellieren. So soll eine selbststeuernde Fortentwicklung von einfachen Objekten nach einem vorgegebenen Optimierungskriterium stattfinden. Der wesentliche Unterschied zu den Genetischen Algorithmen ist dabei die Repräsentation von Individuen. Dadurch verändert sich zwangsläufig auch die Realisierung der evolutionsstrategischen Methoden, wie z. B. Rekombination und insbesondere Mutation. Außerdem sind Evolutionsstrategien gut mathematisch erfaßbar und analysierbar, weil sie weniger eine mechanische Imitation der natürlichen Prozesse sind, sondern versuchen, die Wirkung der biologischen Evolution nachzubilden. In den folgenden Abschnitten soll das Prinzip und die Vorgehensweise der Evolutionsstrategien anschaulich dargestellt werden. Die genauen Formeln, wie sie im Programm VEES implementiert wurden, werden im Kapitel 4, Abschnitt 4.2 gezeigt.

2.4.1 Entstehungsgeschichte

Evolutionsstrategien wurden in den 60er Jahren in Deutschland entwickelt. Entwickler der ersten Stunde waren Ingo Rechenberg, Hans–Paul Schwefel, Peter Bienert, Hans–Jürgen Lichtfuß und Armin Scheel. Die ersten Experimente fanden nicht im Computer statt, sondern beschäftigten sich mit technischen Optimierungen an konkreten Objekten.

Der erste Versuch wurde im August 1964 am Hermann–Föttinger–Institut für Strömungstechnik der Technischen Universität Berlin durchgeführt. Versuchsobjekt war dabei eine ziehharmonikaförmig gefaltete Fläche, die sogenannte Gelenkplatte (s. Abb. 2.3). Sie sollte sich von einer zufälligen Anfangsform im Windkanal zur der Form mit dem geringsten Strömungswiderstand entwickeln. Die optimale Form war dabei bekannt: es ist die ebene Platte. Die Gelenkplatte bestand aus 6 Segmenten, die mit 5 Gelenken verbunden waren. Jedes Gelenk hatte 51 Einraststufen im Abstand von 2°, wodurch sich $51^5 = 345.025.251$ mögliche verschiedene Formen der Platte ergeben. Die Winkel wurden jeweils um eine zufällige Zahl an Raststufen zwischen +5 und -5 verändert. Hierbei wurde eine Binominalverteilung verwendet, die durch das Werfen von 10 Scheibchen mit „+“ und „−“ Aufschrift erzeugt wurde. Die Binominalverteilung bevorzugt dabei kleine Änderungen gegenüber großen. Nach dieser Mutation wurde der Strömungswiderstand der neu eingestellten Platte im Windkanal gemessen. Stellte sich die neue Form als strömungsgünstiger heraus, so wurde sie als Ausgangsbasis für das nächste Experiment genommen, andernfalls verworfen.

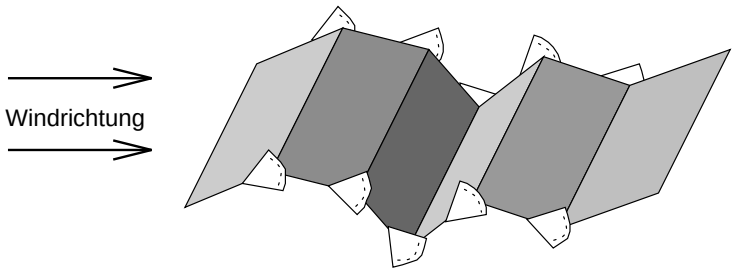


Abbildung 2.3: Das erste Evolutionsstrategie–Objekt, eine Gelenkplatte im Windkanal

Nach durchschnittlich 200 Generationen wurde die Optimalform der ebenen Platte erreicht. Allerdings waren nicht alle Winkel exakt 0, was an der Ungenauigkeit der Meßmethode lag. Mit den Evolutionsstrategien wurden noch andere Objekte optimiert, z. B. ein Rohrkrümmer auf geringsten Durchflußwiderstand, eine Überschalldüse auf maximalen Wirkungsgrad und ein PID–Regelglied auf minimale Ausschwingzeit nach einer Störung der Regelgröße [Rechenberg 73].

Praktische Versuche kosten allerdings Zeit und Geld. Deshalb wird versucht, den aufwendigen Teil der Realisierung und der Messung durch Computersimulationen zu ersetzen. Im Computer läßt sich eine Variable leicht abändern und eine Formel mit den neuen Werten berechnen. In der realen Welt muß z. B. wie beim Gelenkplattenexperiment ein Winkel verstellt und eine Widerstandmessung im Windkanal durchgeführt werden, was ungleich aufwendiger ist.

2.4.2 Grundlagen

Die Qualitätsfunktion

Das Optimierungsproblem wird durch den *Problemraum* beschrieben. Er besteht aus einer Menge von variablen Größen. Beim Gelenkplattenexperiment waren dies die fünf einstellbaren

Winkel. Die Anzahl der Variablen ist die *Dimension* des Problemraumes (das Gelenkplattenproblem hat entsprechend fünf Dimensionen). Hat der Problemraum die Dimension n, so beschreibt die Qualitätsfunktion einen Raum der Dimension $n+1$, denn sie nimmt als Eingabe die n Größen des Problemraumes und bildet sie auf eine weitere Größe, die Qualität, ab. Für einen Problemraum mit 2 Dimensionen (graphisch als Ebene darstellbar), läßt sich die Qualitätsfunktion dreidimensional darstellen. Jeder Punkt in der Ebene ist dabei ein Punkt im Problemraum, welchem durch seine Höhe ein Qualitätswert zugeordnet ist.

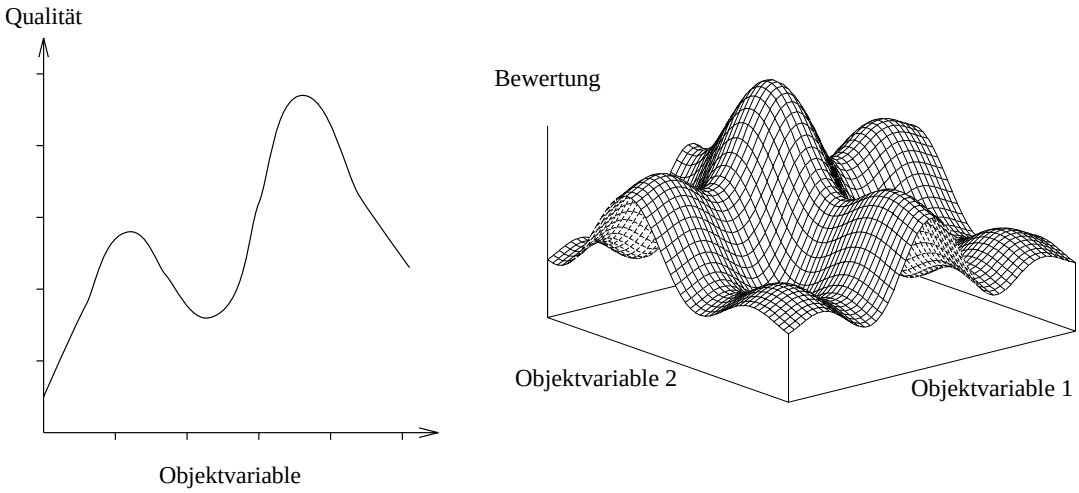


Abbildung 2.4: Darstellung einer Qualitätsfunktion für 1- und 2-dimensionale Problemräume

Das Ziel der Evolutionsstrategie ist es nun, den „Gipfel“ dieses „Qualitätsgebirges“ zu erklimmen, also den höchsten Punkt zu finden. Besitzt das Problem sehr viele Dimensionen, so läßt es sich nicht mehr anschaulich als Gebirge darstellen. Evolutionsstrategien erweisen aber gerade in hochdimensionalen Räumen ihre Stärke. Ein Beispiel für ein komplexes Problem der Dimension 100 ist das *Traveling Salesman Problem*, welches in Abschnitt 5.1.2 vorgestellt wird.

Eine wichtige Eigenschaft der Qualitätsfunktion, auf die Evolutionsstrategien aufbauen, ist die *starke Kausalität*. Zur Erläuterung sei zuerst das Prinzip der *schwachen Kausalität* genannt: „gleiche Ursachen haben die gleiche Wirkung“. Das bedeutet, daß zwei identische Individuen dieselbe Qualität besitzen. Bei auf dem Computer berechneten Qualitätsfunktionen ist dies fast immer gegeben. In der Praxis aber, wenn die Qualität gemessen wird, gilt dies nicht unbedingt. Außerdem nutzt diese Eigenschaft allein nicht viel, denn es wird keine Aussage darüber gemacht, wie sich die Qualität ändert, wenn sich das Individuum ein klein wenig ändert. Springt in diesem Falle die Qualität wild von einem Wert zum anderen, so kann eine Evolutionsstrategie nicht erfolgreich sein. Deshalb muß das Prinzip der starken Kausalität erfüllt sein: „ähnliche Ursachen haben ähnliche Wirkungen“. In diesem Falle wird sich die Qualität nur leicht ändern, wenn sich das Individuum auch nur wenig ändert. Es wird also nicht nur eine Aussage über einen Punkt der Qualitätsfunktion gemacht, sondern auch über seine Umgebung. Ist die Qualitätsfunktion stark kausal organisiert, so kann die Evolutionsstrategie dem Qualitäts–Anstieg folgen.

Das Individuum

Ein Individuum ist ein bestimmter Punkt im Problemraum, welcher durch einen Vektor von Koordinaten beschrieben wird. Sehr wichtig ist hierbei, daß jede Koordinate eine reelle Zahl ist. Ein Individuum eines zweidimensionalen Raums wird also durch ein Zahlenpaar beschrieben, z. B. $(3.4, 2.7)$. Diese reellwertigen Zahlen heißen *Objektvariablen*. Das durch die Qualitätsfunktion bewertete Individuum läßt sich anschaulich als Punkt auf der Oberfläche des Qualitätsgebirges, wie es in Abbildung 2.4 dargestellt ist, vorstellen.

Ein Anmerkung zur Darstellung von Qualitätsgebirgen: Um eine Funktion graphisch darzustellen, muß man im zu zeichnenden Bereich eine Menge von ausreichend fein verteilten Punkten ausrechnen. Diese Darstellung ist dann vom Menschen leicht zu verstehen; man sieht die Beschaffenheit des Gebirges und kann erkennen, wo das Optimum der Funktion liegt. Genau das wird aber in einer Evolutionsstrategie nicht gemacht! Die Strategie verschafft sich keinen globalen Überblick, um das Optimum zu suchen. Ihr Vorteil ist, daß nur wenige Punkte im (oft sehr großen) Suchraum berechnet werden müssen. Anhand derer wird dann dem lokalen Anstieg zum Gipfel gefolgt. Dies kann man sich vorstellen, als wäre die Sicht auf das Qualitätsgebirge durch „Nebel“ verschleiert, der die Blickweite derart einschränkt, daß der Gipfel nicht sichtbar ist.

Die dargestellten Qualitätsgebirge sind so ‘einfach’, daß sie komplett graphisch veranschaulichbar und für eine Evolutionsstrategie sehr leicht zu lösen sind. Mit steigender Anzahl der Dimensionen eines Optimierungsproblems wird es jedoch schnell unmöglich, durch eine breite Suche durch den gesamten Problemraum das Optimum zu finden. Dies ist einfach deshalb so, weil der Suchraum um Größenordnungen zu groß ist, als daß er in einem vernünftigen Zeitraum durchsucht werden könnte. Für diesen Zweck kommen dann z. B. Evolutionsstrategien zum Einsatz.

2.4.3 Einfache Evolutionsstrategien

Evolutionsstrategien lassen sich in verschiedene Klassen, abhängig von ihrem Grad, einteilen. In diesem Abschnitt werden die einfachen My-Lambda–Strategien erläutert. Im nächsten Abschnitt wird dann auf die Strategien der nächst höheren Ordnung, die geschachtelten My-Lambda–Strategienk, eingegangen. Noch tiefere Schachtelungen sind in der Praxis nicht sinnvoll, deshalb wird die Idee nur kurz angerissen, aber nicht weiter ausgeführt.

Für Evolutionsstrategien existiert eine formale Notation, die von Hans-Paul Schwefel eingeführt worden ist [Schwefel 95]. In dieser Notation lassen sich My-Lambda–Evolutionsstrategien allgemein folgendermaßen beschreiben:

$$(\mu/\rho^{+}, \lambda)^{\gamma}$$

Hierbei gelten die folgenden Bezeichnungen:

- μ Anzahl der Elternindividuen
- ρ Anzahl der zu rekombinierenden Individuen
- λ Anzahl der erzeugten Kindindividuen
- γ Anzahl der Generationen, die berechnet werden

Eine konkrete Strategie in dieser Notation ist z. B. eine $(5/2, 20)^{50}$–Evolutionsstrategie.

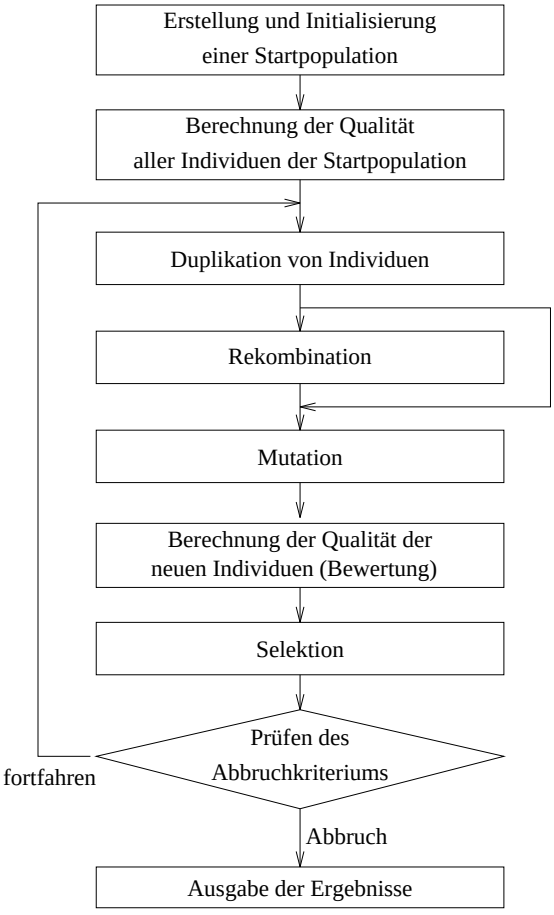


Abbildung 2.5: Ablauf einer einfachen My-Lambda–Evolutionsstrategie

In Abbildung 2.5 ist der allgemeine Ablauf einer Evolutionsstrategie dargestellt. Zunächst soll die Variante $(\mu\overset{+}{,}\ \lambda)^{\gamma}$ ohne Rekombination erläutert werden. Es wird mit einer Population von μ zufällig erzeugten Individuen — den Eltern — gestartet. Zufällig ausgewählte Eltern erzeugen Kinder, indem das Elternindividuum dupliziert und um eine gewisse Entfernung vom ursprünglichen Platz verschoben wird. Dies ist der Vorgang der Mutation. Insgesamt werden λ mutierte Nachkommen erzeugt. Nun liegen also $\mu + \lambda$ Individuen vor. Diese Menge wird durch die Selektion wieder verkleinert. Ist der Strategietyp *Plus* — Notation $(\mu + \lambda)^{\gamma}$ — so werden aus der Menge der μ Eltern *plus* den λ Kindern μ Individuen ausgewählt. Diese bilden dann die Eltern der nächsten Generation. Beim Strategietyp *Komma* — Notation: $(\mu,\ \lambda)^{\gamma}$ — werden die μ Elternindividuen der nächsten Generation nur aus der Menge der λ Kindindividuen ausgewählt. Die bisherigen Elternindividuen werden hier grundsätzlich alle verworfen.

Dieser Ablauf wird γ Generationen lang durchgeführt. Benutzt die Strategie Rekombination, so wird dies durch die Anzahl ρ der zu rekombinierenden Individuen angegeben: $(\mu/\rho\overset{+}{,}\ \lambda)^{\gamma}$. Rekombination wird auf den duplizierten Individuen durchgeführt, welche anschließend noch mutiert werden.

2.4.4 Geschachtelte Evolutionsstrategien

In geschachtelten My-Lambda–Evolutionsstrategien sind die einfachen My-Lambda–Strategien als Untermenge enthalten. Sie erweitern diese um Evolutionsstrategische Methoden auf Populationsebene. Die allgemeine Notation von (einfach) geschachtelten Evolutionsstrategien sieht folgendermaßen aus:

$$[\mu'/\rho'\overset{+}{,}\ \lambda'(\mu/\rho\overset{+}{,}\ \lambda)^{\gamma}]^{\gamma'}$$

Die neu hinzugekommenen Bezeichner haben einen Strich, welcher bedeutet, daß es sich um eine Größe auf Populationsebene handelt.

- μ' Anzahl der Elternpopulationen
- ρ' Anzahl der zu rekombinierenden Populationen
- λ' Anzahl der erzeugten Kindpopulationen
- γ' Anzahl der Zyklen, die auf Populationsebene berechnet werden

Die geschachtelte Strategie wendet dasselbe Schema wie die My-Lambda–Strategie an, doch sie arbeitet neben Individuen auch mit ganzen Populationen. Es wird mit μ' Elternpopulationen gestartet. Davon werden jeweils ρ' verwendet, um daraus eine Kindpopulation zu rekombinieren. Rekombination von Populationen bedeutet dabei, daß alle Individuen der ρ' Populationen zu einer großen gemeinsamen Population zusammengefaßt werden und die große Population dann durchmischt wird. Anschließend werden daraus wieder ρ' neue Teilpopulationen gewonnen, von denen eine als Ergebnis der Rekombination zufällig ausgewählt wird. Die Rekombination wird so oft durchgeführt, bis λ' Kindpopulationen vorliegen, welche dann im nächsten Schritt mutiert werden. Ist $\rho' = 1$, so braucht dieser Parameter nicht angegeben zu werden und die zu mutierenden Populationen werden ohne Rekombination direkt aus den Elternpopulationen ausgewählt.

Der prinzipielle Unterschied zwischen der Rekombination von Individuen und von Populationen soll in diesem Absatz noch einmal deutlich herausgestellt werden. Individuen sind quasi Listen von Objektvariablen. Eine Objektvariable hat aber ihren festen Platz in der Liste und kann nicht etwa mit einer anderen vertauscht werden. Deshalb beläßt die Rekombination auf Individuenebene die Objektvariablen an ihrem Platz. Bei der dominanten Rekombination wird für jeden Platz entschieden, von welchem Individuum die Objektvariable übernommen wird. Eine Population ist eine Liste von Individuen. Individuen sind aber innerhalb der Population vertauschbar. Deshalb wäre es unsauber, die Rekombinationsmethoden für Objektvariablen anzuwenden. Aus diesem Grund wird das oben genannte Mischen der Population durchgeführt. Dadurch bekommt jedes Individuum die gleiche Chance, in die rekombinierte Population zu gelangen.

Die Mutation der λ' Kindpopulationen geschieht durch Verschieben der gesamten Population um einen bestimmten Vektor im Problemraum. Die Schrittweite dieser Verschiebung ist um einige Größenordnungen größer als die Schrittweite der Individuen. So werden die Populationen aus ihrem lokalen Einzugsgebiet herausgebracht und können nun neue Gebiete durchsuchen.

Die λ' mutierten Kindpopulationen führen nun γ Generationen lang das bekannte My-Lambda–Schema durch. In dieser Zeit hat jede Population keinerlei Kontakt zu den anderen Populationen, weshalb γ auch *Isolationsintervall* genannt wird. Nachdem dieses abgelaufen ist, werden die Populationen mit einer Qualitätsfunktion Q' speziell für Populationen bewertet. Einfache Möglichkeiten für Q' sind, daß die Populationsqualität gleich der Qualität des besten Individuums oder der durchschnittlichen Qualität aller Individuen der Population gesetzt wird.

Nun folgt die Selektion, bei der je nach Strategietyp auf Populationsebene (Plus oder Komma) aus den λ' bzw. $\lambda' + \mu'$ Populationen μ' ausgewählt werden. Dies geschieht nach dem Kriterium der Populationsqualität Q'. Dieser Zyklus auf Populationsebene, der γ Zyklen auf Individuenebene beinhaltet, wird γ'-mal durchgeführt.

Zusammenfassend läßt sich das Prinzip der geschachtelten My-Lambda–Strategien wie folgt beschreiben: Populationen werden rekombiniert und setzen in einer gewissen Entfernung Kindpopulationen ab, die dann für eine bestimmte Zeit isoliert ablaufen. Anschließend müssen sich die Kindpopulationen im Selektionswettstreit untereinander messen und werden dadurch wieder auf die ursprüngliche Anzahl reduziert. Dann kann das Spiel von neuem beginnen.

An dieser Stelle sei auf die biologischen Analogien der geschachtelten Evolutionsstrategien hingewiesen. Auch in der natürlichen Evolution kommt es vor, daß verschiedene Arten (mehrere Populationen) um denselben Lebensraum konkurrieren. Die besser angepaßte Art wird hierbei die schlechtere verdrängen. Die große Mutationsschrittweite, die zur Verschiebung von

Populationen benutzt wird, hat ihre Analogie in der biologischen *Großmutation*. Die Frage, ob die beobachtbaren Mutationen von Einzelmerkmalen bei Individuen ausreichen, um die Entstehung ganz neuer Arten zu erklären, ist noch nicht eindeutig gelöst. Deshalb wird die Existenz von Großmutationen angenommen, durch die Nachkommen entstehen sollen, die sich sehr von den Eltern unterscheiden. Schließlich ist auch die Isolation in der Natur zu beobachten. Ist eine Population geographisch von anderen Populationen getrennt, so wird sie sich je nach Vollständigkeitsgrad und Dauer der Isolation in eine eigene Richtung entwickeln.

Das Prinzip der geschachtelten Strategien läßt sich auf beliebig höhere Schachtelungsstufen erweitern:

$$\cdots\{\mu''/\rho''^{+},\ \lambda''[\mu'/\rho'^{+},\ \lambda'(\mu/\rho^{+},\ \lambda)^{\gamma}]^{\gamma'}\}^{\gamma''}$$

Auf der dritten Ebene (mit Doppelstrich $''$ gekennzeichnete Größen) wetteifern dann Populationen von Populationen im Mutations-/Selektions-Kampf untereinander. Dies kann sich in beliebig höhere Ebenen fortsetzen. Ob Evolutionsstrategien mit einer Schachtelungstiefe von mehr als zwei in der Praxis sinnvoll sind, muß sich allerdings erst noch zeigen.

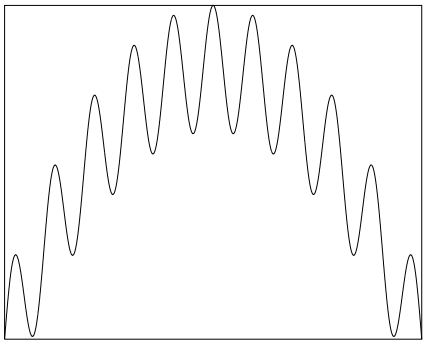


Abbildung 2.6: Zum Optimum hin ansteigende Folge von Bergen

Wie in Abschnitt 2.4.2 erläutert, sollte die Qualitätsfunktion stark kausal organisiert sein, damit eine My-Lambda–Strategie erfolgreich sein kann. Das bedeutet, daß die Spitze eines Berges in der Qualitätsfunktion in einer zusammenhängend ansteigenden Folge erreicht werden kann. Auch hier transformieren die geschachtelten Strategien die Zusammenhänge auf eine höhere Ebene. Damit die Schachtelung das globale Optimum, oder den höchsten Berg unter den Bergen, erreichen kann, müssen diese ebenfalls ansteigend angeordnet sein. Das bedeutet, daß der 'Berg der Berge' in einer zusammenhängenden, ansteigenden Folge von Schritten, quasi von Berg zu Berg springend, erreicht werden kann. In Abbildung 2.6 ist eine Funktion dargestellt, bei der die Anordnung der Berge die geforderte Eigenschaft erfüllt. Ob reale Probleme so beschaffen sind, bleibt fraglich. Auf jeden Fall kann der Problemraum mit geschachtelten Strategien weiträumiger durchsucht werden. Auch lokale Optima können wieder verlassen werden.

2.4.5 Evolutionsstrategische Methoden

In den folgenden Abschnitten werden die evolutionsstrategischen Methoden Duplikation, Rekombination, Mutation, Bewertung, Selektion und die Kappa–Ka Rauschbehandlung erläutert.

Abgesehen von der Rauschbehandlung werden die Methoden auch in dieser Reihenfolge in der Evolutionsstrategie angewendet. Es wird auch auf den Mechanismus der mutativen Schrittweitenregelung (im folgenden mit *MSR* abgekürzt) eingegangen. Die MSR ist eine äußerst wirksame Methode, mit der die Evolutionsstrategie ihr eigenes Fortschreiten optimiert. Die Methoden werden an dieser Stelle weniger formal, zum besseren Verständnis aber mit anschaulichen Beispielen beschrieben. Die konkrete formale Beschreibung, die in die Implementierung umgesetzt wurde, folgt in Abschnitt 4.2.

Duplikation

Bei Evolutionsstrategien bleiben die Elternindividuen einer Generation zunächst noch erhalten, da z. B. die Rekombination evtl. mehrmals auf dasselbe Individuum zugreift. Je nach Typ der Strategie, Komma oder Plus, werden die Eltern am Ende der Generation dann verworfen, oder sie werden mit in den Selektionsprozeß mit einbezogen.

Deshalb werden die Elternindividuen zuerst einmal *dupliziert*. Die Duplikation besteht einfach aus dem Kopieren des originalen Individuums. Die Kopie kann dann durch die Methoden Rekombination und Mutation verändert werden, wobei das ursprüngliche Individuum erhalten bleibt.

Rekombination

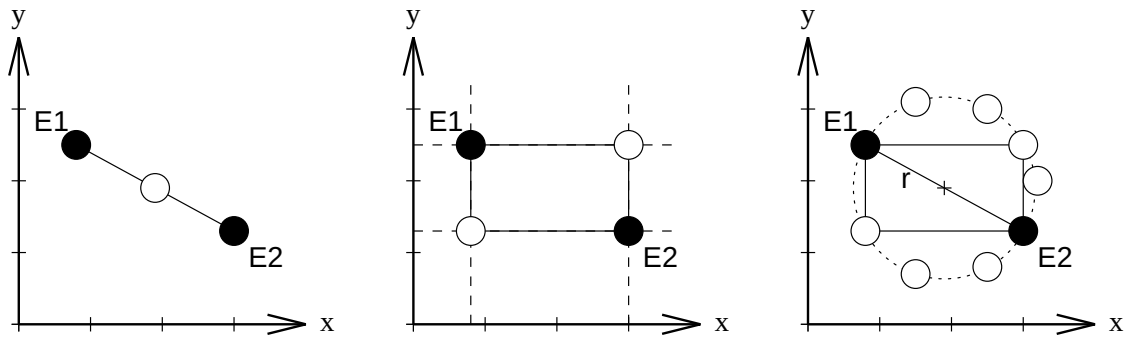
Das Mischen der Chromosomen mehrerer Individuen heißt *Rekombination*. Die Individuen, deren Chromosomen rekombiniert werden, heißen *Eltern*. Rekombination ist die Analogie zur sexuellen Fortpflanzung in der Biologie. Da die kleinste Einheit eines Chromosoms bei Evolutionsstrategien die Objektvariable ist, handelt es sich bei Rekombination also um Methoden zur Mischung von Objektvariablen. Hierbei werden drei verschiedene Arten unterschieden:

1. intermediäre Rekombination
2. dominante Rekombination
3. kontinuisierte Rekombination

Die intermediäre Rekombination (Abbildung 2.7a, Darstellung nach [Rechenberg 94]) bildet in jeder Dimension den Mittelwert der Objektvariablen aller Eltern. Die Lage des rekombinierten Individuums ist dabei eindeutig bestimmt. Es gibt im Gegensatz zu den anderen Rekombinationsarten nicht mehrere Möglichkeiten. Das resultierende Individuum liegt dabei im Schwerpunkt aller Elternindividuen.

Die dominante Rekombination ähnelt dem Crossover mit einem Crossoverpunkt nach jeder Objektvariable. Für jede Objektvariable wird unabhängig entschieden, von welchem Elter sie übernommen wird. Es dominiert also bei jeder Objektvariable genau ein Elter. In Abbildung 2.7b ist die dominante Rekombination zweier Eltern im 2-dimensionalen Problemraum dargestellt. Die ausgefüllten Punkte sind die Eltern, die nicht ausgefüllten Punkte die möglichen Nachkommen. Ein möglicher Nachkomme ist auch der unveränderte Elternpunkt. Dies ist dann der Fall, wenn jede Objektvariable von genau diesem Elter übernommen wurde. Die

Nachkommen liegen hier auf den Ecken eines Rechteckes, dessen Diagonale die Strecke des Elternabstandes ist. Bei ρ Eltern im n–dimensionalen Problemraum gibt es maximal ρ^n verschiedene Möglichkeiten für die Lage eines rekombinierten Individuums.



(a) intermediär (b) dominant (c) kontinuisiert

Abbildung 2.7: Rekombination im 2-dimensionalen Problemraum

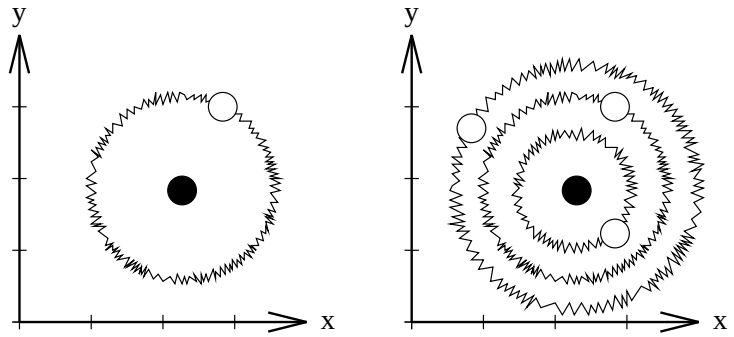
Die kontinuisierte Rekombination ist eine Verallgemeinerung der dominanten Rekombination. Zugrunde liegt hierbei die Beobachtung, daß beim dominanten Rekombinieren im 2–dimensionalen Problemraum die Individuen auf den Ecken eines Rechteckes um den Elternmittelpunkt zu liegen kommen. Die Diagonale des Rechteckes ist der Elternabstand, seine Seiten liegen parallel zu den Achsen des Koordinatensystems. Die Rekombinationspunkte sind also vom gewählten Koordinatensystem abhängig. Dreht man dieses, so erhält man andere Rekombinationsergebnisse. Berücksichtigt man alle Koordinatensystem–Drehungen auf einmal, so fallen die Rekombinanten auf die Linie eines Kreises, dem das Rechteck eingeschrieben ist. In Abbildung 2.7c ist die kontinuisierte Rekombination zweier Individuen im 2-dimensionalen dargestellt. Die Rekombinanten (helle Kreise) können dabei nicht nur die eingezeichneten Positionen einnehmen, sondern auf jedem beliebigen Punkt des Kreisumfangs liegen.

Erweitert man dieses Modell auf den n-dimensionalen Raum, so wird aus dem Rechteck ein Hyperkubus und aus dem Rekombinationskreis eine Hyperkugel, deren Mittelpunkt der Elternschwerpunkt ist. Für nur zwei Eltern bereitet es keine Schwierigkeiten, die Hyperkugel so zu bestimmen, daß die Elternindividuen auf ihrer Oberfläche liegen. Bei mehr als zwei Eltern, ist dies aber im allgemeinen nicht mehr möglich. Hier kann man mit einer anderen Konstruktion eine Annäherung an das obige Modell erreichen: Das Zentrum der Hyperkugel bildet weiterhin der Schwerpunkt aller Eltern. Als Radius wird dann die durchschnittliche Entfernung aller Elternindividuen vom Schwerpunkt genommen. Bei 2 Eltern deckt sich dies genau mit dem vorigen Modell.

Mutation

Mutation ist die zufällige Veränderung eines Individuums, so daß es sich ein klein wenig vom ursprünglichen Individuum unterscheidet. Als Ausgangsbasis kann dazu ein dupliziertes Elternindividuum oder ein aus der Rekombination mehrerer Eltern entstandenes Individuum ge-

nommen werden. Da es sich bei den Individuen um Punkte im Problemraum handelt, wird Mutation durch Addieren eines Vektors erreicht. Die Richtung des Mutationsvektors wird dabei zufällig gewählt, seine Länge heißt *Mutationsschrittweite*. Das mutierte Individuum kommt (im 2-dimensionalen Fall) mit gleichverteilter Wahrscheinlichkeit auf dem Umfang eines Kreises um das ursprüngliche Individuum zu liegen (Abbildung 2.8a). Die Mutationsschrittweite wird aber noch durch eine normalverteilte Zufallsgröße in gewissen Grenzen variiert. Dadurch ist der Radius des Kreises unscharf oder „verrauscht", weshalb die Kreisline in der Abbildung eine Zickzacklinie ist (Darstellung nach [Rechenberg 94]). Analog wird dann in höherdimensionalen Räumen der Mutationskreis zur Hyperkugel.



(a) ohne Mutations-Schrittweitenregelung (b) mit MSR

Abbildung 2.8: Mutation im 2-dimensionalen Problemraum

Nun kann es aber sein, daß das Qualitätsgebirge in manchen Dimensionen stark unterschiedliche Eigenschaften aufweist. In Richtung einer Dimension ändert sich die Qualität nur langsam, in Richtung einer anderen dafür sehr schnell. In diesem Fall ist eine einheitliche Schrittweite nicht sehr angebracht. Deshalb gibt es die Mutation auch mit *separaten* Schrittweiten. Dabei existiert dann für jede Objektvariable eine eigene Schrittweite, mit der sie mutiert wird. Die Hyperkugel, auf deren Oberfläche die möglichen Mutanten sitzen, wird dabei zum Hyperellipsoid verzerrt.

Ein äußerst wichtiger Mechanismus bei der Mutation ist die *Mutationsschrittweitenregelung (MSR)*. Erst durch die MSR ist es möglich, daß sich die Evolutionsstrategie an die lokale Form des Qualitätsgebirges anpaßt und die optimale Fortschrittsgeschwindigkeit erreicht. Ohne MSR werden nur Individuen auf dem Rauschkreis mit der normalen Schrittweite erzeugt (Abb. 2.8a). Die MSR mutiert nun auch die Schrittweite, d. h. es werden zusätzlich Individuen auf einem Rauschkreis mit etwas größerem und etwas kleinerem Radius erzeugt (Abb. 2.8b). Die Schrittweite mit der vom Elter aus mutiert wurde, wird dabei jedem Individuum mitvererbt. Individuen, deren Schrittweite besser den lokalen Verhältnissen angepaßt ist, werden in der Regel auch eine bessere Qualität erreichen. Dadurch wird sich die vorteilhaftere Schrittweite weiter in der Population vererben. Zusammenfassend läßt sich also sagen: bei der MSR unterwirft die Evolutionsstrategie ihr eigenes Fortschreiten dem Optimierungsprozeß.

Die MSR funktioniert sowohl für eine Schrittweite, als auch für separate Schrittweiten. Die

Schrittweite selbst wird dabei üblicherweise mit festen Faktoren mutiert, die über eine gleichverteilte Zufallswahl bestimmt werden. Beispielsweise erhält je ein Drittel der Nachkommen eine um den Faktor 2 größere, eine gleichbleibende und eine um den Faktor 1/2 verkleinerte Schrittweite.

Bewertung

Bei der Bewertung wird die Qualitätsfunktion auf jedes neue Individuum angewendet. Dadurch wird ihnen ein Qualitätswert zugeordnet. Anschaulich entspricht dieser der Höhe des Individuums im Qualitätsgebirge. Je größer der Qualitätswert ist, desto besser ist das Individuum. Die Qualität wird dann im nächsten Schritt von den Selektionsmethoden verwendet.

Welche Qualitätsfunktion zur Bewertung verwendet wird, hängt von der Problemstellung ab. Die Qualitätsfunktion definiert das Problem. Bei komplexen Problemen braucht die Bewertung die meiste (Rechen–)Zeit. Deshalb sollten die Strategieparameter so gewählt werden, daß möglichst wenig Individuen bewertet werden müssen. Dazu muß ein Kompromiß zwischen den Populationsgrößen und der Zahl der berechneten Generationen gefunden werden.

Selektion

Die Selektion entscheidet, welche Individuen als Eltern in die nächste Generation übernommen werden. Hierfür kommen in erster Linie die neu erzeugten Individuen in Frage, eventuell werden aber auch noch deren Eltern mit in den Auswahlprozeß einbezogen. Dieses Merkmal ist der *Typ* der Strategie, und kann die beiden Werte *Komma* und *Plus* annehmen (siehe Abschnitt 2.4.3). Plus bedeutet dabei, daß die Kinder *plus* ihre Eltern die Auswahlmenge darstellen, beim Typ *Komma* dagegen sind diese beiden Mengen getrennt, die Auswahlmenge besteht nur aus den Kindern.

Beim Strategietyp *Komma* werden die μ Elternindividuen der nächsten Generation aus der Menge der λ Kindindividuen ausgewählt. Ist der Typ *Plus*, so werden die μ Inidividuen aus der Menge der λ Kindindividuen plus der Menge der μ bisherigen Elternindividuen selektiert.

Damit eine Selektion überhaupt sinnvoll ist, muß die Menge, aus der ausgewählt wird, größer sein als die Zahl der tatsächlich auszuwählenden Individuen. Es wird also immer eine größere Anzahl auf eine kleinere reduziert. Deshalb muß bei einer Komma–Strategie λ größer sein als μ. Bei einer Strategie vom Typ Plus dagegen, kann λ frei gewählt werden, da $\mu + \lambda$ immer größer ist als μ. Das Verhältnis von μ zu λ (Plus–Strategie), bzw. von μ zu $\mu + \lambda$ wird *Selektionsdruck* oder *Selektionsstärke* genannt.

Es gibt einige verschiedene Selektionsverfahren, hier seien drei davon genannt:

1. Besten–Selektion
2. Roulette-Wheel–Selektion
3. Ranking–Selektion (lineare Rangauswahl)

Bei Evolutionsstrategien ist das ursprünglich benutzte Verfahren die *Besten–Selektion*. Hierbei werden einfach die μ besten Individuen selektiert. Dazu wird vorher die Population sortiert.

Die beiden anderen Verfahren finden ihre hauptsächliche Anwendung in Genetischen Algorithmen, können aber genauso in Evolutionsstrategien angewendet werden. Um die Verfahren mathematisch erläutern zu können, werden die folgenden Bezeichnungen verwendet:

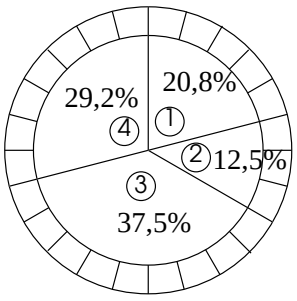
n	Anzahl der Individuen in der Population, aus der selektiert wird
μ	Anzahl der auszuwählenden Individuen
q	Qualität eines Individuums
j	Sortiernummer eines Individuums
r	Rang eines Individuums
g	Ranking–Gradient
s	Selektionswert eines Individuums
$P_s(ind_x)$	Selektionswahrscheinlichkeit des Individuums x

Bei der *Roulette-Wheel–Selektion* werden die Individuen mit einer Wahrscheinlichkeit proportional zu ihrer Qualität ausgewählt. Der Name des Verfahrens kommt von der bildlichen Vorstellung eines Roulette–Rades, auf dem die Vertiefungen, in die die Kugel fallen kann, den Individuen zugeordnet sind. Ein Individuum kann dabei auch mehreren Vertiefungen zugeordnet sein. Je besser die Qualität des Individuums ist, desto mehr Vertiefungen bekommt es (Abb. 2.9 links). Fällt die Kugel in eine Vertiefung, so gilt das zugeordete Individuum als selektiert.

Die Auswahlwahrscheinlichkeit eines bestimmten Individuums ergibt sich dann aus seiner Qualität geteilt durch die Summe aller auftretenden Qualitätswerte:

$$P_s(ind_x) = \frac{q_x}{\sum_{i=1}^{n} q_i} \qquad (1 \leq x \leq n) \tag{2.1}$$

Voraussetzung für die Anwendung dieser Formel und damit des Verfahrens ist, daß nur positive Qualitätswerte auftreten und daß eine größere Qualität ein besseres Individuum bedeutet.



	Individuum Nummer (x)	Qualität q_x	$P_s(ind_x)$
	1	75	20,8 %
	2	45	12,5 %
	3	135	37,5 %
	4	105	29,2 %
$\sum$	–	360	100,0 %

Abbildung 2.9: Beispiel für Roulette-Wheel–Selektion

Die *Ranking–Selektion* oder auch *lineare Rangauswahl* genannt, funktioniert ähnlich wie die Roulette-Wheel–Selektion. Hier wird allerdings zur Berechnung des Anteils an der Roulette–Scheibe nicht die absolute Qualität des Individuums benutzt, sondern sein *Rang* innerhalb der Population. Der Rang ist die Nummer des Individuums in der nach Qualitäten sortierten Population. Die Nummern werden, angefangen beim schlechtesten Individuum mit Rang $r = 0$, aufsteigend bis zum besten Individuum mit Rang $r = n - 1$ verteilt. Der Rang wird vor der

Anwendung des Roulette-Wheel–Prinzips in einen Selektionswert umgerechnet (Formel 2.2). Dazu wird der *Ranking Gradient* g verwendet, welcher bestimmt, wie stark bessere Individuen bevorzugt werden ($g \geq 0$). Dies ist eine lineare Umrechnung, welche dem Verfahren den Namen gab. Dann wird aus dem Selektionswert die Selektionswahrscheinlichkeit P_s des Individuums berechnet (Formel 2.3).

$$s_x = gr_x + 1 \tag{2.2}$$

$$P_s(ind_x) = \frac{s_x}{n + \sum_{i=1}^{n} gr_i} \qquad (x = 1 \ldots n) \tag{2.3}$$

Die Ranking–Selektion legt also weniger Wert auf die absolute Qualität eines Individuums, sondern auf die relative Position der Qualität im Vergleich mit den anderen Individuen.

	Individuum Nummer (x)	Qualität q_x	Rang r	Selektions-wert s	$P_s(ind_x)$
	1	73	1	2	20 %
	2	57	0	1	10 %
	3	123	3	4	40 %
	4	107	2	3	30 %
$\sum$	–	360	–	10	100 %

Abbildung 2.10: Auswahlwahrscheinlichkeiten bei Ranking–Selektion mit Gradient $g = 1$

Kappa–Ka Rauschbehandlung

Bei vielen realen Optimierungsproblemen hat die Qualitätsfunktion keine 'glatte' Oberfläche, wie sie die Abbildung 2.11 links zeigt. Sie ist vielmehr mit 'Rauschen' durchsetzt, d. h. der direkte Qualitätsvergleich zwischen zwei sehr naheliegenden Punkten ist nicht aussagekräftig, er wird von den kleinen Erhebungen des Rauschens verfälscht. Die Funktion in Abblidung 2.11 rechts ist dabei nicht als statisch anzusehen, sondern vielmehr als eine 'Momentaufnahme'. Durch Ungenauigkeiten in der Meßmethode erhält man bei mehrmaligem Messen desselben Individuums leicht unterschiedliche Werte. Paßt sich die Schrittweite der Evolutionsstrategie in den Bereich der kleinen 'Rauschhügel' an, so verliert sie sich im Rauschen und kann das Optimum nicht mehr finden.

Für dieses Problem schafft die sogenannte „Kappa–Ka Rauschbehandlung" Abhilfe. Sie muß dazu an zwei Stellen im Algorithmus der Evolutionsstrategie eingreifen: in der Mutation und in der Selektion.

Die Grundidee läßt sich dabei folgendermaßen umschreiben: „groß mutieren, klein vererben". Dies bedeutet, daß die Mutation mit einem vielfachen der normalen Schrittweite durchgeführt

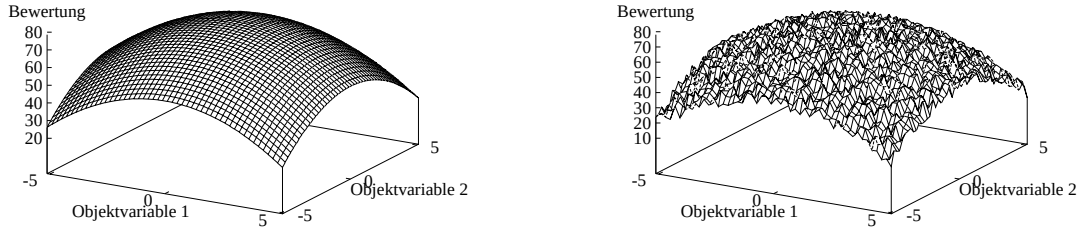


Abbildung 2.11: Funktion ohne und mit Rauschen

wird. Der Radius der Hyperkugel (die Mutationsschrittweite), auf deren Oberfläche die mutierten Individuen plaziert werden, wird durch den Faktor k vergrößert. Außerdem wird der Mechanismus der Mutationsschrittweitenregelung durch den Exponenten κ verstärkt durchgeführt. Dadurch gelangen die Mutanten außerhalb des Wirkungsbereiches der kleinen Rauschschwankungen der Qualitätsfunktion. Diese Individuen werden dann bewertet. Eine große Mutation verringert allerdings die Wahrscheinlichkeit, bessere Individuen hervorzubringen. Deshalb folgt der zweite Ansatzpunkt: ist ein Individuum selektiert worden, so wird die Verstärkung wieder rückgängig gemacht. Dazu muß aber noch das zugehörige Elternindividuum bekannt sein. Von diesem aus wird dann aus den Objektvariablen der Verstärkungsfaktor k herausgenommen und die exponentielle Schrittweitenverstärkung wird ebenso wieder rückgängig gemacht.

Obwohl die Verstärkung wieder rückgängig gemacht wird, hat sie den folgenden Effekt: die Qualität des Individuums wird an einem Ort gemessen, der viel weiter in Richtung des Mutationsschrittes liegt, als das Individuum selbst. Dadurch wird der lokale Störbereich des Rauschens verlassen. Die effektive Schrittweite des Individuums wird dabei durch den Exponenten κ verändert, der bei MSR die Mutationsschrittweiten variiert.

2.4.6 Probleme von Evolutionsstrategien

Der Einsatz von Evolutionsstrategien bei einem Optimierungsvorgang setzt nur sehr wenige Eigenschaften des Problems voraus. Dadurch ist eine universelle Einsetzbarkeit gegeben. Trotzdem gibt es einige Aspekte von denen der Erfolg einer Evolutionsstrategie entscheidend abhängt oder bei denen sich Probleme auftun.

Die Mutationsschrittweite ist einer der entscheidenden Faktoren, der die Fortschrittsgeschwindigkeit der Strategie bestimmt. Die Fortschrittsgeschwindigkeit läßt sich auf verschiedene Weisen definieren: als Qualitätsgewinn oder als Annäherung im Problemraum, jeweils bezogen auf eine Generation. Bei einer großen Mutationsschrittweite ist das Ziel in weniger Schritten zu erreichen. Allerdings wird mit zunehmender Schrittweite die Wahrscheinlichkeit kleiner, daß ein Kindindividuum erzeugt wird, das besser ist, als der Elter. Je kleiner die Schrittweite ist, desto größer wird die Erfolgswahrscheinlichkeit, die Fortschrittsgeschwindigkeit verringert sich aber. Es muß also ein Kompromiß zwischen Erfolgswahrscheinlichkeit und Fortschrittsgeschwindigkeit gefunden werden. Der Bereich, in dem die Schrittweite beides gewährleistet,

wird „Evolutionsfenster“ genannt. Eine Lösung zur Anpassung der Schrittweite in das Evolutionsfenster ist die Mutationsschrittweitenregelung.

Die grundsätzliche Vorgehensweise von Evolutionsstrategien ist es, dem lokalen Anstieg der Qualität zu folgen. Deshalb fallen sie in die Klasse der Gradientenaufstiegsverfahren. Dies bringt es aber mit sich, daß sie ‘kurzsichtig’ das nächstgelegene, evtl. nur lokale Optimum finden. Dieses kann dann meist nicht mehr mit einer Großmutation verlassen werden, da die Mutationsschrittweitenregelung die Schrittweite bereits in einen sehr kleinen Bereich angepaßt hat. Dies ist nötig, damit die Individuen sich dem Optimum immer feiner annähern können. Eine Lösung für dieses Problem ist z. B. der Mechanismus des Individuenaustausches beim Insel–Modell, das im nächsten Kapitel besprochen wird, oder der Einsatz von geschachtelten Strategien. Bei geschachtelten Strategien wird auf der Population eine Großmutation durchgeführt, wodurch der Einflußbereich des lokalen Optimums verlassen werden kann.

Der Einsatz von Evolutionsstrategien bei Strukturierungsproblemen bringt eine weitere Schwierigkeit mit sich. Wenn z. B. zur Optimierung einer Trag– oder Verbindungsstruktur Verbindungsstrecken hinzugefügt oder weggenommen werden können, so muß ein Weg gefunden werden, dies in den Objektvariablen angemessen zu codieren. Solche Probleme, bei denen nicht immer Kontinuität gegeben ist, weil das Vorhandensein oder Nichtvorhandensein einer Verbindung diskrete Zustände sind, bereiten den Evolutionsstrategien Probleme. [Rechenberg 94] schlägt hier wiederum den Einsatz von geschachtelten Strategien vor. Dabei sollen dann die diskreten Größen nur auf Populationsebene mutiert werden. In der eingeschachtelten Strategie wird weiterhin nur an den kontinuierlichen Größen optimiert.

Die grundsätzlich positive Idee, einmal erreichte gute Lösungen erst zu verwerfen, wenn bessere Lösungen gefunden wurden, erweist sich in vielen Fällen als nachteilhaft. Diese Vorgehensweise ist bei den Strategien vom Typ *Plus* anzutreffen. Dabei kann es vorkommen, daß sich ein Individuum schon sehr nahe am Optimum befindet, sich aber nicht weiter nähern kann, da es eine ungünstig große Schrittweite besitzt. Wird es in Richtung des Optimums mutiert, so wird dieses überschritten und das Kindindividuum befindet sich auf der wieder abfallenden Strecke jenseits des Optimums und wird somit nicht selektiert werden. Das Individuum ist dadurch ‘unsterblich’, kann aber keinen weiteren Fortschritt bringen. Die Eigenschaft des ‘Vergessens’ guter Lösungen ist also durchaus erwünschenswert. Diesem Problem schaffen Strategien des Typs *Komma* Abhilfe.

Das Vergessen guter Lösungen kann natürlich auch nachteilhaft sein, z. B. wenn die Lösung schon das Optimum gefunden hat, oder als einzige den Einflußbereich eines lokalen Optimums verlassen hat. Ein guter Kompromiß zwischen Vergessen und der Unsterblichkeit von Individuen wäre vielleicht die Einführung einer maximalen Lebensdauer für Individuen. Dieser Aspekt wird in Standard–Strategien bisher nicht modelliert.

Bei einigen der genannten Probleme schaffen die im nächsten Kapitel vorgestellten Konzepte von parallelen Evolutionsstrategien eine Verbesserung.

Kapitel 3

Parallelisierung von Evolutionsstrategien

In diesem Kapitel wird eine Einführung in Parallelrechner–Architekturen gegeben und auf die konkrete Architektur der Intel Paragon eingegangen. Danach werden Ansatzpunkte der Parallelisierung von Evolutionären Algorithmen auf zwei verschiedenen Rechnerarchitekturen gezeigt.

3.1 Arten von Parallelrechnern

Für Rechnerarchitekturen existieren verschiedene Klassifikationen. Eine sehr bekannte Klassifikation, die Rechner in 4 Klassen einteilt, ist die Klassifikation nach Flynn [Flynn 66]. Dabei werden die folgenden beiden Kriterien unterschieden:

- ein oder mehrere Befehlsströme (Prozessoren)
- ein oder mehrere Datenströme

Daraus ergeben sich die folgenden Abkürzungen für die Rechnerklassen:

- **SISD**: *S*ingle *I*nstruction stream, *S*ingle *D*ata stream.
  (klassisches System mit einer zentralen CPU, von-Neumann–Rechner)
- **SIMD**: *S*ingle *I*nstruction stream, *M*ultiple *D*ata streams.
  (Feld– oder Vektorrechner, Massiv parallele Systeme)
- **MISD**: *M*ultiple *I*nstruction streams, *S*ingle *D*ata stream.
  (Pipelinerechner)
- **MIMD**: *M*ultiple *I*nstruction streams, *M*ultiple *D*ata stream.
  (Multiprozessorsysteme/Multicomputer mit mehreren unabhängigen Prozessoren)

Auf SISD– und MISD–Architekturen soll hier nicht näher eingegangen werden. Interessant sind die beiden Rechnerarchitekturen SIMD und MIMD, auf denen verschiedene Implementierungen von Evolutionsalgorithmen existieren.

3.1.1 SIMD–Parallelrechner

Ein Parallelrechner der Klasse SIMD (Schematische Darstellung s. Abbildung 3.1) besitzt einen zentralen Steuerprozessor und sehr viele Datenprozessoren (*PE* – *p*rocessing element). Der Steuerprozessor hat einen eigenen lokalen Speicher für Programme und Daten. Er liest das Programm ein und führt die Befehlsdecodierung durch. Befehle, die auf einem einzelnen Datum arbeiten (*skalares* Datum), führt er direkt aus. Skalare Daten werden in seinem lokalen Speicher gehalten. Die anderen Befehle gibt er über Steuerleitungen an die PEs weiter.

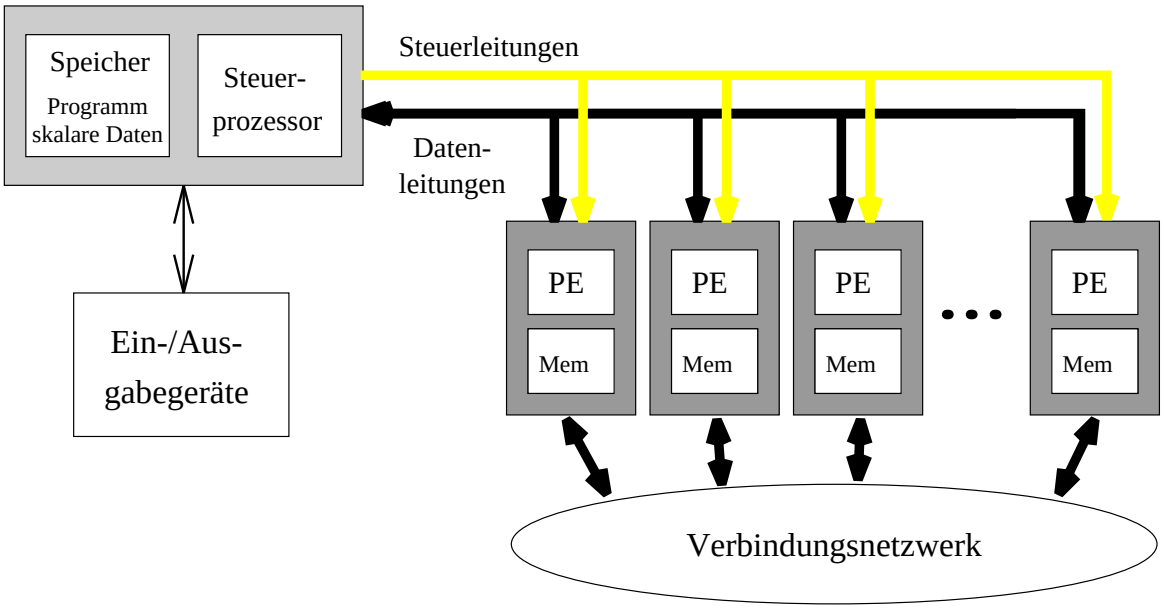


Abbildung 3.1: Schematischer Aufbau eines SIMD–Parallelrechners

Jedes PE führt dabei synchron mit allen anderen denselben Befehl aus, jeweils aber mit seinen lokalen Daten. Dabei besteht die Möglichkeit, daß ein PE bei einem Befehl untätig bleibt. Auf diese Weise kann eine bedingte Ausführung erreicht werden. Es ist jedoch nicht möglich, daß verschiedene PEs zum selben Zeitpunkt verschiedene Befehle ausführen. Jedes PE besitzt einen — meist sehr kleinen — lokalen Speicher, in dem sich ausschließlich Daten befinden (vektorielle Daten). Durch die große Zahl an PEs summiert sich der vorhandene lokale Speicher jedoch zu einer beträchtlichen Größe auf. Die PEs sind dabei in einer bestimmten Topologie angeordnet, z. B. linear oder in einem Feld (Array). Weiterhin existiert ein Verbindungsnetzwerk, über das synchron sehr schnell Daten zwischen den PEs ausgetauscht werden können.

SIMD–Rechner eignen sich wegen ihrer großen Zahl von PEs sehr gut zur Bildverarbeitung, zur Simulation Neuronaler Netze und zur Vektor– und Matrixrechnung, wobei dann ein PE jeweils für die Berechnung eines Bildpunktes, eines Neurons oder eines Vektor- bzw. Matrixelementes zuständig ist. Für solche Probleme, die sich gut auf alle PEs parallelisieren lassen, bringen diese Rechner einen sehr großen Speedup der Rechengeschwindigkeit im Verhältnis zu SISD–Rechnern. Wegen ihrer großen Anzahl an PEs werden diese Rechner oft auch als *massiv parallel* bezeichnet. Es handelt sich dabei um sogenannte *feinkörnige* oder *synchrone* Parallelität.

Ein Beispiel für einen Array–Rechner ist die *MasPar MP-1*. Sie besitzt 16384 PEs, die in einem zweidimensionalen Feld von 128×128 angeordnet sind. Es existieren zwei verschiedene

Verbindungsnetzwerke, das X–Net und der Router. Das X–Net verbindet jedes PE mit seinen 8 Nachbarn (4 horizontal und vertikal, 4 diagonal). Der Router ist frei konfigurierbar und erlaubt es, allgemeinere Verbindungstopologien einzustellen. Auf der MasPar wurde neben MPGA auch das Programm MPES implementiert, mit welchem in Kapitel 5 Vergleichsmessungen angestellt werden.

Weitere Beispiele für Rechner der SIMD–Klasse sind der *Distributed Array Processor DAP-610* und der Neurocomputer *Adaptive Solutions CNAPS*, für den im Rahmen des EA–Gruppe eine Implementierung von Genetischen Algorithmen erstellt wurde (siehe 1.3 und [Ebner 95]).

3.1.2 MIMD–Parallelrechner

Ein Rechner der Klasse MIMD ist in der Lage, mehrere verschiedene Programme auszuführen, wobei jedes wiederum auf eigenen, lokalen Daten arbeitet. Damit er mehrere Befehlsstöme verarbeiten kann, muß jeder Prozessor — im Gegensatz zu einem PE beim SIMD–Modell — eine eigene Decodiereinheit für Befehle besitzen. Die Prozessoren eines MIMD–Rechners sind wesentlich komplexer und damit auch teurer als die PEs eines SIMD–Rechners. Aus diesem Grund besitzt er auch deutlich weniger davon. Normalerweise werden in MIMD–Rechnern handelsübliche Mikroprozessoren verwendet, wie sie auch in Einzelplatzrechnern Verwendung finden. Weiterhin sind die Prozessoren über ein Kommunikationsnetzwerk untereinander verbunden. Dieses wird aber im Gegensatz zu SIMD–Rechnern nicht zum synchronen Datenaustausch benutzt, sondern für asynchron versendete Nachrichten.

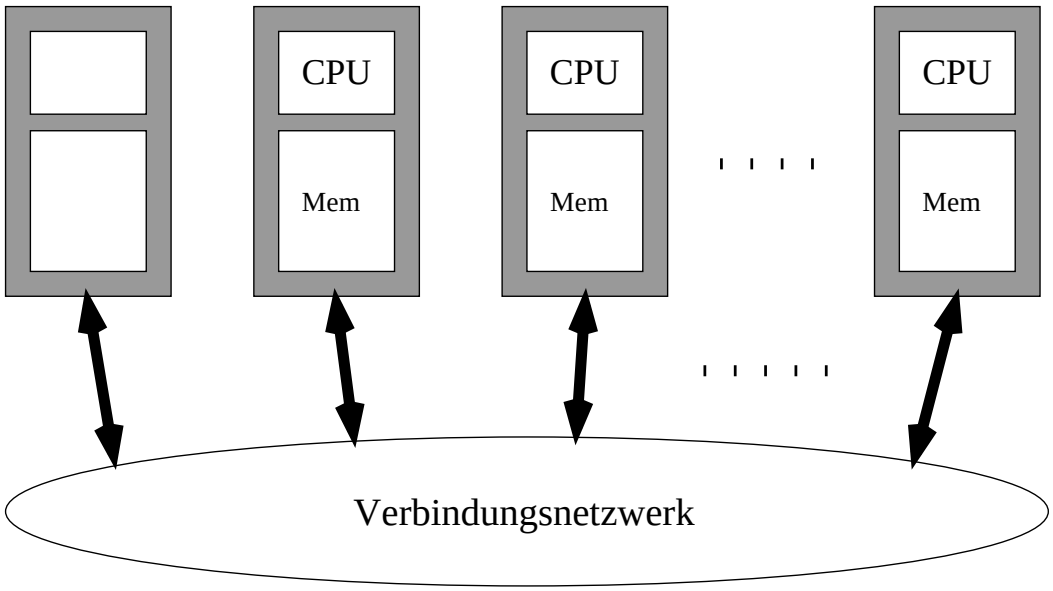


Abbildung 3.2: MIMD-Multicomputer–Modell

Abhängig von der Anordung des Speichers läßt sich die MIMD–Klasse noch in zwei Unterklassen aufteilen: *Multiprozessorsysteme* und *Multicomputer*. Bei den Multiprozessorsystemen besitzt ein Prozessor nur wenig oder gar keinen eigenen Speicher. Dafür kann über das Verbindungsnetzwerk auf *gemeinsamen Speicher (shared memory)* zugegriffen werden. Über

diesen kann auch die Kommunikation abgewickelt werden. Der gemeinsame Speicher stellt eine Ressource dar, auf die der gemeinsame Zugriff durch Synchronisationsmechanismen, wie z. B. *Semaphore*, geregelt werden muß. Dabei kann es zu Ressourcenkonflikten kommen, die zu einer *Verklemmung (Deadlock)* führen. Im Fall eines Deadlocks warten Prozessoren zyklisch auf die Freigabe einer Ressource, die von einem anderen Prozessor belegt ist. Somit kann keiner der Prozessoren weiterarbeiten.

In Abbildung 3.2 ist das Schema eines *Multicomputers* dargestellt. Zur Verdeutlichung der Unterschiede zum SIMD–Modell wurde hier der Prozessor mit *CPU (central processing unit)* bezeichnet und der Speicher vergößert dargestellt. Bei einem Multicomputer existiert im Gegensatz zum Multiprozessorsystem kein gemeinsamer Speicher. Jeder Prozessor besitzt seinen eigenen, lokalen Speicher, auf den nur er zugreifen kann. Die Kommunikation und der Datenaustausch kann hier ausschließlich durch Nachrichten über das Verbindungsnetzwerk geschehen. Auch hier kann es zu Verklemmungen kommen, wenn Prozessoren zyklisch auf Nachrichten warten.

Um die Parallelität eines MIMD–Rechners möglichst gut auszunutzen, sollten die Aufgaben so verteilt werden, daß jeder Prozessor immer arbeitet und nicht untätig warten muß. Algorithmen, die dies gewährleisten sollen und dazu verschiedene Strategien verfolgen, sind die *Lastbalancierung (Load Balacing)* oder das *Scheduling*.

Weil ein MIMD–Rechner nur eine verhältnismäßig kleine Anzahl an Prozessoren besitzt, und die Berechnung in relativ großen, dafür aber unabhängigen Teilen auf die Prozessoren abgebildet wird, spricht man hier von *grobkörniger* oder *asynchroner* Parallelität. Beispiele für Rechner dieser Klasse sind die *Sequent Symmetry* (Multiprozessorsystem) und die Rechner *Connection–Machine CM-5*, *Cray T3D*, *IBM SP1* und *SP2*, *BBN Butterfly*, *nCube* und *Intel Paragon* (alles Multicomputer). Die Intel Paragon wird im nächsten Abschnitt detailliert beschrieben.

3.1.3 Die Intel Paragon

Die Intel Paragon ist nach der im vorigen Abschnitt erläuterten Klassifikation als MIMD–Multicomputer einzustufen. Die genaue Typbezeichnung lautet *Intel Paragon XP/S-5*. Der Rechner wird gemeinsam vom IPVR[1] und RUS[2] betrieben.

Hardware

Die Paragon besteht aus insgesamt 82 Rechenknoten, die je nach Aufgabe verschiedenen Partitionen angehören. Sie sind wie folgt aufgeteilt: Der größte Teil davon, 72 Stück, können exklusiv von Anwendungen belegt werden und stellen diesen ihre gesamte Rechenleistung zur Verfügung (*Compute–Partition*). 5 Knoten gehören der *Service–Partition* an, auf der Benutzer–Shells und Dienstprogramme laufen. Weiterhin sind 3 Knoten für I/O (Harddisks) und 2 Knoten für die HiPPI[3]–Netzanbindung zuständig. In Abbildung 3.3 ist der Aufbau der Paragon schematisch zu sehen, wie ihn das Tool *SPV*[4] darstellt.

[1] *I*nstitut für *P*arallele und *V*erteilte *H*öchstleistungsrechner
[2] *R*echenzentrum der *U*niversität *S*tuttgart
[3] *Hi*gh *P*erformance *P*arallel *I*nterface
[4] *S*ystem *P*erformance *V*isualization Tool

Die Rechenknoten sind in einem 2-dimensionalen Gitter angeordnet. Ein Knoten kann mit 16–192 MByte Speicher bestückt werden, auf den zwei Prozessoren vom Typ *intel i860XP* gemeinsam zugreifen. Der eine Prozessor dient zur Bereitstellung von Rechenleistung (Applikationsprozessor), der andere wird als Kommunikationsprozessor eingesetzt. Desweiteren sind auf einem Knoten ein Netzwerkinterface und zwei Transferengines vorhanden, die den Zugang zu den 4 Kommunikationskanälen ermöglichen. Diese verbinden einen Knoten bidirektional mit seinen 2 horizontalen und vertikalen Nachbarknoten und haben eine Übertragungsleistung von je 200MByte pro Sekunde.

Durch die zweidimensionale Gitterarchitektur kann die Paragon sehr leicht um weitere Knoten erweitert werden. Dies kann sogar in sehr kleinen Schritten geschehen. Nachteilig ist jedoch, daß die Distanz zwischen zwei Knoten relativ groß werden kann, wodurch sich die Latenzzeit beim Versenden von Nachrichten erhöht (in Tabelle 3.4 mit *Kommunikationsstartup* bezeichnet). Nachrichten werden zwischen zwei beliebigen Knoten über die dazwischenliegenden Knoten geleitet. Die Wege der Nachrichten werden von iMRC Routing–Chips nach bestimmten Routing–Strategien (Wormhole–Routing) festgelegt. Durch diese leistungsfähigen Komponenten wird eine erhöhte Latenzzeit zum größten Teil wieder ausgeglichen.

In Tabelle 3.4 ist eine Übersicht der Hardware und ihrer Leistungsdaten gegeben (Quelle: Internet www–Page http://www.uni-stuttgart.de/pcg/intel/hardware.html).

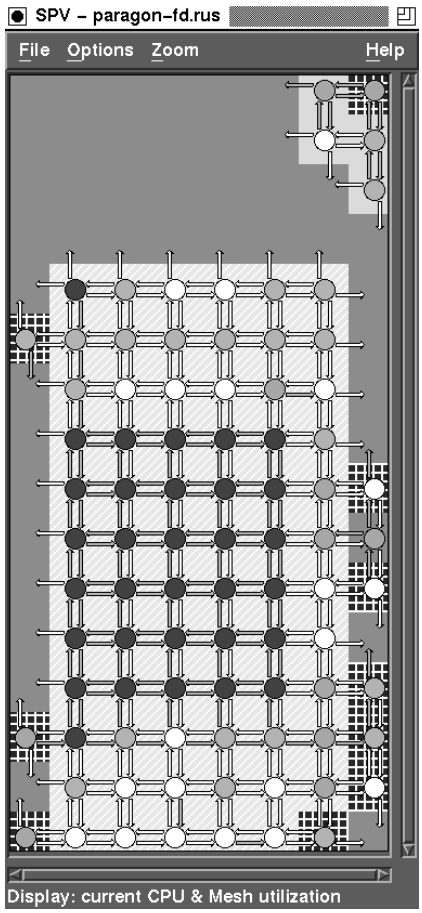


Abbildung 3.3: Aufbau der Intel Paragon

Software–Aspekte

Als Betriebssystem wird auf der Paragon OSF/1, ein verteiltes UNIX auf Microkernel–Basis verwendet. Sie wird nicht im Front–End/Back–End Modus betrieben, wie die meisten anderen Parallelrechner (z. B. die MasPar). In diesem Modus laufen auf dem Parallelrechner selbst (Back–End) nur die speziellen parallelen Programme, die er von einem anderen Rechner erhält (Front–End). Auf dem Front–End laufen die Benutzerprozesse und ein Cross–Compiler, der die Parallelprogramme erzeugt.

Im Gegensatz dazu führt die Paragon auch die Benutzerprozesse aus und compiliert Parallelprogramme. Dafür werden eigene Knoten, die sogenannte *Service–Partition*, zur Verfügung gestellt, auf der die Prozesse dem üblichen Zeitscheiben–Scheduling unterworfen werden. Diese Partition verhält sich also wie eine normale Mehrprozessor–UNIX-Workstation. In der Compute–Partition dagegen können Rechenknoten exklusiv angefordert werden, welche der Applikation dann mit ihrer vollen Rechenleistung zur Verfügung stehen. Allerdings werden, wenn nötig, Nachrichten anderer Programme durch diese Partition geleitet, wodurch jedoch

Hardware–Übersicht Intel Paragon	
Knotenzahl	72 Compute, 5 Service, 3 I/O, 2 HiPPI
CPU	i860XP, 50 MHz, 75 MFLOPS (64 Bit), 50 MIPS
System Peak Performance	5,4 GFLOPS
Memory auf Einzelknoten	32 MB, 64 MB auf 4 Service–Knoten
Memory des Gesamtsystems	2,3 GB
Kommunikationsbandbreite der Einzelverbindung	200 MBytes/s
Kommunikationsbandbreite des Gesamtsystems	105 GBytes/s
Kommunikationsstartup	25 μs
I/O–Bandbreite	12,5 MBytes/s lokal
Local Disk Space	19,2 GB
Topologie	2-D Mesh
Programmiermodell	Message–Passing, HPF
Betriebssystem	OSF/1
Netzeinbindung	Ethernet, HiPPI, FDDI

Abbildung 3.4: Tabellarische Hardwareübersicht der Intel Paragon

nur die Kommunikationsleistung beeinträchtigt werden kann. Um eine gerechte Verteilung der Rechenknoten zu gewährleisten, werden die Knoten dem Benutzer nach einer gewissen Zeitspanne wieder entzogen. Je weniger Knoten angefordert werden, um so länger darf darauf gerechnet werden.

In der Zeit von 20.30 Uhr bis 7.30 Uhr ist die Paragon auf Batch–Betrieb umgeschaltet. In dieser Zeit stehen nur noch 6 Knoten der Compute–Partition für interaktiven Betrieb zur Verfügung, auf den restlichen 66 werden Batch–Jobs gefahren. Batch–Jobs bieten den Vorteil, daß auch größere Knotenzahlen für längere Zeit als im interaktiven Betrieb benutzt werden dürfen. Sollten während der Zeit des interaktiven Betriebes einmal sämtliche Knoten der Compute–Partition frei werden, so wird automatisch auf Batch–Betrieb umgeschaltet. Sobald dann wieder interaktive Jobs gestartet werden, führt dies unmittelbar zur Suspendierung der Batch–Jobs.

Als Programmiermodell bietet die Paragon die folgenden drei Alternativen an:

- Message Passing
- Data–/Worksharing (HPF)
- Shared Virtual Memory

Bei Message Passing können dabei wiederum drei verschieden Mechanismen benutzt werden:

- Die *nx*–Bibliothek. Dies ist die Hardware–naheste Möglichkeit.
- Die *MPI*–Bibliothek (*M*essage *P*assing *I*nterface).

- Die *PVM*–Bibliothek. PVM bedeutet *P*arallel *V*irtual *M*achine und implementiert ein allgemeines Nachrichtenmodell, das auch für sehr viele andere Rechnerarchitekturen verfügbar ist. PVM ist Hardware–unabhängig und ist besonders zum Einsatz in heterogenen Netzen geeignet. Auf der Paragon ist PVM auf *nx* aufgesetzt.

Als Programmiersprachen sind auf der Paragon C und Fortran verfügbar. Für beide Sprachen stehen die oben genannten Message–Passing Bibliotheken zur Verfügung. Der C–Compiler *icc* hält sich sehr nahe an den ANSI-C Standard.

3.2 Ansatzpunkte der Parallelisierung

Nachdem nun die Klassifikation von Parallelrechnern in die beiden Hauptgruppen SIMD und MIMD vorgestellt und die konkrete MIMD–Architektur der Intel Paragon gezeigt wurde, soll nun auf die Abbildung von Evolutionsstrategien auf die Parallelarchitekturen eingegangen werden.

Die Natur selbst ist ein hochgradig paralleles System. Da die Evolutionsstrategien nach dem Vorbild der Natur arbeiten, wohnt auch ihnen eine hohe Parallelität inne. Deshalb liegt es nahe, zur Geschwindigkeitssteigerung und effizienteren Durchführung Parallelrechner einzusetzen. Verschiedene Teile einer Evolutionsstrategie besitzen allerdings unterschiedliche Grade an Parallelität. Die Berechnung der Qualität eines Individuums kann normalerweise völlig unabhängig von den anderen Individuen durchgeführt werden. Genauso geht die Mutation von einem einzelnen Individuum aus und ist somit unabhängig anwendbar. Da die Rekombination auf zwei oder mehr Individuen zugreifen muß, ist hier keine absolute Unabhängigkeit vorhanden. Die Selektion schließlich muß Zugriff auf alle Individuen der Population haben, wodurch die Parallelität weiter eingeschränkt wird.

3.2.1 Parallelisierung auf SIMD–Architekturen

Auf einem Rechner vom SIMD–Typ bietet es sich wegen der großen Zahl an Prozessoren und der synchronen Parallelität an, auf Ebene der Individuen zu parallelisieren. Dabei befindet sich im lokalen Speicher jedes PEs ein Eltern– und ein Kind–Individuum. Es kann dann gleichzeitig für alle Individuen die Qualität berechnet und die Mutation angewandt werden. Bei der Rekombination ist es mit dieser Abbildung allerdings nicht sinnvoll, beliebige Rekombinationspartner zuzulassen. Dafür wäre es nötig, Verbindungen zu beliebigen anderen PEs herzustellen, und das für alle PEs gleichzeitig. Dies würde sich in der Praxis nur durch sequentielles Holen der Rekombinationspartner erreichen lassen. Deshalb kann jedes PE die Individuen für die Rekombination nur aus einer festgelegten Menge von Nachbar–PEs holen, z. B. nur aus den direkten Nachbarn. Abbildung 3.5 stellt die Rekombination dar und wird wie folgt interpretiert: die grauen Individuen sind die in Frage kommenden Rekombinationspartner, von denen die erforderliche Anzahl ausgewählt wird. Daraus entsteht das rekombinierte Individuum (schwarz).

Bei der Selektion ergeben sich bei diesem Modell zwei Probleme: Zum einen ist es, wie bei der Rekombination, nicht sinnvoll, global aus allen Individuen zu selektieren, da dies zu viel Kommunikationsaufwand erfordern würde. Zum anderen besitzen Evolutionsstrategien die für

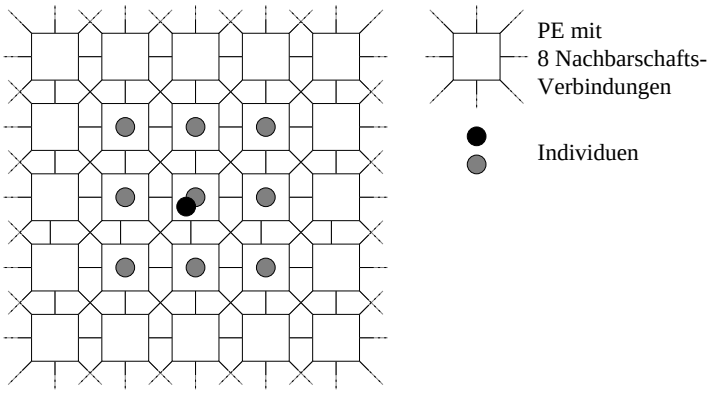


Abbildung 3.5: Lokale Rekombination und Selektion auf den direkten Nachbarindividuen bei Evolutionsstrategien auf SIMD–Rechnern

dieses Modell ungünstige Eigenschaft, daß sich die Populationsgröße von μ auf λ und wieder zurück ändert. Paßt man einen der beiden Parameter auf die Zahl der PEs an, so bleiben entweder manche PEs untätig, oder manche PEs müssen zwei Individuen bearbeiten und andere nur eines. Zur Lösung dieses Problems ändert man den Algorithmus der Evolutionsstrategie ab. Die Populationsgröße wird beibehalten und jedes PE selektiert die Individuen nur aus der Menge der lokalen Nachbar–PEs. Für jedes PE sieht es dabei trotzdem so aus, als würde aus einer größeren Menge eine kleinere Menge selektiert, wodurch wiederum ein Selektionsdruck hergestellt wird. Bei diesem Modell eröffnen sich dadurch neue Perspektiven der Ausbreitung guter Individuen. Je nachdem, wie weit die lokale Nachbarschaft definiert ist, kann sich ein gutes Individuum nicht zu schnell über die gesamte Population vermehren. Dadurch wird es möglich, daß auch schlechtere Individuen, die sich vielleicht in der Nähe des globalen Optimums befinden, nicht sofort ausselektiert werden. Es verringert sich also die Chance durch kurzsichtigen Opportunismus das globale Optimum zu verfehlen.

Auch Plus– und Komma–Strategien lassen sich hierbei realisieren. Im Falle von Komma wird die Selektion nur innerhalb der Kindindividuen gemacht. Bei Plus wird zu der Selektionsmenge eines bestimmten PEs noch das auf diesem PE vorhandene Elternindividuum hinzugenommen. Dazu läßt sich Abbildung 3.5 wie folgt interpretieren: Das schwarze Individuum ist ein Elter, die grauen sind die durch Mutation entstandenen Kinder. Bei Komma wird von den grauen Individuen eines ausgewählt, welches das schwarze ersetzt. Bei Plus wird das schwarze nur dann ersetzt, wenn das neue Individuum eine bessere Qualität hat.

Normale My-Lambda–Evolutionsstrategien haben die Eigenschaft, daß sich schon nach kurzer Anlaufzeit die Individuen auf einen nah zusammenliegenden Bereich gruppieren. Dies liegt daran, daß sich jedes Individuum im Selektionskampf gegen jedes andere Individuum der gesamten Population bewähren muß. Dadurch werden schlechte Individuen recht schnell ausselektiert. Deshalb werden je nach Anzahl der Dimensionen auch nur sehr kleine Populationen benötigt. Bei dem oben beschriebenen Modell dagegen wird der Problemraum weiträumiger durchsucht, weil die Individuen durch die räumliche Entfernung mehr Zeit haben, sich zu verbessern.

Evolutionsstrategien auf einer SIMD–Architektur, wie sie in diesem Abschnitt beschrieben wurden, sind z. B. in MPES [Görzig 95] implementiert.

3.2.2 Parallelisierung auf MIMD–Architekturen

Auf einem MIMD–Rechner lohnt es sich nicht, auf jeden Prozessor nur ein einzelnes Individuum zu legen. Dies liegt zum einen daran, daß im allgemeinen nicht so viele Prozessoren wie bei einer SIMD–Architektur zur Verfügung stehen. Zum anderen dauert die Kommunikation mit anderen Prozessoren viel länger als die Berechnungszeit für ein einzelnes Individuum. Dadurch würde also keine große Effizienz erzielt werden.

Wegen der grobkörnigen Parallelität bietet sich hier ein anderer Ansatz an: Jeder Prozessor hält eine komplette lokale Population auf welcher er eine Evolutionsstrategie ausgeführt. Je nachdem, was außer der isolierten Berechnung einer Strategie getan wird, gibt es hier zwei verschiedene Modelle der Abbildung: das Insel–Modell und die Abbildung von geschachtelten My-Lambda–Strategien auf die Prozessoren.

Das Insel–Modell

Die Verteilung von mehreren Subpopulationen auf die Prozessoren und deren getrennte Bearbeitung entspricht in dieser simplen Form der mehrfachen Ausführung einer normalen My-Lambda–Evolutionsstrategie. Dabei muß sinnvollerweise jede Population mit anderen Startwerten beginnen. Dies wird in Analogie zur realen Welt *Insel–Modell* genannt, da die Populationen voneinander getrennt sind.

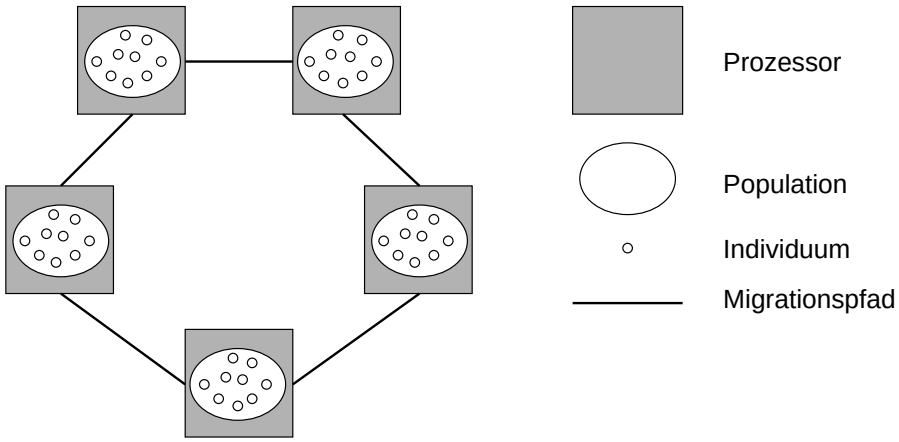


Abbildung 3.6: Insel–Modell mit ringförmiger Verbindungstopologie

Aber auch in der Natur ist eine völlige Isolierung meist nicht gegeben. Es können durchaus einige Individuen von einer Population zur anderen überwechseln. Dadurch findet ein Austausch von Gen–Material statt, der neue Informationen in eine Population einbringen kann. Der Austausch wird auch *Migration* genannt, entsprechend heißen die Austauschpfade *Migrationspfade*. Zwischen welchen Populationen Migrationspfade existieren und in welche Richtung sie verlaufen, legt die Verbindungs–*Topologie* fest. Es müssen also nicht alle Populationen untereinander verbunden sein und die Verbindungen müssen auch nicht bidirektional beschaffen sein. In Abbildung 3.6 ist das Prinzip der Teilpopulationen mit der Verbindungstopologie dargestellt.

Da beim Insel–Modell eine Generation einer Evolutionsstrategie komplett lokal auf einem Prozessor abläuft, kommen als Rekombinationspartner nur die Individuen der lokalen Population in Frage. Auch die Selektion wählt die Individuen der nächsten Generation nur aus der Subpopulation aus.

Der Individuenaustausch kann wahlweise synchron oder asynchron stattfinden. Beim synchronen Austausch müssen sich nach der Berechnung einer Generation alle Prozessoren synchronisieren, erst dann wird der Austausch vollzogen. Vollzieht sich der Austausch dagegen asynchron, so können quasi zu jedem Zeitpunkt der Berechnung Individuen versendet oder empfangen werden. Dabei werden die evtl. anfallenden Wartezeiten bei der Synchronisation eingespart.

Für den Individuenaustausch müssen dabei zusätzlich zu den normalen Parametern der Strategie noch folgende Punkte festgelegt werden:

- Wann Individuen versendet werden.
  Beim synchronen Austausch wird dies durch eine Anzahl von Generationen festgelegt.
- Welche Individuen versendet werden.
  Hierfür können Selektionsmethoden eingesetzt werden. Sinnvoll ist es aber auch, einfach nur die besten Individuen zu versenden.
- Wieviele Individuen versendet werden.
  Dies kann durch einem absoluten Wert oder einen Prozentsatz (Austauschrate) festgelegt sein und ggf. auch variiert werden.
- Wohin die Individuen versendet werden.
  Dies wird, wie oben schon erläutert, über die Verbindungs–Topologie bestimmt.
- Welche Individuen der vorhanden Population durch die neuen Individuen ersetzt werden (Ersetzungsstrategie).

Geschachtelte My-Lambda–Evolutionstrategien

Die in Abschnitt 2.4.4 erläuterten geschachtelten Evolutionsstrategien lassen sich ähnlich wie beim Insel–Modell auf die Prozessoren abbilden. Hier noch einmal die allgemeine Notation dieses Strategietyps:

$$[\mu'/\rho'^{+},\ \lambda'(\mu/\rho^{+},\ \lambda)^{\gamma}]^{\gamma'}$$

Dabei führen λ' Kindpopulationen jeweils eine einfache Strategie vom Typ $(\mu/\rho^{+},\ \lambda)^{\gamma}$ für den Zeitraum von γ Generationen isoliert durch. Deshalb bietet es sich an, λ' auf die Anzahl der zur Verfügung stehenden Prozessoren festzulegen. Ist der Strategietyp auf Populationsebene Komma, so muß die Bedingung $\mu' \leq \lambda'$ erfüllt sein und somit muß ein Prozessor höchstens jeweils eine Eltern– und eine Kindpopulation speichern. Bei einer Plus–Strategie darf μ' theoretisch größer werden als λ'. Dies kann entweder künstlich auf $\mu' \leq \lambda'$ eingeschränkt werden, oder es müssen auf einem Prozessor evtl. mehr als nur eine Population gehalten werden.

Zu Beginn werden von den μ' Elternpopulationen jeweils ρ' rekombiniert und auf einem Prozessor als noch unmutierte Kindpopulation abgelegt. Hierzu muß natürlich ein Individuenaustausch von den zu rekombinierenden Populationen zum Zielprozessor stattfinden.

Nun liegen λ' Kindpopulationen vor. Falls keine Rekombination stattgefunden hat, so werden diese einfach durch Duplizierung einer zufällig gewählten Elternpopulation gewonnen. Die Kindpopulationen werden nun noch mutiert. Dies geschieht, wie in Abschnitt 2.4.4 erläutert durch Verschieben der gesamten Population im Problemraum. Nach der Mutation kann die isolierte Berechnung der eingeschachtelten My-Lambda–Strategie erfolgen.

Sind die γ Generationen dieser Strategie durchgerechnet worden, so folgt nun die Selektion auf Populationsebene. Hierbei müssen diejenigen Populationen selektiert werden, die die Elternpopulationen des nächsten Zyklus bilden sollen. Dazu können die bekannten Selektionsverfahren verwendet werden. Allerdings arbeiten sie hier nicht auf den Qualitäten von Individuen, sondern auf den Populationsqualitäten. Nach der Selektion kann ein neuer Zyklus der geschachtelten Evolutionsstrategie beginnen.

Nun soll noch auf die Unterschiede und Gemeinsamkeiten der erläuterten Abbildung von geschachtelten Evolutionsstrategien und der Abbildung nach dem Insel–Modell eingegangen werden.

Zunächst die Gemeinsamkeiten:

- Auf jedem Prozessor wird genau eine Population nach dem My-Lambda–Schema bearbeitet.
- Für eine gewisse Zahl an Generationen werden die Kindpopulationen isoliert berechnet.
- Es findet ein Austausch von Individuen zwischen den Prozessoren statt.

Unterschiede zwischen den beiden Modellen gibt es in folgenden Punkten:

- Beim Insel–Modell wird auf jedem Prozessor nur die „Arbeitspopulation“ gehalten.
  Bei den geschachtelten Stratgien muß zusätzlich noch eine oder sogar mehrere Elternpopulationen gehalten werden.
- Bei geschachtelten Strategien kommt es vor, daß ganze Populationen verworfen werden.
  Beim Insel–Modell bleiben die Populationen von Anfang bis Ende bestehen. Große Veränderungen kann es nur durch neu hinzukommende Individuen geben.
- Beim Insel–Modell wird nur ein kleiner Anteil Individuen versendet.
  Bei den geschachtelten Strategien werden dagegen bei der Rekombination größere Teile der Population versendet. Bei der Duplikation muß sogar eine gesamte Population bewegt werden.
- Beim Insel–Modell kann der Individuenaustausch über verschiedene Topologien erfolgen.
  Bei den geschachtelten Strategien besteht auf Populationsebene grundsätzlich zwischen allen Prozessoren eine Verbindung.
- Bei geschachtelten Strategien werden Kindpopulationen mutiert. Dies ist beim Insel–Modell nicht vorgesehen.

Kapitel 4

Das Programm VEES

In diesem Kapitel wird detailliert auf das während dieser Diplomarbeit entstandene Programm VEES eingegangen. Zuerst werden die konkreten, parallelen Evolutionsstrategien vorgestellt, wie sie in VEES implementiert sind. Es wird der zentrale Datentyp *genus* erläutert, welcher zur Verwaltung aller Teilpopulationen dient, die zur Ausführung einer einfachen My-Lambda–Strategie notwendig sind. Im Anschluß daran werden die auf diesem Datentyp operierenden Formeln der evolutionsstrategischen Methoden aufgezeigt.

Es folgen Abschnitte über die Compilierung und Bedienung des Programmes. Darin wird erläutert, wie man aus den Quelldateien das ausführbare Programm erhält, und wie es im interaktiven und Batch–Modus betrieben wird. Anschließend werden die wichtigsten Aspekte bei der Implementierung genannt, die Funktionsweise der Einprozessorversion erklärt und eine Anleitung zur Erweiterung von VEES um neue Applikationsfunktionen gegeben.

4.1 Parallele Evolutionsstrategien in VEES

Wie in Abschnitt 3.2.2 gezeigt, gibt es verschiedene Möglichkeiten, Evolutionsstrategien auf parallele MIMD–Computerarchitekturen abzubilden. In VEES wurden zwei Arten von Strategien implementiert: die *VeES–Strategien*, welche nach Art des Insel–Modells operieren, und die *geschachtelten My-Lambda–Strategien*. Die Implementierung dieser beiden Strategiearten wird in den nächsten Abschnitten erklärt.

4.1.1 VeES–Strategien

Die Evolutionsstrategien vom Typ *VeES* (*V*erteilte *E*volutions–Strategien) sind nach dem Insel–Modell mit synchronem Individuenaustausch implementiert.

VEES ist allerdings, abweichend vom Insel–Modell, nach dem *Master-Slave–Prinzip* aufgebaut. Das bedeutet, daß einer der Rechenknoten, der *Master*, die Steuerung der Strategie übernimmt. Alle anderen Knoten sind die sogenannten *Slaves*, welche den Kern der Evolutionsstrategie ausführen (s. Abb. 4.1). Auf ihnen sind die Populationen lokalisiert, deren Individuen dupliziert, rekombiniert, mutiert und selektiert werden. Die Slaveknoten sind in ständiger Empfangsbereitschaft und warten auf Befehlsnachrichten vom Master. Dieser bestimmt, ob z. B.

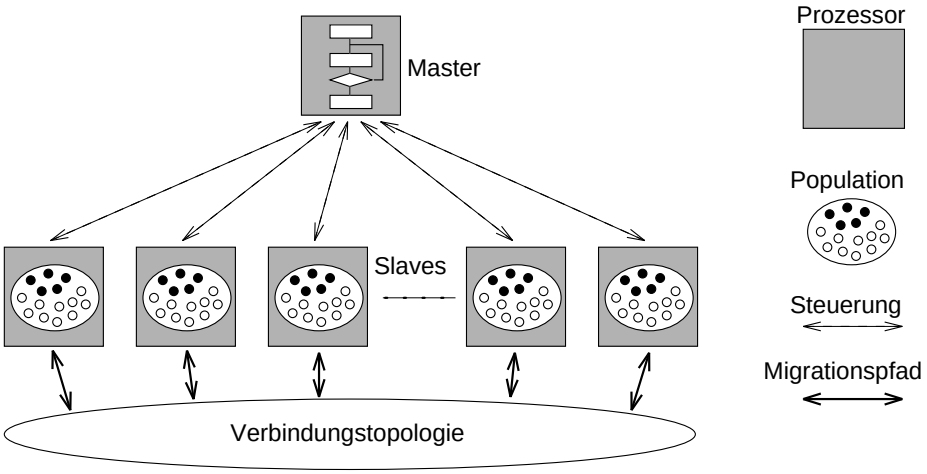


Abbildung 4.1: Schema einer Strategie vom Typ VeES

weitere Generationen berechnet werden oder ob ein Individuenaustausch stattfinden soll. Die Architektur von VEES ist in Abschnitt 4.5.3 genauer erklärt.

Abbildung 3.3 auf Seite 29 zeigt die Knoten der Intel Paragon, während VEES mit der Durchführung einer Evolutionsstrategie beschäftigt ist. Die dunklen Kreise stellen rechnende Knoten dar, die hellen (weiß oder hellgrau) sind gerade unbenutzt. Wie man gut erkennen kann, läuft VEES auf den 5×6 Knoten in der Mitte. Auf dem einzelnen Knoten, der sich unterhalb des 5×6 Rechteckes ganz links befindet, läuft der Master. Die Knoten des Rechteckes sind die Slave–Knoten. Ganz oben links ist ein weiterer dunkler Kreis zu sehen. Auf diesem Knoten lief zu der Zeit ein Programm eines anderen Benutzers.

VeES–Strategien lassen sich mit der Notation für My-Lambda–Strategien beschreiben:

$$(\mu/\rho\overset{+}{,}\lambda)^{\gamma}$$

Allerdings muß sie hier anders interpretiert werden: auf jedem der Slaveknoten wird eine My-Lambda–Strategie ausgeführt. Die Gesamtzahl an Eltern– und Kindindividuen ergibt sich also durch Multiplikation von μ bzw. λ mit der Anzahl der Slaveknoten.

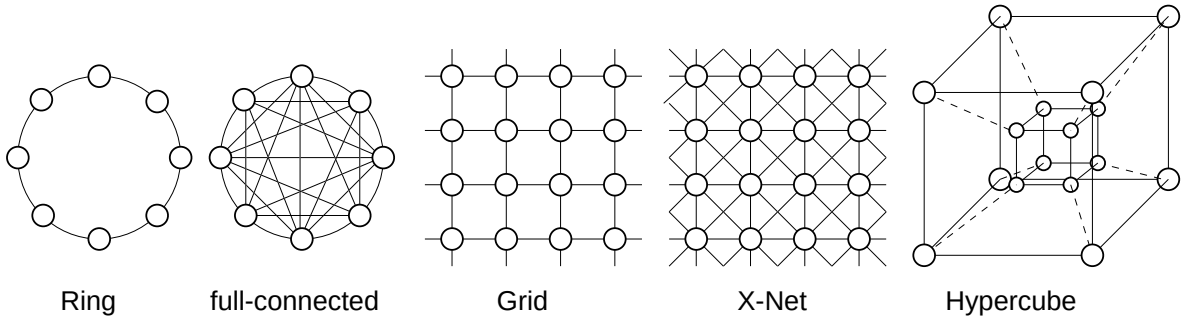


Abbildung 4.2: Für eine VeES–Strategie verfügbare Austauschtopologien

Jeweils nach einer gewissen Anzahl an Generationen findet der Individuenaustausch über eine festgelegte Verbindungstopologie statt. Die in VEES verfügbaren Verbindungstopologien sind in Abbildung 4.2 dargestellt. Verbindungen sind dabei grundsätzlich bidirektional.

Eine wichtige Größe beim Individuenaustausch ist die Austauschrate. Sie wird in Prozent angegeben und bezieht sich auf die Anzahl μ der Elternindividuen. Dies kommt daher, daß der Austausch zwischen der Berechnung zweier Generationen stattfindet. Zu diesem Zeitpunkt liegen bereits die μ Eltern der nächsten Generation vor. Die λ Kindindividuen sind nicht mehr verfügbar, da sie nur zeitweilig, während der Berechnung einer Generation existieren.

Die Parameter μ, ρ, λ und γ allein sind jedoch nicht ausreichend, um alle Abläufe bei einer VeES–Strategie festzulegen. Die dazu benötigten Größen sind in den folgenden Tabellen aufgelistet und erklärt.

Parameter	**Beschreibung**
μ	Zahl der Elternindividuen pro Knoten
λ	Zahl der Kindindividuen pro Knoten
ρ	Zahl der zu rekombinierenden Individuen
γ	Zahl der zu berechnenden Generationen
Plus/Komma	Selektionstyp der Strategie
r_{exch}	Austauschrate in Prozent bezogen auf μ
i_{exch}	Austauschintervall in Generationen
t_{exch}	Austauschtopologie

Ist $\rho = 1$ so findet keine Rekombination statt. Bei $\rho > 1$ wird Rekombination mit ρ Individuen durchgeführt. Rekombiniert werden in erster Linie die Objektvariablen. Aber auch die Strategievariablen, welche eine einzelne *(uniform)* oder mehrere *(separate)* Schrittweiten sein können, werden rekombiniert. Die Rekombinationsart kann für die Objekt– und die Strategievariablen getrennt bestimmt werden. Es stehen die folgenden Möglichkeiten zur Verfügung:

Rekombinations-Methode	**Beschreibung**
dominant	Für jede Variable wird entschieden, von welchem Individuum sie übernommen wird
intermediär	Die resultierende Variable ergibt sich als Durchschnitt der Variablen der Rekombinanten
kontinuisiert	Verallgemeinerung der dominanten Rekombination; die Objektvariablen kommen auf einer Hyperkugel um den Rekombinantenschwerpunkt zu liegen; diese Methode ist nicht für Strategievariablen (Schrittweiten) möglich

Weiterhin können verschiedene Selektionsmethoden verwendet werden, welche hier mit ihren Parametern aufgelistet sind:

Selektionsmethode	**Beschreibung**
best	Bestenwahl
roulette–wheel	Proportionale Auswahl von Individuen
ranking	Lineare Rangauswahl
g	Ranking Gradient

Auch die Mutation kann durch zahlreiche Größen beeinflußt werden. Es handelt sich dabei um die folgenden Parameter:

Parameter	Beschreibung
δ	Anfangswert der Mutationsschrittweite für jedes Individuum
uniform/separate	Art der Mutationsschrittweite δ. Bei *uniform* wird eine einheitliche Schrittweite für das ganze Individuum verwendet, bei *separate* existiert für jede Objektvariable eine eigene Schrittweite δ_i
MSR	Mutationsschrittweitenregelung an oder aus
α	Änderungsfaktor der Mutationsschrittweite bei MSR
noise	Rauschunterdrückung an oder aus
κ und k	Verstärkungsfaktoren für die Mutation bei Rauschunterdrückung

4.1.2 My-Lambda–Strategien

Echte verteilte My-Lambda–Strategien sind in VEES nicht implementiert. Der Grund dafür ist, daß sie sich nicht effizient auf die MIMD–Architektur umsetzen lassen. Die Population müßte auf mehrere Prozessoren aufgeteilt werden. Es wäre dann sehr viel Kommunikation nötig, da die evolutionsstrategischen Methoden auf Individuen der gesamten Population zugreifen. Der Kommunikationsaufwand würde den potentiellen Geschwindigkeitsgewinn zunichte machen. Ein weiterer Grund, der gegen eine parallele My-Lambda–Strategie spricht, ist, daß VeES-Strategien und geschachtelte My-Lambda–Strategien leistungsfähiger sind. Sie erfordern weniger Kommunikation und die Wahrscheinlichkeit, in ein lokales Optimum zu geraten, ist ebenfalls geringer.

Es ist jedoch trotzdem möglich, eine reine My-Lambda–Strategie in VEES auszuführen. Wird VEES auf einer Workstation oder auf nur einem Knoten der Paragon gestartet, so wird erkannt, daß nur ein Rechenprozessor zur Verfügung steht. Durch die Master-Slave–Architektur wird ebenfalls nur ein Rechenprozessor verwendet, wenn VEES auf zwei Knoten der Paragon läuft, da ein Knoten für den Master verwendet wird. Da auf jedem Rechenprozessor nur eine Population bearbeitet wird, stellt dies eine einfache My-Lambda–Strategie dar. Die Kommunikationsparameter sind in diesen Fällen unwirksam und erscheinen auch nicht im Menü.

4.1.3 Geschachtelte My-Lambda–Strategien

Geschachtelte Evolutionsstrategien der Form $[\mu'/\rho'^{+}, \lambda'(\mu/\rho^{+}, \lambda)^{\gamma}]^{\gamma'}$ sind, wie in Abschnitt 3.2.2 erläutert, auf die MIMD–Parallelarchitektur der Intel Paragon abgebildet. Strategien mit höheren Schachtelungstiefen als 2 sind in VEES nicht implementiert.

Auch bei den geschachtelten My-Lambda–Strategien läuft auf einem Prozessor der Master, der die Slaves steuert. Jeder Slaveknoten führt die eingeschachtelte My-Lambda–Strategie der Form $(\mu/\rho^{+}, \lambda)^{\gamma}$ für γ Generationen isoliert durch. Die Zahl λ' der Kindpopulationen kann nicht verändert werden und ist gleich der Zahl der zur Verfügung stehenden Slaveknoten. Auf jedem Slaveknoten befindet sich zusätzlich zu der Kindpopulation höchstens eine Elternpopulation. Deshalb ist auch bei Strategien vom Typ Plus auf Populationsebene die Einschränkung $\mu' \leq \lambda'$ gegeben. Dies wäre theoretisch nicht nötig, vereinfacht aber einerseits die Implementierung und begrenzt andererseits den Kommunikationsaufwand bei Populations–Duplikation und Rekombination.

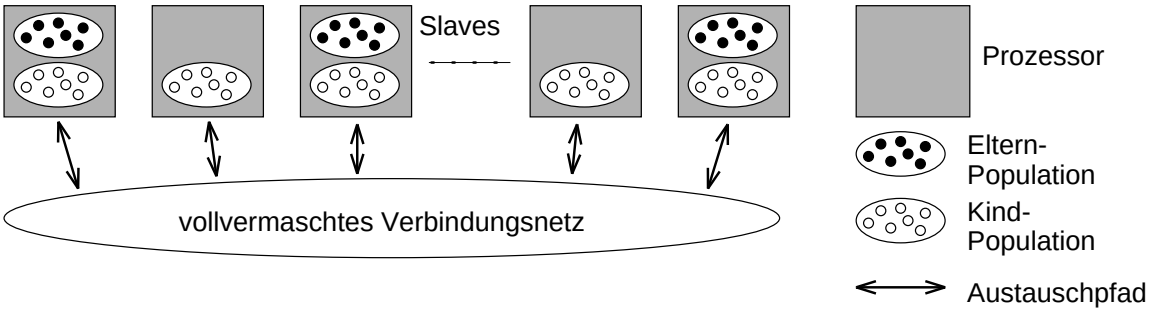


Abbildung 4.3: Abbildung einer geschachtelten My-Lambda–Strategie auf die Prozessoren (der Masterknoten und die Koordination ist hier weggelassen)

In den nächsten Absätzen wird der genaue Ablauf der geschachtelten My-Lambda–Strategien auf dem Master und den Slaves erläutert. Der Master signalisiert zunächst μ' der λ' Slaves, sich mit einer Elternpopulation zu initialisieren. Daraufhin folgt sofort die Duplikation oder Rekombination.

Bei der Duplikation legt der Master fest, welche der Elternpopulationen dupliziert werden. Hierbei wird dann eine Optimierung angewendet, um unnötiges Versenden ganzer Populationen zu vermeiden: Soll eine bestimmte Population dupliziert werden, so wird die Kopie auf demselben Prozessor abgelegt. Soll die Population mehrfach dupliziert werden, so müssen alle weiteren Kopien versendet werden.

Auch bei der Rekombination bestimmt der Masterknoten für jeden Slaveknoten die Rekombinationspopulationen. Dabei bekommt jeder Knoten sowohl die Information, auf welchen Knoten seine Rekombinationspopulationen sind, als auch, für welche Knoten die eigene Elternpopulation Rekombinant ist. Damit kann dann jeder Knoten die gewünschten Individuen anfordern, welche anschließend ausgetauscht werden.

Die Mutation der duplizierten oder rekombinierten Populationen geht sehr einfach: Die Population wird komplett um einen Zufallsvektor der Länge δ' verschoben. δ' ist dabei die Mutationsschrittweite der Population. Die Verschiebung der kompletten Population geschieht durch Verschieben jedes einzelnen Individuums.

Nachdem nun auf jedem Slaveknoten eine Kindpopulation vorliegt, bekommen sie vom Master das Signal, γ Generationen der eingeschachtelten My-Lambda–Strategie zu berechnen. Wenn sie fertig sind, fordert der Master die Populationsqualitäten an und entscheidet, welche Populationen selektiert werden. Wieder wird zur Vermeidung von unnötiger Kommunikation die selektierte Population auf dem Knoten belassen, auf dem sie sich befindet. Dadurch sind die Elternpopulationen der nächsten Generation unregelmäßig auf die Rechenknoten verteilt. Dies ist jedoch unproblematisch, da der Master diese Informationen hat und damit die weiteren Schritte koordinieren kann.

Nach der Selektion schließt sich die Schleife zur Duplikation oder Rekombination. Dies wird dann γ' Zyklen lang wiederholt. Insgesamt werden also γ mal γ' Generationen berechnet.

Für die geschachtelten Evolutionsstrategien können grundsätzlich dieselben Parametereinstel-

lungen wie bei VeES–Strategien gemacht werden. Eine Ausnahme hiervon bilden nur die drei Austauschparameter r_{exch}, i_{exch} und t_{exch}. Sie sind hier nicht verfügbar. Dafür gibt es eine Reihe von Parametern, die die evolutionsstrategischen Methoden auf Populationsebene bestimmen:

Parameter	Beschreibung
μ'	Zahl der Elternpopulationen
λ'	Zahl der Kindpopulationen (ist auf die Zahl der Slaveknoten festgelegt)
ρ'	Zahl der zu rekombinierenden Populationen
γ'	Zahl der zu berechnenden Zyklen auf Populationsebene
Plus'/Komma'	Selektionstyp für Populationen
δ'	Anfangs–Mutationsschrittweite für Populationen
MSR'	Mutationsschrittweitenregelung auf Populationsebene an oder aus
α'	Änderungsfaktor der Mutationsschrittweite δ' bei MSR'
dominant'	Rekombinationsmethode für Populationen; jedes Individuum wird per gleichverteilter Zufallswahl aus den ρ' Populationen ermittelt
best-average'	Selektionsmethode für Populationen; es wird die Population mit der besten durchschnittlichen Qualität gewählt

4.2 Formale Darstellung von Evolutionären Mechanismen

In diesem Abschnitt werden die konkreten Formeln gezeigt, mit denen die evolutionsstrategischen Methoden auf den Populationen und auf den Individuen operieren. Dazu ist allerdings die Kenntnis des Datentyps *genus* nötig, in dem die Populationen realisiert sind. Deshalb folgt zunächst die Vorstellung dieses Datentyps.

4.2.1 Der Datentyp genus

Wie in den vorigen Abschnitten 4.1.1 und 4.1.3 gezeigt wurde, läuft sowohl bei VeES–Strategien, als auch bei geschachtelten My-Lambda–Strategien auf jedem Prozessor für sich gesehen, jeweils eine Strategie vom Typ My-Lambda ab. Genaugenommen läuft jeweils eine Generation nach dem Schema $(\mu/\rho^{+}, \lambda)$ ab. Die Abläufe zwischen der Berechnung von einzelnen Generationen hängen dann vom jeweiligen Strategietyp ab. Bei der VeES–Strategie werden evtl. Individuen ausgetauscht. Bei der geschachtelten My-Lambda–Strategie wird auf der Populationsebene Selektion, Duplikation, Rekombination und Mutation durchgeführt.

In VEES ist der Haupt–Datentyp, auf dem die Evolutionsstrategien durchgeführt werden der Typ *genus*. Er verwaltet die verschiedenen Teilpopulationen, die intern nötig sind, um eine Generation nach dem My-Lambda–Schema durchzuführen. In Abbildung 4.4 sind diese Teilpopulationen dargestellt. Außerdem ist zu sehen, wie die Individuen kopiert und von welchen evolutionsstrategischen Methoden sie dabei modifiziert werden.

Ausgangsbasis sind die μ Anfangsindividuen der μ–Population (P^{μ}). Die Individuen selbst sind die μ–Individuen (I^{μ}). Die Duplikation kopiert sich die zufällig ausgewählten ρ Rekombinanten in die ρ–Population (P^{ρ}). Diese Population kann nur die ρ Individuen aufnehmen, die zur Bildung eines einzigen rekombinierten Individuums nötig sind. Das resultierende Individuum

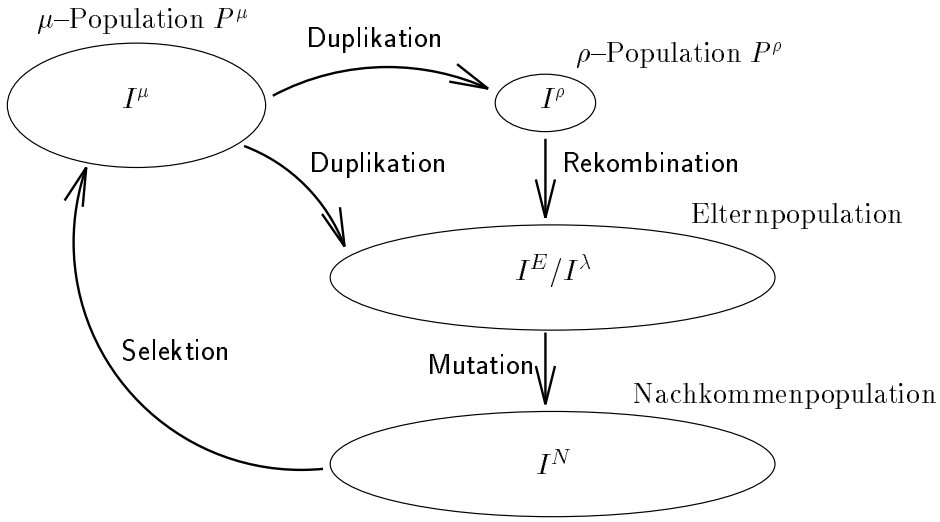


Abbildung 4.4: Schematische Darstellung der Populationen des Typs *genus*

ist noch unmutiert und heißt deshalb *Elter*–Individuum (I^E oder I^λ). Ohne Rekombination kopiert die Duplikation direkt aus der μ–Population in die Eltern–Population.

Die Elternindividuen werden mutiert und die resultierenden Individuen in der Nachkommen–Population (P^N) abgelegt. Diese umfaßt ebenso wie die Eltern–Population λ Individuen. Die unmutierten Elternindividuen müssen wegen der Kappa-Ka–Rauschbehandlung noch abrufbar sein.

Die Selektion kopiert schließlich gewisse Individuen aus der Menge der Nachkommenpopulation in die μ–Population der nächsten Generation.

4.2.2 Formeln

Die Operationen auf den Populationen des Datentyps *genus* werden in den nächsten Abschnitten formal beschrieben. Die Formeln sind größtenteils dieselben, wie sie auch in MPES implementiert wurden. Sie wurden in Zusammenarbeit mit Steffen Görzig ausgearbeitet.

Duplikation

Die bestimmenden Parameter der Duplikation sind $\mu, \rho, \lambda, \mu', \rho'$ und λ'. Zur Duplikation von Individuen und Populationen werden die Formeln 4.1 angewendet. Die Duplikation von Populationen findet natürlich nur bei geschachtelten My-Lambda–Evolutionsstrategien Anwendung.

$$I_i^\lambda = I_{z_i}^\mu, \quad P_j^{\lambda'} = P_{r_j}^{\mu'} \tag{4.1}$$
$$i = 1, \ldots, \lambda \quad j = 1, \ldots, \lambda'$$
$$z_i = [\,1, \mu\,]\text{–gleichverteilte Zufallszahl}$$
$$r_j = [\,1, \mu'\,]\text{–gleichverteilte Zufallszahl}$$

Wie in Abbildung 4.4 zu sehen ist, kopiert die Duplikation bei Verwendung von Rekombination nicht in die Eltern–Population, sondern jeweils ρ Individuen für eine Rekombinationsvorgang in die ρ–Population (Formel 4.2). Soll Populationsrekombination stattfinden, so werden in VEES keine Populationen dupliziert, stattdessen werden erst bei der Rekombination die gewünschten Individuen von den Prozessoren angefordert. Die Auswahl der zu rekombinierenden Populationen funktioniert nach dem in Formel 4.2 angegebenen Prinzip.

$$
\begin{aligned}
I_i^{\rho} = I_{z_i}^{\mu}, \quad P_j^{\rho'} = P_{r_j}^{\mu'} \\
i = 1, \ldots, \rho \quad j = 1, \ldots, \rho' \\
z_i = [\,1, \mu\,]\text{–gleichverteilte Zufallszahl} \\
r_j = [\,1, \mu'\,]\text{–gleichverteilte Zufallszahl}
\end{aligned}
\tag{4.2}
$$

Rekombination

Die Rekombination bildet aus den ρ Individuen der ρ–Population ein neues Individuum und legt es in der Eltern–Population ab. Für Individuen sind folgende Rekombinationsarten implementiert:

- dominant
- intermediate
- continuous

Die Rekombination betrifft sowohl die Objektvariablen o als auch Strategievariablen δ bzw. $\vec{\delta}$. Außerdem wird noch das Alter und die Qualität rekombiniert. Ist jede dieser Variablen gemeint, so werden sie unter a zusammengefaßt.

Bei der dominanten Rekombination wird jede Variable des λ–Individuums nur durch einen Rekombinanten bestimmt. Durch eine Zufallswahl wird für jede Variable entschieden, von welchem Individuum sie kopiert wird:

$$
a_{\lambda} = a_{\rho z} \quad , \quad z = [\,1, \rho\,]\text{–gleichverteilte Zufallszahl} \tag{4.3}
$$

Bei der intermediären Rekombination werden zur Bildung einer neuen Variablen jeweils die entsprechenden Variablen der ρ Rekombinanten gemittelt:

$$
a_{\lambda} = \frac{\sum_{i=1}^{\rho} a_{\rho i}}{\rho} \tag{4.4}
$$

Die kontinuisierte Rekombination ist eine erweiterte Form der dominanten Rekombination und betrifft nur die Objektvariablen o. Wie in Abschnitt 2.4.5 beschrieben, wird das λ–Individuum dabei auf einer Hyperkugel um den Schwerpunkt S der ρ–Individuen gesetzt. Der Schwerpunkt S wird bestimmt durch:

$$
o_i^S = \frac{\sum_{j=1}^{\rho} o_i^{\rho j}}{\rho} \quad , \quad i = 1, \ldots, n \tag{4.5}
$$

Analog der Mutation wird das λ–Individuum auf die Schale der Hyperkugel mit dem Radius r um S gesetzt. Dazu wird wieder der auf die Länge 1 normierte Verschiebungsvektor $\vec{v}$ gebildet (bei zwei Dimensionen durch Formel 4.6, sonst durch Formel 4.7).

$$v_1 = \cos(2\pi z_i), \quad v_2 = \sin(2\pi z_i) \qquad z_i = [0,1)\text{–gleichverteilte Zufallszahl}, \quad i = 1,2 \tag{4.6}$$

$$v_i = N(0, \frac{1}{\sqrt{n}})\text{–verteilte Zufallszahl}, \quad i = 1, \ldots, n \tag{4.7}$$

Der normierte Verschiebungsvektor muß nun noch mit dem Radius r multipliziert werden, um der Hyperkugel die richtige Größe zu geben. Sind nur zwei Eltern vorhanden ($\rho = 2$), so berechnet sich r nach der Formel:

$$r = \frac{1}{2}\sqrt{\sum_{i=1}^{n}(o_i^{\rho 1} - o_i^{\rho 2})^2} \tag{4.8}$$

Für $\rho > 2$ läßt sich eine Hyperkugel, auf der sich alle ρ–Individuen befinden, nur sehr selten bestimmen. Deshalb wird r zwischen den Abständen der ρ–Individuen R zum Schwerpunkt S interpoliert. Für den Abstand A gilt Formel 4.9, der Radius r berechnet sich nach Formel 4.10.

$$A_i = \overline{SR_i} = \sqrt{\sum_{j=1}^{n}(o_j^S - o_j^{\rho i})^2} \quad , \quad i = 1, \ldots, \rho \tag{4.9}$$

$$r = \frac{\sum_{i=1}^{\rho} A_i}{\rho} \tag{4.10}$$

Nachdem nun die Hyperkugel bestimmt wurde, kann das λ–Individuum auf die Schale dieser Kugel gesetzt werden:

$$o_i^\lambda = o_i^S + r\,v_i \quad , \quad i = 1, \ldots, n \tag{4.11}$$

Individuen besitzen neben Objekt– und Strategievariablen auch ein Alter. Dieses läßt sich schlecht rekombinieren. Hier wird einfach zwischen allen ρ–Individuen gemittelt:

$$a_\lambda = \frac{\sum_{i=1}^{\rho} a_{\rho i}}{\rho} \tag{4.12}$$

Geschachtelte Evolutionsstrategien besitzen auch eine Rekombination auf *Populationsebene*. Die Objekte der Rekombination sind nun ganze Individuen, nicht mehr ihre Objektvariablen. Für Populationen kann nur die dominante Rekombination verwendet werden. Populationen werden allerdings aus Effizienzgründen nicht zuerst dupliziert. Stattdessen führt jeder Prozessor für sich die Zufallswahl aus, welche Individuen aus welcher der ρ' Populationen in die rekombinierte Population gelangen. Diese werden dann von dem entsprechenden Prozessor angefordert.

Wie in Abschnitt 2.4.4 erläutert, unterscheidet sich die dominante Rekombination bei Populationen von derjenigen bei Individuen. Jedes Individuum hat unabhängig von seiner Position innerhalb der Population die gleiche Chance, in die rekombinierte Population zu gelangen. Ein Individuum darf dabei nicht mehrfach ausgewählt werden. Um dieses Prinzip zu implementieren, gibt es verschiedene Möglichkeiten. Diese sind aber algorithmischer Natur und lassen sich nicht kompakt als Formel darstellen. Deshalb sei an dieser Stelle nur die Formel für die Auswahlwahrscheinlichkeit P_r eines Individuums genannt, in die rekombinierte Population zu gelangen:

$$P_r(I^{\rho'}) = \frac{1}{\rho' \cdot \mu} \tag{4.13}$$

Mutation

Die λ–Individuen bilden die Eltern des Mutationsvorganges auf Individuenebene. Wie in Abschnitt 2.4.5 erläutert, wird ein Nachkommenindividuum auf einen zufällig gewählten Punkt einer Hyperkugel gelegt. Mittelpunkt der Hyperkugel ist das Elternindividuum, der Radius ist die Mutationsschrittweite δ.

Als erster Schritt der Mutation muß die Mutationsschrittweite δ_N, bzw. $\vec{\delta_N}$ des Nachkommens bestimmt werden. Die Mutationsschrittweite selbst kann eine skalare Größe, oder bei separaten Schrittweiten ein Vektor sein. Der Schrittweiten–Vektor hat dann genausoviele Elemente (n) wie das Individuum Objektvariablen besitzt. Ohne Mutationsschrittweitenregelung (MSR) wird einfach die Schrittweite des Elters kopiert. Im Falle einer einzelnen Schrittweite wird Formel 4.14 angewendet, bei separaten Schrittweiten Formel 4.15.

$$\delta_{Ni} = \delta_{Ei} \tag{4.14}$$

$$\vec{\delta_{Ni}} = \vec{\delta_{Ei}} \tag{4.15}$$

$$i = 1 \ldots \lambda$$

Bei Anwendung der MSR wird die Schrittweite des Nachkommens mit einer Wahrscheinlichkeit von jeweils 1/3 vergrößert, gleich belassen oder verkleinert. Die Größe der Änderung wird durch den Modifikationsfaktor α festgelegt.

$$\delta_{Ni} = \xi_i \cdot \delta_{Ei} \tag{4.16}$$

$$\xi_i = \begin{cases} \alpha & : \quad 0 \leq z_i < \frac{1}{3} \\ 1 & : \quad \frac{1}{3} \leq z_i < \frac{2}{3} \\ \frac{1}{\alpha} & : \quad \frac{2}{3} \leq z_i < 1 \end{cases}$$

$$z_i = [\,0,1)\text{–gleichverteilte Zufallszahl}, \quad i = 1, \ldots, \lambda$$

Bei der Verwendung von separaten Schrittweiten wird nicht der gesamte Schrittweitenvektor vergößert oder verkleinert, sondern jede Komponente j einzeln:

$$\delta_{Ni}^j = \xi_i^j \cdot \delta_{Ei}^j \tag{4.17}$$

$$\xi_i^j = \begin{cases} \alpha & : \quad 0 \leq z_i^j < \frac{1}{3} \\ 1 & : \quad \frac{1}{3} \leq z_i^j < \frac{2}{3} \\ \frac{1}{\alpha} & : \quad \frac{2}{3} \leq z_i^j < 1 \end{cases}$$

$$z_i^j = [\,0,1)\text{–gleichverteilte Zufallszahl}, \quad i = 1, \ldots, \lambda \quad j = 1, \ldots, n$$

Nachdem nun der Betrag der Mutationsschrittweite (einheitlich oder für jede Objektvariable getrennt) feststeht, muß noch die Richtung bestimmt werden, in die mutiert wird. Diese wird ergibt sich aus einem Zufallsvektor $\vec{v}$. Um ihn von der Anzahl der Objektvariablen $\sharp(o) = n$ unabhängig zu machen, wird seine Länge $|\vec{v}|$ auf den Betrag 1 normiert. Im zweidimensionalen Fall wird aus der Hyperkugel ein Kreis, dessen Radius leicht über Sinus/Kosinus–Terme auf 1 gesetzt werden kann:

$$
\begin{aligned}
(4.18) \qquad & v_1 = \cos(2\pi z_i), \quad v_2 = \sin(2\pi z_i) \\
& z_i = [0,1)\text{–gleichverteilte Zufallszahl}, \quad i = 1,2
\end{aligned}
$$

Für höhere Dimensionen lassen sich diese Formeln nicht mehr anwenden. Hier wird ein anderes Verfahren benutzt, das die Elemente des Verschiebungsvektors aus $N(0,\frac{1}{\sqrt{n}})$–verteilten Zufallszahlen bestimmt (Formel 4.19). Für große n ergibt sich dadurch wiederum für den Vektor $\vec{v}$ ein Betrag von 1.

$$
(4.19) \qquad v_i = N(0,\frac{1}{\sqrt{n}})\text{–verteilte Zufallszahl}, \quad i = 1,\ldots,n
$$

Mit dem Verschiebungsvektor $\vec{v}$ und der Mutationsschrittweite δ_N können dann die Objektvariablen des Elternindividuums o_E zu denen des Nachkommens o_N mutiert werden:

$$
(4.20) \qquad \vec{o}_{Ni} = \vec{o}_{Ei} + \delta_{Ni} \cdot \vec{v}_i \quad , \quad i = 1,\ldots,\lambda
$$

Bei separaten Schrittweiten wird auf jede Objektvariable die zugehörige Schrittweite, multipliziert mit der entsprechenden Komponente von $\vec{v}$, aufaddiert:

$$
(4.21) \qquad o^j_{Ni} = o^j_{Ei} + \delta^j_{Ni} \cdot v_j \quad , \quad i = 1,\ldots,\lambda \quad , \quad j = 1,\ldots,n
$$

Die Mutation von Populationen verschiebt die gesamte Population, und damit jedes Individuum, um einen Zufallsvektor $\vec{v}$. Dabei entsteht eine mutierte Kindpopulation. Es wurden hier die hochgestellten Indizes K und P für die Kind– bzw. Eltern–Population (Parent) verwendet. Wie schon bei den Individuen, wird zuerst die Schrittweite der Kindpopulation bestimmt. Ohne MSR wird sie von der Elternpopulation kopiert (Formel 4.22), mit MSR wird sie evtl. mit dem Modifikationsfaktor α' variiert (Formel 4.23).

$$
\begin{aligned}
(4.22) \qquad & \delta^{Ki}_{Nj} = \delta^{Pi}_{Ej} \\
& i = 1,\ldots,\lambda' \quad , \quad j = 1,\ldots,\mu
\end{aligned}
$$

$$
\begin{aligned}
(4.23) \qquad & \delta^{Ki}_{Nj} = \xi^i_j \cdot \delta^{Pi}_{Ej} \\
& \xi^i_j = \begin{cases} \alpha' & : \quad 0 \le z^i_j < \frac{1}{3} \\ 1 & : \quad \frac{1}{3} \le z^i_j < \frac{2}{3} \\ \frac{1}{\alpha'} & : \quad \frac{2}{3} \le z^i_j < 1 \end{cases} \\
& z^i_j = [0,1)\text{–gleichverteilte Zufallszahl} \\
& i = 1,\ldots,\lambda' \quad , \quad j = 1,\ldots,\mu
\end{aligned}
$$

Der normierte Richtungsvektor $\vec{v}$ zur Bestimmung der Mutationsrichtung wird auf die gleiche Art wie in Formel 4.18 und 4.19 gewonnen. Mit $\vec{v}$ und der Mutationsschrittweite δ_N^K können dann die Objektvariablen aller Individuen der Elternpopulation zu denen der Nachkommenpopulation mutiert werden:

$$\vec{o}_{Nj}^{Ki} = \vec{o}_{Ej}^{Pi} + \delta_{Nj}^{Ki} \cdot \vec{v}_i \qquad i = 1, \ldots, \lambda' \quad , \quad j = 1, \ldots, \mu \tag{4.24}$$

Bewertung

Die Bewertung der λ Nachkommenindividuen wird durch die Qualitätsfunktion f vorgenommen (Formel 4.25). Sie bekommt als Eingabe den Vektor $\vec{o}$ der Objektvariablen. Die bewerteten Nachkommen werden dann der Selektion unterworfen.

$$f(\vec{o}_{Ni}) = q_{Ni} \quad , \quad i = 1, \ldots, \lambda \tag{4.25}$$

Bei geschachtelten Evolutionsstrategien werden auch ganze Populationen bewertet. In VEES ist dabei die Bewertungsfunktion f' fest vorgegeben. Eine einfache Erweiterung durch den Benutzer ist nicht vorgesehen. Die Qualität einer Population ergibt sich als die durchschnittliche Qualität aller Individuen. Da die Bewertung einer Population zwischen der Berechnung von Generationen stattfindet, werden hierzu die μ–Individuen verwendet:

$$f'(P^\mu) = \frac{\sum_{i=1}^{\mu} q_i^\mu}{\mu} \tag{4.26}$$

Selektion

Für die Selektion von Individuen stehen die drei Verfahren *Bestenwahl*, *Roulette-Wheel–Selektion* und *Ranking* zur Verfügung. Die beiden letzten Verfahren wurden schon in Abschnitt 2.4.5 mitsamt den verwendeten Formeln zur Bestimmung der Selektionswahrscheinlichkeit eines Individuums erläutert. Deshalb wird hier auf eine Wiederholung verzichtet.

Die Bestenwahl ist sehr einfach zu realisieren. Zuerst wird die Menge der λ (Typ Komma) bzw. $\mu + \lambda$ (Typ Plus) Individuen sortiert. Die Individuen dieser Menge werden im folgenden mit I^S für Selektionsindividuen bezeichnet. Das Individuum mit der höchsten Qualität hat dabei den Index 1. Aus dieser Menge werden nun die ersten μ Individuen in die μ–Population übernommen:

$$I_i^\mu = I_i^S \qquad i = 1 \ldots \mu \tag{4.27}$$

Bei der Selektion von Populationen existiert nur die Bestenwahl, sie wird auf die gleiche Art gehandhabt:

$$P_i^{\mu'} = P_i^{S'} \qquad i = 1 \ldots \mu' \tag{4.28}$$

Kappa-Ka Rauschbehandlung

Die Kappa-Ka Rauschbehandlung benötigt zusätzlich den Verstärkungsexponenten κ und den Verstärkungsfaktor k. Sie kann nur in Zusammenspiel mit MSR angewendet werden und ist sowohl mit der Mutation, als auch mit der Selektion verschränkt. Während der Mutation werden die Schrittweite und die Objektvariablen verstärkt mutiert. Bei der Selektion wird die Verstärkung wieder herausgenommen.

Bei der Mutation werden für eine einheitliche Schrittweite die Formeln 4.16 und 4.20 zu den Formeln 4.29 und 4.30 ergänzt. Bei separaten Schrittweiten werden entsprechend aus den Formeln 4.17 und 4.21 die Formeln 4.31 und 4.32.

(4.29)
$$\delta_{Ni} = (\xi_i)^\kappa \cdot \delta_{Ei}$$
$$\xi_i = \begin{cases} \alpha & : \quad 0 \le z_i < \frac{1}{3} \\ 1 & : \quad \frac{1}{3} \le z_i < \frac{2}{3} \\ \frac{1}{\alpha} & : \quad \frac{2}{3} \le z_i < 1 \end{cases}$$
$$z_i = [\,0,1)\text{–gleichverteilte Zufallszahl}, \quad i = 1,\ldots,\lambda$$

(4.30)
$$\vec{o}_{Ni} = \vec{o}_{Ei} + \delta_{Ni} \cdot k \cdot \vec{v}_i \quad , \quad i = 1,\ldots,\lambda$$

(4.31)
$$\delta^j_{Ni} = (\xi^j_i)^\kappa \cdot \delta^j_{Ei}$$
$$\xi^j_i = \begin{cases} \alpha & : \quad 0 \le z^j_i < \frac{1}{3} \\ 1 & : \quad \frac{1}{3} \le z^j_i < \frac{2}{3} \\ \frac{1}{\alpha} & : \quad \frac{2}{3} \le z^j_i < 1 \end{cases}$$
$$z^j_i = [\,0,1)\text{–gleichverteilte Zufallszahl}, \quad i = 1,\ldots,\lambda \quad j = 1,\ldots,n$$

(4.32)
$$o^j_{Ni} = o^j_{Ei} + \delta^j_{Ni} \cdot k \cdot v_j \quad , \quad i = 1,\ldots,\lambda \quad , \quad j = 1,\ldots,n$$

Die solchermaßen mutierten Individuen werden nun bewertet und selektiert. Nach der Selektion werden die Verstärkungsfaktoren wieder aus den Individuen herausgerechnet. Um die Faktoren herausrechnen zu können, muß der jeweilige Elter I^E eines Nachkommens I^N noch bekannt sein. Hierfür wird die Elternpopulation im Datentyp *genus* gebraucht, der in Abbildung 4.4 dargestellt ist.

Für alle selektierten Nachkommen–Individuen I^N werden nun die Formeln 4.33 angewendet, bei separaten Schrittweiten entsprechend die Formeln 4.34.

(4.33)
$$\vec{o}_{Ni} = \vec{o}_{Ei} + \frac{1}{k}(\vec{o}_{Ni} - \vec{o}_{Ei})$$
$$\delta_{Ni} = \delta_{Ei}(\frac{\delta_{Ni}}{\delta_{Ei}})^{\frac{1}{\kappa}}$$
$$i = 1,\ldots,\mu$$

(4.34)
$$o^j_{Ni} = o^j_{Ei} + \frac{1}{k}(o^j_{Ni} - o^j_{Ei})$$
$$\delta^j_{Ni} = \delta^j_{Ei}(\frac{\delta^j_{Ni}}{\delta^j_{Ei}})^{\frac{1}{\kappa}}$$
$$i = 1,\ldots,\mu \quad , \quad j = 1,\ldots,n$$

4.3 Compilieren von VEES

In diesem Abschnitt wird erläutert, wie man aus den Quelldateien von VEES ein lauffähiges Programm erzeugt. Grundsätzlich können zwei verschiedene Versionen von VEES erzeugt werden: Die 'normale' Version mit textuellem Menü und eine Version, die mit graphischer Oberfläche unter X–Windows läuft. Beide Versionen sind für den Batchbetrieb (s. 4.4.3) geeignet. Die graphische Oberfläche wird in [Hasel 95] ausführlich beschrieben, deshalb geht diese Arbeit nur am Rande darauf ein.

Um VEES compilieren zu können, werden die Quelldateien in den Verzeichnissen `veea/source/` und `ea/source/menu/` benötigt. Das menügeführte Skript und dessen Hilfsdateien, die zur Übersetzung von VEES gebraucht werden, befinden sich im Verzeichnis `make/` (siehe Anhang C). Soll die Version von VEES erzeugt werden, die mit der graphischen Oberfläche interagieren kann, so sind zusätzlich die Dateien aus dem Verzeichnis `ea/source/rpc/` vonnöten.

Die Compilierung von VEES wird mit der Skriptdatei `make/build` gestartet, die über eine einfache menügesteuerte Benutzerführung verfügt. Zuerst muß man in das Verzeichnis `make/` wechseln und hier das Skript `build` starten:

```
cd make; build
```

Auf der Paragon verwendet man dafür am besten einen eigenen Rechenknoten:

```
cd make; isub -sz 1 build
```

Dann erscheint ein kleines Menü, dessen Punkte man durch Eingabe der vorangestellten Zahl (+Return) an- und ausschalten kann:

```
------------------------------------------------------------
Build-Menu - Hosttype paragon
------------------------------------------------------------
       1 VEGA      stand-alone  off
       2 VEGA_UIEA with UIEA    off
       3 VEES      stand-alone  on
       4 VEES_UIEA with UIEA    off

       7 UIEA                   off

       0 Exit/Abort
[RETURN] Run
------------------------------------------------------------
Press a number from above and then [RETURN]...
```

Es können mehrere Ziele gleichzeitig aktiviert sein. Um die textuelle Version von VEES zu erstellen, muß der Punkt `3 VEES stand-alone` auf `on` sein, für die graphische Version entsprechend der Punkt `4 VEES_UIEA with UIEA` und Punkt `7 UIEA`. In der Datei `make/build.doing` können Voreinstellungen des Menüs auf bestimmten Rechnertypen festgelegt werden. Sind die

gewünschten Ziele im Menü aktiviert, so muß nur noch die Return–Taste gedrückt werden, um den Compilierungsvorgang zu starten.

Die ausführbaren Dateien von VEES und diversen Testprogrammen werden während der Übersetzung im Verzeichnis `veea/bin/$HOSTTYPE/` abgelegt. Es sind dies die folgenden Dateien:

`vees`	–	Das Programm VEES
`testGenus`	–	Testprogramm für das Modul vees_Genus
`recombination`	–	Testprogramm für Rekombinationsmethoden
`populationSort`	–	Testprogramm für das Sortieren einer Population
`individualSelection`	–	Testprogramm für die Individuenauswahlverfahren
`bit2ind`	–	Testprogramm für die Umwandlung eines Individuums in ein Byte–Array und umgekehrt (für das Versenden von Individuen)
`test_gnuplot`	–	Testprogramm für Gnuplot–Steuerungsmodul
`testIndOutput`	–	Testprogramm für Individuenausgabe

Die Compilierung kann durch setzen spezieller Flags beeinflußt werden, z. B. können Code–Optimierungen aktiviert werden. Dazu muß im `build`–Skript die folgende Zeile in dem Abschnitt für die entsprechende Computerarchitektur editiert werden:

```
setenv SPECIAL_C_FLAGS      "-nx -Xc -O4 -D_VEES_OPTIMIZE_"
```

Das Flag `-O4` ist für den icc–Compiler der Paragon bestimmt und aktiviert die höchste Stufe der Code–Optimierung. Dadurch dauert es länger, VEES zu übersetzen, aber dafür ist der entstandene Code schneller. Die Definition `-D_VEES_OPTIMIZE_` aktiviert „von Hand“ optimierte Code–Teile. Die in Kapitel 5 gemachten Messungen, wurden mit einer Version von VEES durchgeführt, die mit diesen beiden Flags übersetzt wurde.

4.4 Bedienung

In den folgenden Abschnitten wird gezeigt, wie VEES im interaktiven und im Batch–Betrieb aufgerufen wird. Es wird die Bedienung des textuellen Menüs im interaktiven Betrieb erklärt und kurz auf das graphische Menü eingegangen. Desweiteren werden die Formate der Statistiken erläutert, welche auf dem Bildschirm angezeigt und in Dateien geschrieben werden können.

4.4.1 Aufruf des Programms

In diesem Abschnitt wird beschrieben, wie die textuelle Version von VEES gestartet wird. Um VEES mit der graphischen Oberfläche bedienen zu können, muß auf der Kommandozeile `uiea` eingegeben werden. Die weitere Bedienung ist [Hasel 95] zu entnehmen.

Um VEES aus jedem Verzeichnis heraus starten zu können, sollte in den Suchpfad für ausführbare Programme das Verzeichnis `veea/bin/$HOSTTYPE/` aufgenommen werden. Die Variable `HOSTTYPE` ist normalerweise vom System vordefiniert und enthält einen String, der die Art des Rechners charakterisiert. Um den Suchpfad dauerhaft zu setzen, sollte in der Datei `.cshrc`

(oder einer entsprechenden Datei bei Benutzung anderer Kommando–Shells) die folgende Zeile ergänzt werden:

```
set path=($path $HOME/veea/bin/$HOSTTYPE)
```

Auf einer Workstation wird das Programm einfach durch Eingabe von `vees` aufgerufen:

```
Workstation:~# vees
```

Auf der Paragon wird VEES über den Befehl `isub` gestartet, dem zusätzlich noch angegeben werden muß, auf wievielen Knoten das Programm laufen soll.

```
Paragon:~# isub -sz 17 vees          oder          Paragon:~# isub -sz 4x5 vees
```

Die zweite Alternative gibt dabei die gewünschte zweidimensionale Anordnung der Knoten vor. Zu beachten ist hierbei, daß VEES nach dem Master-Slave–Konzept arbeitet, d. h. daß immer auf einem Knoten weniger gerechnet wird, wie beim `isub`–Kommando angegeben ist. Denn auf einem Knoten läuft grundsätzlich der Master–Prozeß.

VEES verarbeitet auch einige Kommandozeilenparameter, die den Modus beeinflussen, in den sich VEES nach dem Start versetzt. Es existieren die folgenden Modi:

- interaktiver Betrieb
- Batch–Betrieb
- RPC[1]–Betrieb

Sind keine Kommandozeilenparameter angegeben, so startet VEES im interaktiven Betrieb. Es wird das Titelbild und das Menü angezeigt und anschließend auf Benutzereingaben gewartet. Der interaktive Betrieb wird weiter unten erläutert. Zuvor wird kurz auf die beiden anderen Modi eingegangen.

Sind Kommandozeilenargumente angegeben, so wird zuerst getestet, ob es sich um spezielle RPC–Optionen handelt. Wie diese aussehen, soll hier nicht erläutert werden, weil es nicht vorgesehen ist, daß sie vom Benutzer angegeben werden. Sie werden vielmehr automatisch von der graphischen Oberfläche generiert, um eine RPC–Verbindung mit VEES herzustellen. Werden die Argumente als RPC–Argumente erkannt, so startet VEES im RPC–Modus und wird von der graphischen Benutzeroberfläche gesteuert (siehe [Hasel 95]).

Werden die Argumente nicht als RPC–Optionen erkannt, interpretiert VEES alle Kommandozeilenparameter als Dateinamen von Skript–Dateien. Diese werden nacheinander eingelesen und ausgeführt. Für jede der Dateien verhält sich das Programm so, als wäre es neu gestartet worden. Nach der Abarbeitung der Skript–Dateien beendet sich VEES von selbst. Der Batchbetrieb wird in Abschnitt 4.4.3 beschrieben.

Im interaktiven Betrieb meldet sich VEES zuerst mit dem Titelbild und dem Konfigurationsbildschirm, welcher auch während des Betriebes mit dem Befehl `info` aufgerufen werden kann.

[1] Remote Procedure Call

```
+------------------------------------------------------------------------------+
|   _/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/|
|  _/                                                                      _/  |
| _/                              V  E  E  S                              _/   |
|_/                                                                      _/    |
|_/                     VErteilte Evolutions-Strategien                  _/    |
|_/                                                                      _/    |
|_/                            by Juergen Wakunda                        _/    |
|_/                                                                      _/    |
| _/                         Version 1.0 (15.11.1995)                    _/   |
|  _/                                                                      _/  |
|   _/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/|
+------------------------------------------------------------------------------+

+------------------------------------------------------------------------------+
|                            VEES Configuration Info                           |
|                                                                              |
| VEES-Version          :           1.0 (15.11.1995)                         |
| Operating System      :           Paragon OSF/1 Version 1.0.4 Release 1.2.5 |
| Total number of nodes :           17 (1 control, 16 compute)               |
| Mode                  :           Master-Slave                             |
+------------------------------------------------------------------------------+
```

Danach wird das Menü angezeigt und es können Befehle eingegeben werden. Die weitere interaktive Bedienung der Oberfläche folgt im nächsten Abschnitt. Beendet wird VEES durch Eingabe von `quit` oder `q`.

4.4.2 Die Benutzeroberfläche

Das Menü von VEES faßt mehrere Parameter, die zur gleichen Funktionsgruppe gehören, zu einer Klasse zusammen. Beispiele dafür sind Applikationsparameter oder Statistikparameter. Klassen sind in der textuellen Oberfläche durch doppelte Linien voneinander getrennt. In der graphischen Oberfläche gibt es für jede Klasse ein eigenes Fenster. Innerhalb von Klassen wird durch einfache Trennlinien nochmals eine logische Gruppierung vorgenommen.

Die textuelle Oberfläche

Ein Beispiel für das Menü von VEES ist in Abbildung 4.5 dargestellt. Es kann sich je nach gemachten Einstellungen auch sehr abweichend von diesem Beispiel präsentieren. Verschiedene Punkte des Menüs können abhängig von anderen Punkten aktiv oder inaktiv sein. Inaktive Parameter können nicht verändert werden. Durch Eingabe des Befehls `menu` oder `m` wird das Menü mit allen gerade aktiven Parametern angezeigt. Wird z. B. durch Eingabe von `noiseTreatment off` die Kappa/Ka–Rauschbehandlung ausgeschaltet, so werden die beiden Parameter `noiseKappa` und `noiseKa` inaktiv, da sie keine Auswirkungen mehr auf den Ablauf der Evolutionsstrategie haben.

Eine Parameterzeile des Menüs ist folgendermaßen aufgebaut: ganz links steht die Abkürzung des Befehls, in der zweiten Spalte folgt der vollständige Name des Befehls und ganz rechts ist der aktuell eingestellte Wert angezeigt. In Anhang B sind alle Parameter und ihre Bedeutung aufgelistet.

```
+*************************************************************************+
|                              VEES - menu                                |
+*************************************************************************+
| esst   esStrategy                  =                             vees |
+=========================================================================+
| vemi   veMy                        =                                5 |
| veri   veRho                       =                                2 |
| veli   veLambda                    =                               20 |
| vegi   veGamma(generations)        =                               10 |
| veti   veType                      =                            comma |
+-------------------------------------------------------------------------+
| vedi   veDelta                     =                              0.5 |
+-------------------------------------------------------------------------+
| vesa   veStepAdaption              =                               on |
| vesc   veStepControl               =                          uniform |
| veai   veAlpha                     =                              1.3 |
+-------------------------------------------------------------------------+
| veroi  veRhoRecombObjVars          =                         dominant |
| versi  veRhoRecombStratVars        =                     intermediate |
+-------------------------------------------------------------------------+
| vesmi  veSelectionMethod           =                          ranking |
| vergi  veRankingGradient           =                              0.8 |
+=========================================================================+
| noise  noiseTreatment              =                               on |
| kappa  noiseKappa                  =                                2 |
| ka     noiseKa                     =                                2 |
+=========================================================================+
| app    application                 =                              f10 |
| objvar objectVariables             =                              100 |
| con    convergence                 =                               on |
| conlim convergenceLimit            =                           321285 |
| conapp convergenceApproxGrade      =                               95 |
+=========================================================================+
| seed   seedRandomGenerator         =                                1 |
+=========================================================================+
| statg  statisticGnuPlot            =                              off |
| statv  statisticVerbose            =                               on |
| stati  statisticInterval           =                                1 |
| filepi filenameProtocolIndividual  =                         es_pi_01 |
| filec  filenameCollection          =                             es_c |
+*************************************************************************+
| r      run                         | s      step                      |
| c      continueEs                  | i      info                      |
+-------------------------------------------------------------------------+
| m      menu                        | h      help                      |
| q      quit                        |                                  |
+*************************************************************************+
```

Abbildung 4.5: Beispiel für das Aussehen des Menüs

Der Wert eines Parameters kann dadurch geändert werden, daß man seinen Namen (wahlweise auch die Abkürzung) nach dem Eingabeaufforderungsprompt eingibt, gefolgt von dem neuen Wert. Dabei können, bedingt durch die Tcl–Syntax (s. nächster Abschnitt), auch mehrere Einstellungen durch einen Strichpunkt voneinander getrennt angegeben werden:

```
VEES>veLambda 100
VEES>veStepAdaption on ; veStepControl separate
VEES>application f22
```

Im unteren Teil des Menüs stehen die Steuerbefehle, welche keinen Wert annehmen können, sondern zum Auslösen bestimmter Aktionen dienen. Es stehen die folgenden Steuerbefehle zur Verfügung:

Befehl	Abkürzung	Beschreibung
`run`	`r`	Starten der Berechnung der Evolutionsstrategie mit den aktuellen Menüeinstellungen
`continueEs`	`c`	Fortsetzen der Berechnung
`step`	`s`	Berechnen einer einzelnen Generation
`menu`	`m`	Anzeigen des Menüs mit den aktuellen Einstellungen
`help`	`h`	Anzeigen von Hilfsinformationen über einen Befehl
`info`	`i`	Anzeigen von Konfigurations–Information
`quit`	`q`	Beenden von VEES

Der Befehl `run` startet die Berechnung der Evolutionsstrategie. Beendet wird die Berechnung, wenn die eingestellte Zahl an Generationen erreicht ist. Dies ist der Wert von `veGamma` bei VeES–Strategien oder Wert von `nmlGamma` mal `nmlGammaPopulation` bei geschachtelten Strategien. Das Abbruchkriterium kann aber auch das Erreichen eines bestimmten Qualitätswertes oder eine prozentuale Annäherung daran sein. Hierzu muß der Parameter `convergence` auf `on` geschaltet sein. Dadurch werden die Parameter `convergenceLimit` und `convergenceApproxGrade` aktiviert. `convergenceLimit` ist dabei der absolute Qualitätswert, der erreicht werden soll. Bei den in VEES eingebauten Applikationen ist das Optimum bekannt und wird automatisch beim Wechsel auf eine andere Applikation in diesem Parameter eingetragen. Der Parameter `convergenceApproxGrade` ist die prozentuale Annäherung an das Optimum, die erreicht werden soll. Die Abbruchbedingung läßt sich formal so formulieren (q_{max} - Qualität des besten Individuums, q_{limit} - Wert von `convergenceLimit`, r_{approx} - Wert von `convergenceApproxGrade`):

$$q_{max} \geq q_{limit} \cdot \frac{r_{approx}}{100}$$

Wurde die Berechnung beendet, weil die vorgegebene Anzahl an Generationen erreicht ist, so kann sie durch Erhöhen der Generationenzahl und Eingabe des Befehls `continueEs` auf den bisherigen Ergebnissen fortgesetzt werden.

Tcl Skriptbefehle

In die textuelle Oberfläche ist die Sprache *Tcl* integriert. *Tcl* (sprich: „tickel“) steht für *T*ool *C*ommand *L*anguage und ist eine Kommandosprache, mit der man unter anderem Applikationen mit einem Grundvorrat an Befehlen ausstatten kann. Die Applikationen können zusätzlich

eigene Befehle definieren, die die programmspezifische Funktionalität anbieten. Dies wurde so auch in VEES verwirklicht. Alle Eingaben werden vom Tcl–Interpreter entgegengenommen und geparst. Alle Parameter des Menüs sind zusätzlich definierte Befehle, die jeweils den neuen Wert des Parameters entgegennehmen und speichern. Es stehen aber auch alle vordefinierten Tcl–Befehle und Strukturen zur Verfügung, wie z. B. Schleifen, Variablen, Listen, Operatoren und Unterprogramme. Diese können sowohl auf der Kommandozeile von VEES, als auch in Batchskripten verwendet werden.

Tcl bietet damit eine sehr komfortable Möglichkeit, um z. B. Meßreihen mit verschiedenen Parametereinstellungen durchzuführen. Dies wurde auch für die Messungen im Kapitel 5 angewendet. Hier ein Beispiel zur Durchführung einer Meßreihe mit verschiedenen Anfangswerten für den Zufallszahlengenerator:

```
set functionName f10
set numNodes     33
set my            8
set lambda       30
set fileName     ${functionName}_nodes$numNodes

filenameCollection $fileName
application        $functionName

esStrategy                    vees
veLambda                      $lambda
veMy                          $my
veRho                         3
veGamma(generations)          600
veType                        comma
veDelta                       0.5
veAlpha                       1.07
veRhoRecombObjVars            dominant
veRhoRecombStratVars          intermediate
noiseTreatment                on
noiseKappa                    2.0
noiseKa                       2.0
statisticVerbose              on
statisticInterval             10

for {set actualseed 1} {$actualseed <= 10} {incr actualseed} {
  puts "----------------------------"
  puts "Messung mit seed $actualseed my: $my lambda $lambda"
  puts ""

  seedRandomGenerator $actualseed
  filenameProtocolIndividual ${fileName}_seed${actualseed}my${my}lambda${lambda}
  run
}
```

Zu Beginn werden 5 Variablen definiert, die an dieser Stelle abgeändert werden können, um eine Meßreihe mit anderen Werten durchzuführen. Der Name der Sammeldatei wird auf `f10_nodes33` eingestellt (VEES fügt dem noch die Endung `.ESsum` hinzu). Die eigentliche Meßreihe besteht aus 10 Einzelmessungen, welche ihre Ergebnisse in Individuen– und Protokolldateien mit dem Namen `f10_nodes33_seed1my8lambda30` und den von VEES angefügten

Endungen `.ESind` und `.ESpro` ablegen. Bei den erzeugten Dateinamen steht in der Mitte jeweils `seed1` bis `seed10`.

Die for–Schleife wird zunächst mit dem Wert 1 in der Variablen `actualseed` initialisiert. In der zweiten Klammer steht die Schleifenbedingung. Solange der Wert von `actualseed` kleiner oder gleich 10 ist, wird die Schleife ausgeführt. Der Befehl `incr actualseed` erhöht die Variable bei jedem Durchlauf um 1. Die `puts`–Befehle schreiben eine Nachricht auf den Bildschirm. Der Befehl `seedRandomGenerator` ist ein VEES–Kommando und setzt den Wert für den Zufallszahlengenerator, im Beispiel wird dazu der Wert der Schleifenvariablen `actualseed` verwendet. Mit dem Befehl `filenameProtocolIndividual` wird der Name der Protokoll– und Individuendateien auf die oben erläuterten Namen eingestellt. Schließlich wird mit `run` die Berechnung der Evolutionsstrategie gestartet. Die Schleife wird insgesamt 10 mal durchlaufen.

Dies war nur ein kleines Beispiel, um die Fähigkeiten der Tcl–Kommandosprache zu demonstrieren. Sie bietet noch andere Schleifenkonstrukte, weitere Variablenhandhabungs–Möglichkeiten und mehr an. Tiefere Einblicke erhält man z. B. in dem Werk [Ousterhout 95], das vom Autor von Tcl, John K. Ousterhout, stammt.

Die graphische Oberfläche

Die gemeinsame graphische Oberfläche für VEES und die vier weiteren Programme des EA–Projektes wurde in einer separaten Arbeit von Alexander Hasel erstellt [Hasel 95]. Sie läuft unter X–Windows und Motif und bietet exakt dieselben Menüpunkte an, wie die textuelle Oberfläche. Sie hat aber noch weitergehende Funktionalitäten. So können die aktuellen Parametereinstellungen abgespeichert und wieder geladen werden. Es existiert ein HTML[2]–Hilfesystem mit Querverweisen und ein Tool–Menü. Über dieses können Hilfsprogramme aufgerufen werden. Es kann auch leicht um eigene Tools erweitert werden. Schon vorhanden ist die Möglichkeit, mit den aktuellen Parametereinstellungen eine Tcl–Skriptdatei automatisch zu erzeugen. Diese kann anschließend mit dem integrierten Editor noch modifiziert werden.

Die Oberfläche ist aber nicht nur in der Lage, VEES zu starten, sondern kann auch die anderen EA–Programme aufrufen. Diese werden dann über Fern–Prozeduraufrufe (RPC) auf der entsprechenden Computerarchitektur gesteuert. So kann z. B. die Oberfläche auf einer Workstation laufen, VEES wird aber auf der Paragon und MPES auf der MasPar gestartet. An dieser Stelle soll nicht weiter auf die graphische Oberfläche eingegangen werden. Für detailliertere Informationen sei auf die oben genannte Arbeit verwiesen.

4.4.3 Batchbetrieb

VEES geht in den Batchbetrieb, wenn beim Aufruf der Name einer Kommandodatei angegeben wird. VEES liest dann die Kommandos aus der Datei und führt sie aus. Danach wird das Programm beendet. Auf einer Workstation sieht ein Batchaufruf z. B. folgendermaßen aus:

```
Workstation:~# vees batchinput
```

Die Datei `batchinput` enthält dabei die Kommandos für VEES. Dies sind exakt dieselben, wie sie auch im interaktiven Betrieb am Eingabeprompt eingegeben werden können. Als mögliche

[2] *H*yper*t*ext *M*arkup *L*anguage

Befehle stehen dabei einfache Änderungen von Parameterwerten des Menüs zur Verfügung, andererseits können aber auch sämtliche Batch–Kommandos der Skriptsprache Tcl verwendet werden. Ein Beipiel für den Inhalt einer Batchdatei wurde im Abschnitt 4.4.2 gezeigt.

Analog werden auf der Intel Paragon Batchdateien mit VEES abgearbeitet:

```
Paragon:~# isub -sz 17 vees batchinput
```

Dadurch wird VEES sofort gestartet. Auf der Paragon ist es aber auch möglich, Batchjobs an eine Batch–Warteschlange zu übergeben, welche sich dann um die Ausführung des Jobs kümmert. Dies bietet sich vorallem dann an, wenn eine besonders lange Berechnung gemacht werden soll, oder wenn besonders viele Knoten benötigt werden, die man im interaktiven Betrieb kaum erhält.

Für Batchjobs auf der Paragon wird statt des Befehls `isub` der Befehl `qsub` benutzt. Allerdings müssen hierbei noch weitere Angaben gemacht werden. Batchjobs werden hauptsächlich in den Nachtstunden gestartet. Wenn tagsüber keine interaktiven Jobs laufen, wird ebenfalls auf Batchbetrieb umgestellt. Aktuelle Informationen darüber, und wie ein Batchjob gestartet wird, sind auf der Paragon durch Eingabe von `news Betrieb` abrufbar. Ein typischer Batchaufruf auf der Paragon sieht folgendermaßen aus:

```
Paragon:~# qsub -lP 33 -lT 1800 startscript
```

Mit der Option `-lP` wird die Anzahl der Rechenknoten bestimmt, auf denen der Batchjob läuft. Durch `-lT` wird die maximale Rechenzeit des Jobs festgelegt. Im Beispiel sind dies 1800 Sekunden = 30 Minuten. Sollte sich das Programm innerhalb dieser Zeitspanne nicht von selbst beendet haben, so wird es abgebrochen. Die Rechenzeit wird also am besten inklusive einer gewissen Pufferzeit angegeben. Allerdings sollte diese nicht zu hoch gewählt werden, da sich aus der Zahl der Knoten und der Laufzeit die Priorität des Jobs errechnet. Ist diese niedrig, so werden kürzere Jobs vorgezogen.

Die Datei `startscript` kann z. B. folgenden Inhalt haben:

```
. /etc/profile                  # Durchlauf der
. $HOME/.profile                # login-Dateien

cd evolution/batchdir           # ins Arbeitsverzeichnis wechseln
vees batchinput                 # VEES mit Skriptdatei starten
```

Zuerst werden die Benutzereinstellungen, z. B. Umgebungsvariablen, Pfade, etc., eingelesen. Da das Skript immer im Home–Verzeichnis des Benutzers startet, muß zunächst in das gewünschte Arbeitsverzeichnis gewechselt werden, in dem sich auch die Batchdatei für VEES befindet. Dann wird VEES aufgerufen. Die Datei `batchinput` enthält alle Befehle, die VEES ausführen soll. Sind diese abgearbeitet, so beendet sich das Programm.

4.4.4 Statistiken

Damit der Ablauf von Evolutionsstrategien mitverfolgt und überwacht werden kann, bietet VEES die Möglichkeit, statistische Daten zu erfassen und auszugeben. Diese können sowohl im interaktiven Betrieb auf dem Bildschirm angezeigt, als auch in Dateien mitprotokolliert werden. Letztere Möglichkeit ist vorallem zur Durchführung von Meßreihen und im Batchbetrieb von Nutzen.

Gnuplot Visualisierung

Der Verlauf der erreichten Qualitätswerte kann während des Ablaufs in einem Schaubild graphisch dargestellt werden. Dies wird über ein Ausgabefenster des Programmes Gnuplot erreicht. Zur Aktivierung der Ausgabe muß der Menüpunkt `statisticGnuPlot` auf `on` geschaltet werden. Die Statistikdaten werden dann im Intervall, das der Parameter `statisticInterval` bestimmt, in das Gnuplot–Fenster ausgegeben. Ein Beispiel für das Aussehen der graphischen Ausgabe ist in Abbildung 4.6 dargestellt.

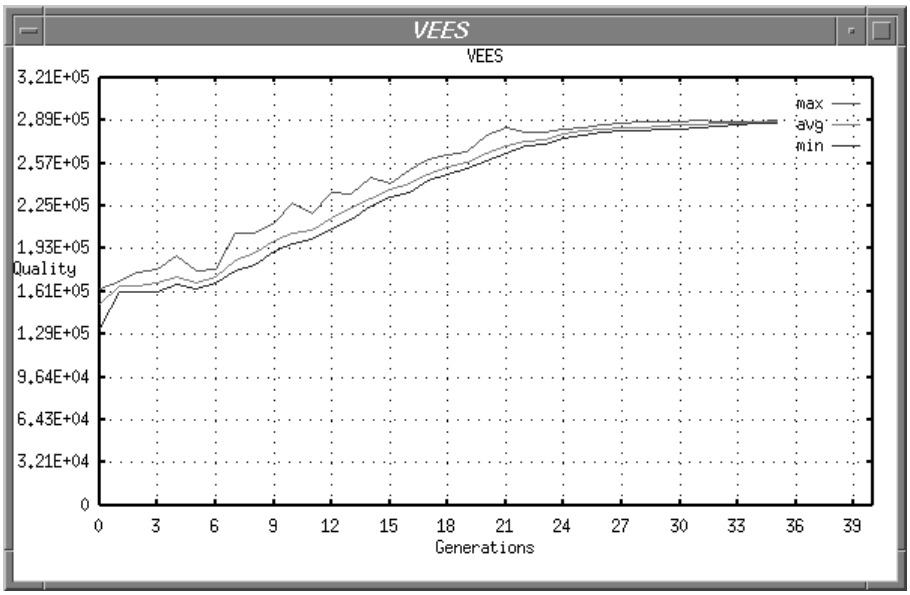


Abbildung 4.6: Graphische Ausgabe von Statistikdaten mit Gnuplot

Angezeigt wird der Verlauf der folgenden drei Werte:

- die beste Qualität
- die durchschnittliche Qualität
- die schlechteste Qualität

Diese Größen beziehen sich jeweils auf alle Individuen aller Prozessoren. Wird der Schalter `statisticGnuPlot` auf `off` gestellt, so schließt sich das Ausgabefenster wieder.

Text– und Datei–Ausgabe

Mit dem Schalter `statisticVerbose` kann die Text– und die Dateiausgabe eingeschaltet werden. Die Textausgabe gibt die weiter unten erläuterten Statistikdaten zeilenweise auf den Bildschirm aus. Die Dateiausgabe erzeugt insgesamt drei verschiedene Dateien, wobei in eine davon die gleichen Daten geschrieben werden, wie sie auch auf dem Bildschirm erscheinen. Text– und Dateiausgabe sind also eng miteinander verknüpft. Sie können nur gemeinsam aktiviert oder

inaktiviert werden. Der Parameter `statisticInterval` bestimmt das Ausgabeintervall für beide Ausgabearten.

Die von der Textausgabe erzeugten Datenzeilen sehen beispielsweise so aus:

```
0       175266.0876902287400    150294.4558697005200    127445.0558029104400  |  0.281250s
 |  0.000000s  0.000000s  0.048720s  0.000000s  |  0.000163s  0.000000s
10      182991.5360898718000    167697.2749275536000    158946.3863990280600  |  3.537109s
 |  0.770996s  1.056424s  1.193414s  0.099338s  |  0.142687s   0.037489s
```

Die Einträge einer Zeile bedeuten von links nach rechts:

1. die Nummer der aktuellen Generation
2. die beste Qualität
3. die durchschnittliche Qualität
4. die schlechteste Qualität

Die drei Qualitätswerte sind dieselben, die auch in im Gnuplot–Fenster angezeigt werden können. Nach den Qualitäten folgen durch einen senkrechten Strich getrennt, die Zeitstatistiken. Die gemessenen Zeiten sind dabei in dieser Reihenfolge:

1. die Gesamtzeit
2. die Duplikations– und Rekombinationszeit
3. die Mutationszeit
4. die Applikationszeit
5. die Selektionszeit
6. die Wartezeit
7. die Austauschzeit

Die Gesamtzeit des Algorithmus wird im Master gemessen und beinhaltet alle Berechnungs– und Nachrichtenzeiten. Alle anderen Zeiten werden in den Slaves gemessen und sind Durchschnittswerte. Sie stellen eine genauere Aufschlüsselung der Gesamtzeit dar und sind von dieser durch einen senkrechten Strich getrennt. Der erste Wert beinhaltet die Zeit, die zur Duplikation oder, falls verwendet, zur Rekombination gebraucht wurde. Dann folgt die Zeit für die Mutation. Die Applikationszeit stellt die Zeit dar, die für die Berechnung der Qualität von Individuen benötigt wurde. Danach folgt die Zeit, die für die Selektion von Individuen aufgewendet wurde. Die letzten beiden Zeiten sind wieder durch einen Strich abgetrennt und erscheinen nicht, wenn nur ein Prozessor benutzt wurde. Bei der ersten handelt sich um die durchschnittliche Wartezeit der Slaves, die diese untätig auf Nachrichten des Masters gewartet haben. Die Wartezeit kann nur berechnet werden, wenn mindestens zwei Prozessoren verwendet wurden. Die Austauschzeit gibt die bisher für die Phase des Individuenaustausches

aufgewendete Rechenzeit wieder. Individuenaustausch kann nur stattfinden, wenn mindestens zwei Rechenprozessoren, also insgesamt mindestens drei Prozessoren, verwendet wurden. Deshalb erscheint die Austauschzeit nur ab drei Prozessoren.

Die letzte Sektion (im obigen Beispiel weggelassen) enthält die drei Zeiten, die für die evolutionsstrategischen Methoden auf Populationsebene aufgewendet wurden. Sie erscheinen deshalb nur bei geschachtelten My-Lambda–Strategien. Es handelt sich dabei um die Duplikations–/Rekombinationszeit, die Mutationszeit und die Selektionszeit.

Das erste mal werden die Statistikdaten noch vor Berechnung der ersten Generation, aber nach der Initialisierung aller Slaves mit Populationen erfaßt. Deshalb kann es sein, daß wie im Beispiel, die Gesamtzeit bei der Generation 0 nicht 0 Sekunden ist, sondern die Zeit die zur Initialisierung benötigt wurde. Dasselbe gilt für die Applikationszeit, denn bei der Initialisierung der Anfangspopulation wurde auch schon die Qualität der Elternindividuen mit der Applikationsfunktion berechnet.

Wenn Statistikdaten auf den Bildschirm erscheinen, so werden sie auch gleichzeitig in Dateien geschrieben. Insgesamt werden drei verschiedene Dateien erzeugt:

1. Die Protokolldatei
2. Die Individuendatei
3. Die Sammeldatei

In der Protokoll– und Individuendatei befinden sich jeweils die Daten eines einzelnen Durchlaufes (Session) einer Evolutionsstrategie. Wird deren Name nicht geändert, so überschreibt ein erneuter Durchlauf evtl. schon vorhandene Dateien. Die Sammeldatei dagegen enthält die gesammelten Daten mehrerer Durchläufe (Summary) und wird nie überschrieben. Alle neuen Daten werden am Ende angehängt. Deshalb kann für die Sammeldatei auch ein separater Name angegeben werden. Er wird vom Parameter `filenameCollection` festgelegt. Der Name der Protokoll– und Individuendatei, der sich nur durch die automatisch erzeugten Endungen (s. o.) unterscheidet, kann mit dem Parameter `filenameProtocolIndividual` eingestellt werden.

Die Protokolldatei enthält zuerst die Parameterwerte des Durchlaufes, wie sie im Menü eingestellt waren. Dann folgt ein Abschnitt der jeder Spalte der nachfolgenden Protokolltabelle über eine Nummer ihre Bedeutung zuordnet. Die danach folgende Protokolltabelle enthält genau dieselben Statistikdaten, wie sie auch die Bildschirmausgabe erzeugt (s. o.). Sie ist mit einer Zeile überschrieben, die die Spaltennummern angibt.

Die Individuendatei enthält für jeden Prozessor das beste Individuum, das im bisherigen Verlauf der Strategie aufgetreten ist. Die Individuen werden jedesmal in diese Datei geschrieben, wenn ein `run`–, `step`– oder `continueEs`–Kommando beendet wurde. Die Individuendatei enthält jeweils in einer Zeile die folgenden Einträge:

1. Die Nummer des Prozessors beginnend mit 0
2. Die Qualität des Individuums
3. Alle Objektvariablen des Individuums

Die drei Statistikdateien beginnen alle mit einer eindeutigen Kopfzeile, mit der ihr Inhalt und Format eindeutig bestimmt werden kann. Außerdem werden von VEES spezifische Endungen automatisch an die Dateinamen angehängt. Die Endungen und Kopfzeilen lauten folgendermaßen:

Datei	**Endung**	**Kopfzeile**
Protokoll	`.ESpro`	`Evolution Strategy Session Protocol File Version 1.0`
Individuen	`.ESind`	`Evolution Strategy Session Individual File Version 1.0`
Sammel	`.ESsum`	`Evolution Strategy Summary Protocol File Version 1.0`

4.5 Aspekte der Implementierung

In den folgenden Abschnitten wird erläutert, unter welchen Gesichtspunkten des Software–Engineerings der Code von VEES geschrieben wurde. Es wird die komplette Programmarchitektur erläutert und auf die Realisierung der Einprozessorversion eingegangen. Außerdem werden Besonderheiten im Mechanismus des Datenaustausches besprochen.

4.5.1 Programmiergrundsätze

Namenskonventionen

Für alle Teilprojekte in der Gruppe Evolutionäre Algorithmen (EA) wurden einheitliche Namenskonventionen festgelegt. Jedem Teilprojekt wurde dabei ein eindeutiges Kürzel zugeordnet, das in Modul–, Prozedur– und Typnamen erscheint. Für gemeinsam genutzten Code verwandter Programme wurden zusammenfassende Kürzel festgelegt. Der Begriff 'Evolutions–Algorithmen' umfaßt dabei die beiden Arten 'Genetische Algorithmen' und 'Evolutionsstrategien'.

Kürzel	Verwendung für
`mpga`	*M*assiv *P*arallele *G*enetische *A*lgorithmen
`mpes`	*M*assiv *P*arallele *E*volutions–*S*trategien
`vega`	*V*erteilte *G*enetische *A*lgorithmen
`vees`	*V*erteilte *E*volutions–*S*trategien
`cnga`	*CNAPS G*enetische *A*lgorithmen
`uiea`	*U*ser *I*nterface *E*volutions–*A*lgorithmen
`mpea`	*M*assiv *P*arallele *E*volutions–*A*lgorithmen von mpga und mpes gemeinsam genutzte Teile
`veea`	*V*erteilte *E*volutions–*A*lgorithmen von vega und vees gemeinsam genutzte Teile
`ea`	*E*volutionäre *A*lgorithmen von allen Programmen gemeinsam genutzte Teile

Tabelle 4.1: Bedeutung der Projekt–Kürzel

Diese Kürzel wurden wie folgt verwendet:

- Alle File- und Prozedurnamen enthalten das Kürzel als Präfix klein geschrieben mit einem Unterstrich vom Rest des Namens getrennt: `vees_...`.
- Alle Typbezeichner enden mit dem String `Type`, vor dem unmittelbar das Kürzel mit großgeschriebenem Anfangsbuchstaben steht: `...VeesType`.
- Alle Konstantennamen werden großgeschrieben und erhalten das Kürzel als ebenfalls großgeschriebenes Präfix, welches mit einem Unterstrich vom Rest des Namens getrennt ist: `VEES_...`.

In VEES wurden außerdem noch die folgenden Namenskonventionen eingehalten:

- Konstantenbezeichner werden durchweg groß geschrieben; Teilworte werden durch einen Unterstrich voneinander getrennt. Beispiel: `VEES_NO_QUALITY`.
- Alle sonstigen Bezeichner beginnen mit einem Kleinbuchstaben; Teilworte werden durch Großschreiben des ersten Buchstabens kenntlich gemacht.
  Beispiele: `numberOfGenerations`, `individualVeesType`.
- Zeigertypen haben das Kürzel `Ptr` direkt vor dem normalen Typ–Postfix `VeesType`.
  Beispiel: `individualPtrVeesType`.

Code–Verwaltung

Da in der EA–Gruppe mehrere Projekte gleichzeitig bearbeitet wurden, die teilweise auch dieselben Quelldateien benutzen, wurde es für sinnvoll erachtet, die Quelldateien mithilfe eines Codeverwaltungs–Werkzeuges zu pflegen. Dadurch können Konflikte zwischen gleichzeitigen Änderungen zweier Autoren an derselben Datei leichter aufgelöst werden. Dies trat an mehreren Stellen auf, weil z. B. die graphische Benutzeroberfläche in alle 5 EA–Programme integriert werden mußte und auch die Tcl–basierte Textoberfläche von mehreren Programmen gleichzeitig benutzt wird.

Für diese Verwaltungsaufgabe wurde des System CVS *(Concurrent Versions System)* eingesetzt. Das Grundprinzip dieses Systems ist folgendes: Die Verwaltung legt an einer zentralen Stelle, dem sogenannten *Repository*, alle Quelldateien eines oder mehrerer Projekte ab. Jeder, der nun mit den Dateien arbeiten will, bekommt von CVS eine lokale Kopie der Dateien. Hat ein Benutzer größere Änderungen gemacht, so kann er die neue Version der Datei mit einem CVS–Befehl an das Repository übergeben. Dies funktioniert nur, solange noch kein anderer Benutzer Änderungen an dieser Datei gemacht hat. Wurde die Datei von zweiter Seite verändert, zeigt sich dies dadurch, daß sich die veränderte Datei als neue Version im Repository befindet. In diesem Fall wird die Version des Benutzers zuerst mit der neuen Version gemischt. Ergeben sich dabei Konflikte, d. h. wurden sich widersprechende Änderungen an denselben Zeilen der Datei gemacht, so müssen diese vom Benutzer aufgelöst werden. Erst danach, oder wenn es keine Konflikte gab, kann die Datei nun an das Repository übergeben werden. Damit jedem Benutzer immer die aktuellsten Änderungen der anderen Autoren zur Verfügung stehen, sollte die lokale Kopie regelmäßig aufgefrischt werden.

Es hat sich gezeigt, daß sich nur in seltenen Fällen Konflikte ergeben, die vom Benutzer aufgelöst werden müssen. Meistens werden gleichzeitige Änderungen nicht in überschneidenden Bereichen gemacht, dann genügt die automatische Mischung der veränderten Dateien.

Einheitliche Header

Alle Programmdateien im EA–Projekt haben einen einheitlichen Vorspann (Header), in dem die folgenden Informationen über die Datei enthalten sind:

- Art der Datei: Export- oder Implementierungsteil des Moduls
- Autor
- Zweck des Moduls
- zu welcher Arbeit die Datei gehört
- Ziel–Computer-Architektur
- verwendete Programmiersprache
- verwendeter Compiler
- verwendetes Betriebssystem

Außerdem sind in dem Vorspann CVS–Variablen enthalten, die bei jedem Registierungsvorgang einer neuen Version der Datei in das zentrale CVS–Repository aktualisiert werden. Diese Variablen bewirken, daß in den Vorspann automatisch folgende Informationen eingefügt werden:

- Verzeichnis und Name der Datei im CVS–Repository
- Autor der letzten Änderung
- Versionsnummer und Datum
- 'Log' mit Versions–Historie der Datei und Kommentaren dazu

Fehlerbehandlung

Die Fehlerbehandlung wurde leicht abgewandelt aus dem Programm VEGA übernommen. Das Prinzip von VEGA hat den Vorteil, daß es Programme robust macht, der Programmtext aber gut lesbar bleibt, weil er nicht mit Fehler–Abfragen überfrachtet ist. Diese Arbeit übernehmen Makros. Der einzige Unterschied in VEES ist der, daß sich alle Fehlermeldungen zentral in der Datei `vees_ErrorHandling.h` befinden und nicht lokal in jeder Prozedur neu definiert werden müssen.

Die Fehlerbehandlung funktioniert folgendermaßen: Jeder Prozeduraufruf gibt im Falle eines aufgetretenen Fehlers einen besonderen Wert zurück, der von den normalen Rückgabewerten unterschieden werden kann. Zu diesem Zweck wurde auch für jeden Datentyp ein besonderer

Fehlerwert definiert. Beispielsweise existiert zum Typ für Qualitäten (qualityVeesType, definiert als C–Typ `double`) die Konstante `VEES_NO_QUALITY`, die als `-MAXDOUBLE` definiert ist. Da für Qualitätswerte festgelegt ist, daß sie größer oder gleich 0 sein müssen, kann der Wert `-MAXDOUBLE` also nie bei korrekten Qualitäten auftreten.

Diese Fehlerwerte werden nun bei jedem Aufruf einer Prozedur abgefragt. Zu diesem Zweck existieren verschiedene Makros, von denen hier stellvertretend das Makro für `NULL`–Zeiger abgebildet ist.

```
#define VEES_EXEC_NULL( statement, ErrMsg )                    \
                                                               \
        if ( RetVal == OK ) {                                  \
          if ( (void*) (statement) == NULL ) {                 \
            RetVal=FAIL;                                       \
            veea_Error( VEES_PROCEDURE_NAME, ErrMsg );         \
          } /* if */                                           \
        } /* if */
```

Damit das Makro verwendet werden kann, müssen die folgenden Variablen in der Prozedur, die es benutzt, deklariert sein:

```
static const char * const VEES_PROCEDURE_NAME = "Name";
RetValVeeaType RetVal=OK;
```

Benutzt wird das Makro beispielsweise so:

```
VEES_EXEC_NULL( objVars = vees_individualNewArrayObjectVars( 5 ),
                VEES_ERROR_NO_MEMORY_FOR_OBJECT_VARS );
```

Die Prozedur `vees_individualNewArrayObjectVars` legt Speicherplatz für die angegebene Anzahl an Objektvariablen an. Ist dies nicht möglich, weil nicht mehr genügend Speicherplatz vorhanden ist, oder weil die übergebene Anzahl kleiner oder gleich 0 ist, dann gibt sie den Wert `NULL` zurück. Dieser wird durch das `VEES_EXEC_NULL`–Makro erkannt und daraufhin eine Fehlermeldung ausgegeben. Die Fehlermeldungen sind ebenso wie die `VEES_EXEC`–Makros in der Datei `vees_ErrorHandling.h` definiert.

Das Makro bewirkt aber nicht nur die Ausgabe der Fehlermeldung, sondern setzt auch den Wert der Variablen `RetVal` auf `FAIL`. Dadurch wird ein kontrollierter Programmabbruch ausgelöst. Jedes folgende `VEES_EXEC`–Makro bewirkt nun wegen der Abfrage von `RetVal`, daß der enthaltene Befehl erst gar nicht mehr ausgeführt wird.

Am Ende jeder Prozedur steht dann noch das Makro `VEES_RETURN`, welches im Fehlerfalle den Fehlerwert an die aufrufende Prozedur zurückgibt. Diese löst dann wiederum eine Fehlermeldung aus, usw. bis die höchste Prozedureebene erreicht ist. Der Fehlerfall wird mithilfe der Variablen `RetVal` erkannt. Wenn kein Fehler auftritt, wird das normale Ergebnis zurückgegeben.

Außer `VEES_EXEC_NULL` existieren noch die Makros `VEES_EXEC_ZERO` für Integer–Rückgabewerte, `VEES_EXEC_BOOL` für boolesche Rückgabewerte, `VEES_EXEC_FAIL` für Werte vom Typ `RetValVeeaType`, der ein allgemeiner Rückgabewert ist, und für Ausdrücke ohne Rückgabewert `VEES_EXEC`. Das letztere Makro kann selbst keinen Fehler auslösen, bewirkt aber, daß wenn schon ein Fehler ausgelöst wurde, der enthaltene Code nicht mehr ausgeführt wird.

4.5.2 Wiederverwendung existierender Module

Ein erklärtes Ziel bei der Entwicklung von VEES war, fertige Module von VEGA ganz zu übernehmen oder entsprechend zu erweitern, sofern dies möglich ist. Dies hat in erster Linie natürlich den Vorteil, daß Programmierarbeit eingespart wird. Außerdem entfällt auch Testaufwand, da die Module in VEGA bereits ausgiebig benutzt worden sind, und ihre Funktionsfähigkeit bewiesen haben.

Die übernommenen Module wurden auf die in Abschnitt 4.5.1 beschriebenen Namenskonventionen gebracht. Hierzu wurde in Modul–, Prozedur–, Typ– und Variablen–Namen das Kürzel `vega` in `veea` umgeändert. Dadurch kann man schon anhand des Namens erkennen, welche Objekte gemeinsam von VEGA und VEES benutzt werden.

Ganz übernommen wurden die folgenden Module:

- `veea_defines` (allgemeine Definitionen und Makros)
- `veea_ErrorHandling` (Fehlerausgabe)
- `veea_TraceHandling` (Ausgabe von Trace–Informationen)
- `veea_GnuplotHandling` (Graphische Statistik–Ausgabe mit Gnuplot)
- `veea_TimerHandling` (Prozeduren zur Zeitmessung)

An den folgenden Modulen wurden Erweiterungen vorgenommen:

- `veea_types` (Typen und Konstanten, erweitert um eigene Typen für VEES)
- `veea_MessageHandling` (Nachrichtenprozeduren, erweitert um eigene Nachrichtentypen)

4.5.3 Programmarchitektur

Das Programm ist nach dem Master-Slave–Prinzip konzipiert, d. h. es existiert ein Masterknoten, auf dem die Benutzerschnittstelle läuft und der den Ablauf der Evolutionsstrategie steuert. Der Kern der eigentlichen Evolutionsstrategie läuft auf den Slave–Knoten ab. Die Aufrufhierarchie des Programmes ist in Abbildung 4.7 dargestellt. Allgemeine Zugriffsmodule, wie z. B. `vees_Parameters` oder `veea_types`, die von vielen Modulen benutzt werden und somit die Darstellung unübersichtlich machen würden, sind dort weggelassen. Eine vollständige Aufstellung aller Dateien von VEES befindet sich in Anhang C.

Beim Start auf n Rechnerknoten prüft jeder Knoten seine Nummer. Derjenige mit der höchsten Knotennummer $(n-1)$ wird der Master. Die Knoten mit den Nummern $0 \ldots n-2$ sind Slaveknoten und verzweigen in die Slave–Warteschleife. Die Hauptaufgabe der `main`–Prozedur ist es, diese Unterscheidung zu treffen. Auf dem Masterknoten wird die Benutzerschnittstelle zur textuellen oder graphischen Oberfläche aufgerufen, welche dann wiederum die Steuerbefehle des Masters startet.

Das Modul `vees_Master` bekommt von der Benutzeroberfläche einen der drei Befehle `run`, `step` oder `continueEs` übergeben. Es sendet den Typ der auszuführenden Strategie (*VeES* oder

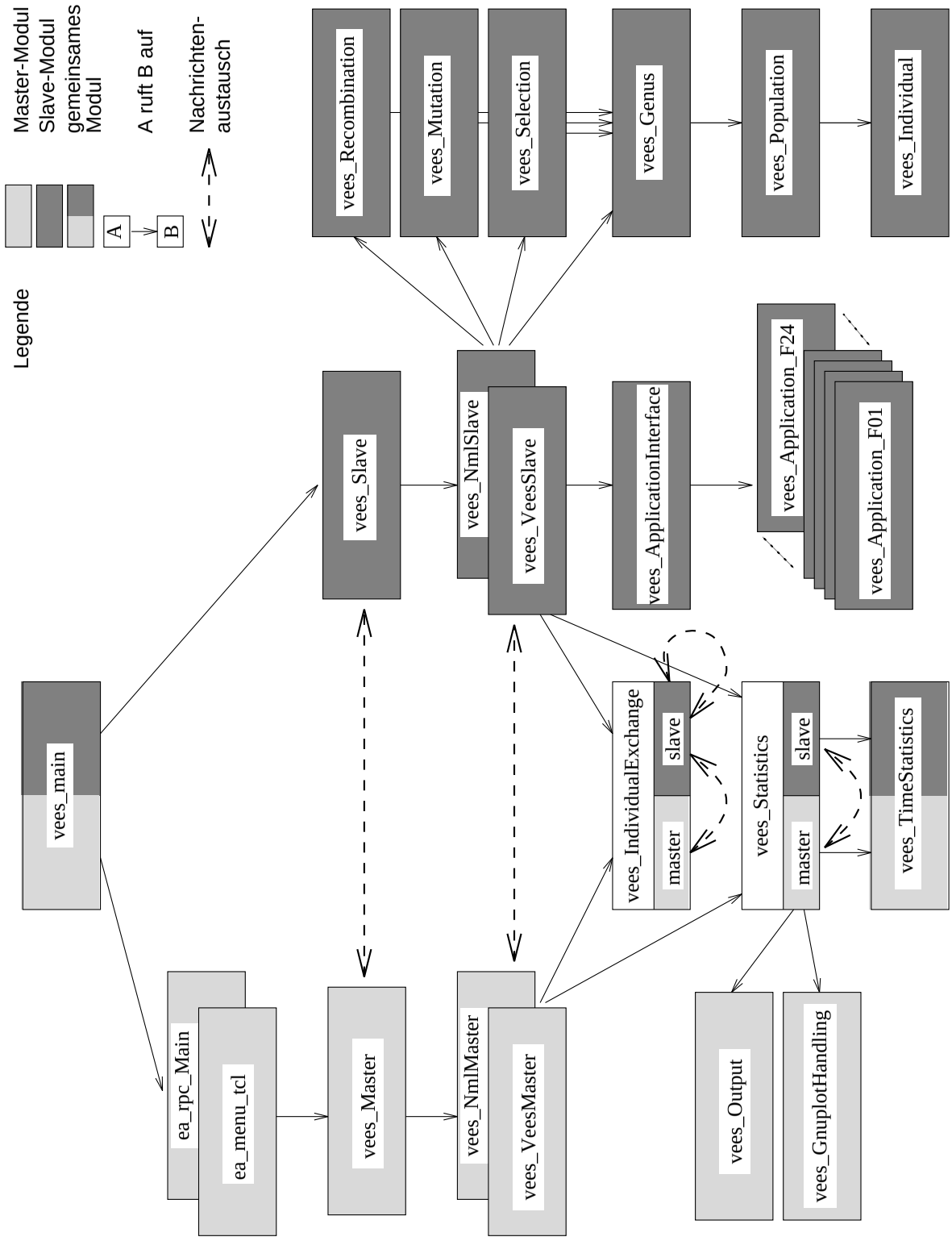


Abbildung 4.7: Aufrufhierarchie und Nachrichtenaustausch in VEES

geschachtelte My-Lambda) an die Slaves, die sich in einer Warteschleife im Modul `vees_Slave` befinden. Der Master ruft dann den entsprechenden Befehl in dem durch die Strategieart festgelegten Master–Modul auf (`vees_VeesMaster/vees_NmlMaster`). Dabei wird übergeben, ob die Slaves neu initialisiert werden müssen. Dies ist nicht immer der Fall, z. B. wenn der Befehl `continueEs` an der Stelle fortfahren soll, an der ein voriger Lauf unterbrochen wurde. Die Slaves empfangen die Nachricht mit der Strategieart und rufen ebenfalls das speziellere Slave–Modul auf (`vees_VeesSlave/vees_NmlSlave`).

Das Mastermodul der gewählten Strategieart steuert dann den kompletten Ablauf der Strategie, d. h. es sendet Steuerbefehle an die Slaves. Die Steuerbefehle lösen dann auf Seite der Slaves eine der folgenden Aktionen aus:

- Empfangen aller nötigen Parameter und Initialisierung der Populationen
- Berechnung einer gewissen Anzahl von Generationen, ohne weitere Kommunikation
- Individuenaustausch mit den anderen Slave–Knoten
- Berechnung der Statistikdaten und Übermittlung an den Master
- Übermittlung des besten Individuums an den Master
- Beendigung des speziellen Slave–Moduls und Rückgabe der Kontrolle an das allgemeine Slave–Modul (`vees_Slave`).

Außer der Steuerung des Algorithmus führt der Master noch die Ausgabe der Statistikdaten durch. Diese werden auf den Bildschirm, in Dateien (`vees_Output`) und in ein Fenster zur Visualisierung der Qualitätsdaten (`veea_GnuplotHandling`) ausgegeben.

Die gemeinsam genutzten Module `vees_IndividualExchange und vees_Statistics` werden jeweils synchron vom Master und den Slaves aufgerufen. Die jeweiligen Master– und Slave–Teile kommunizieren dann miteinander. Beim Individuenaustausch kommunizieren zusätzlich die Slaves untereinander.

Das Hauptmodul, das die Slaves zur Verwaltung und Manipulation der Populationen benutzen, ist `vees_Genus`. Der abstrakte Datentyp *genus* (s. Abschnitt 4.2.1), den das Modul zur Verfügung stellt, bietet alle Prozeduren, die für die evolutionsstrategischen Methoden Duplikation, Rekombination, Mutation, Selektion und Kappa-Ka–Rauschbehandlung nötig sind. Das genus–Objekt wird in den spezialisierten Slaves angelegt. Es wird sowohl direkt von den Slave–Modulen darauf zugegriffen, als auch von den Methoden–Modulen `vees_Recombination`, `vees_Mutation` und `vees_Selection`.

Die Applikationen sind über das Schnittstellenmodul `vees_ApplicationInterface` angekoppelt. Dieses liefert Informationen über die Anzahl der verfügbaren Applikationen, den Applikationsnamen, den Beschränkungstyp (s. 4.6), die voreingestellte Mutations–Schrittweite, die Anzahl der Dimensionen, für die die Applikation lauffähig ist, und den wahrscheinlichen Qualitätswert des absoluten Optimums der Qualitätsfunktion. Weiterhin bietet es Zugriff auf die Qualitätsfunktion selbst und die Initialisierungs– bzw. Beendigungsfunktion jeder Applikation. Die Anbindung neuer Applikationen wird in Abschnitt 4.6 detailliert erläutert.

4.5.4 Einprozessorversion

Da sich die Programmierung der Intel Paragon nur durch die hinzukommende Kommunikation über die *nx*–Nachrichtenprozeduren von der Programmierung einer UNIX–Workstation unterscheidet, bietet es sich an, VEES auch auf Einprozessor–Rechner anzupassen. Damit kann das Programm auch auf der großen Anzahl an vorhandenen Workstations genutzt werden. Allerdings geht hierbei natürlich der Vorteil des Geschwindigkeitsgewinns durch die parallele Ausführung verloren. Aber auch im Einprozessorbetrieb läßt sich VEES sinnvoll einsetzen. Beispielsweise kann die Funktionsfähigkeit neuer Applikationen auch ohne parallele Ausführung getestet werden. Der Einprozessorbetrieb eignet sich weiterhin gut zur Einarbeitung in das Programm und für die ersten Experimente neuer Benutzer mit Evolutionsstrategien. Ferner ist nicht jede Applikation so komplex, daß sie den Einsatz eines Parallelrechners rechtfertigt.

Funktionsprinzip

Im vorangegangenen Abschnitt wurde die Master-Slave–Architektur von VEES beschrieben. Für diese Architektur müssen mindestens zwei Prozessoren zur Verfügung stehen – einer für den Master und einer für den Slave. VEES ist jedoch auch auf nur einem Prozessor der Paragon oder einer Workstation lauffähig. Dazu müssen dann die Master– und Slave–Teile des Programms auf demselben Prozessor ablaufen. Eine einfache Möglichkeit, dies zu realisieren, wäre gewesen den steuernden Teil des Algorithmus aus dem Master mit dem rechnenden Teil aus dem Slave in einem neuen Modul zu mischen. Dies hätte aber den großen Nachteil gehabt, daß duplizierter Code vorläge. In diesem Fall müßten Änderungen an einem Teil des Codes im anderen Teil nachgeführt werden. Dabei entstehen aber sehr leicht Fehler.

Deshalb wurde in VEES ein anderes Prinzip realisiert. Es hat sich gezeigt, daß der Mechanismus, wie der Master den Slave steuert, sehr leicht auf das Prinzip eines synchronen *Remote Procedure Call* (RPC) zurückgeführt werden kann. Dieser kann wiederum auf nur einem Prozessor durch einen direkten Aufruf der (normalerweise entfernten) Prozedur ersetzt werden.

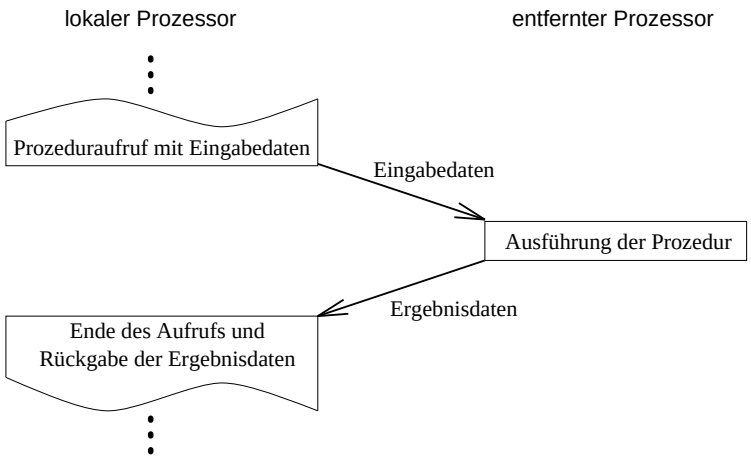


Abbildung 4.8: Prinzip des synchronen *Remote Procedure Call*

Beim RPC ruft das lokale Programm über einen Nachrichtenmechanismus eine Prozedur auf einem entfernten Prozessor auf. Nachdem die Prozedur fertig abgearbeitet wurde, wird wieder über eine Nachricht das Ergebnis der Prozedur zurückgegeben und das lokale Programm bekommt wieder den Kontrollfluß. Während die entfernte Prozedur ausgeführt wird, muß der lokale Prozessor warten. Dieses Prinzip ist in Abbildung 4.8 dargestellt.

Um die Steuerung der Slaves durch Nachrichten auf dieses Prinzip abzubilden, mußte der Ablauf im Master klar in zwei Teile gegliedert werden (s. Abblidung 4.9):

1. Senden einer Nachricht mit Befehl und Daten
2. Empfangen der Ergebnisdaten

Dazwischen darf nichts getan werden. Auf der Seite der Slaves muß der Ablauf in drei Teile aufgetrennt sein:

1. Empfangen der Daten
2. Ausführen der Berechnungen
3. Zurücksenden des Ergebnisses

Das Wichtigste hierbei ist, daß der mittlere Teil zwischen Empfangen und Senden in eine separate Prozedur ausgelagert ist. Diese kann dann, wenn VEES auf nur einem Prozessor läuft, direkt vom 'Master' aufgerufen werden, denn einen Slave gibt es ja nicht. Die Schnittstelle der Prozedur muß genau so beschaffen sein, daß sie die Eingabedaten bekommt und die Ausgabedaten zurückgibt. Wenn sie noch andere Daten zum Rechnen benötigt, die nur im Slavemodul vorhanden sind, so muß auf diese über globale Variablen zugegriffen werden. Ein Beispiel für solche Daten ist das im vorigen Abschnitt angesprochene genus–Objekt.

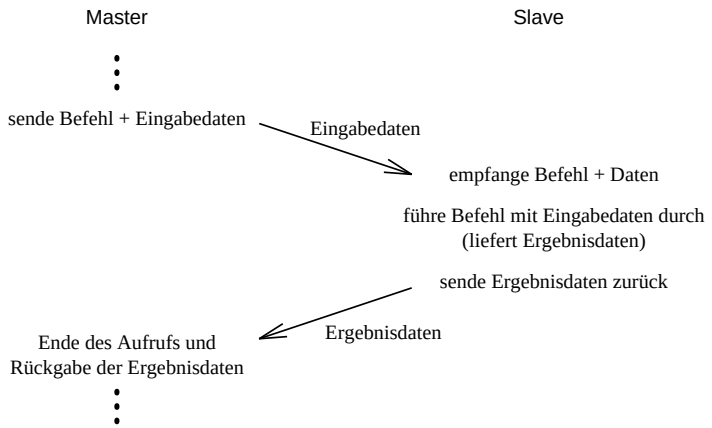


Abbildung 4.9: Steuerung der Slaves durch den Master

Startet man VEES auf der Paragon auf nur zwei Prozessoren, so gibt es einen Master und nur einen Slave. Zwischen diesen finden keine echten parallelen Abläufe statt. Dies ist aber

kein Nachteil, da der Master immer nur kurze Verwaltungsaufgaben übernimmt und den Slave schnell wieder mit Arbeit versorgt. Die echte Parallelität findet zwischen den Slaves statt, wenn mehr als zwei Prozessoren zur Verfügung stehen.

4.5.5 Datenaustausch

Die komplette Kommunikation wird von dem Modul `veea_MessageHandling` gekapselt, welches von VEGA übernommen wurde. Dieses benutzt zum Nachrichtenaustausch die *nx*–Bibliothek der Paragon. Durch die Kapselung ist es ohne große Schwierigkeiten möglich, das Modul auf eine andere Kommunikationsbibliothek umzuschreiben, so daß VEES auch auf anderen verteilten Umgebungen lauffähig wird, z. B. auf einem Netz von Workstations.

Die Kommunikationskanäle zwischen den Prozessoren sind zwar sicher, da aber nur ein begrenzter Pufferplatz für die Nachrichtenspeicherung auf jedem Knoten zur Verfügung steht, kann es in seltenen Fällen zu Nachrichtenverlusten kommen. Dies geschieht allerdings nur, wenn sehr große oder sehr viele Nachrichten übermittelt werden. Bei den kleinen Nachrichten, die zur Steuerung der Slaves und zur Übertragung der Statistikdaten benutzt werden, stellt dies kein Problem dar. Die Begrenzung kann aber beim Individuenaustausch kritisch werden, da hier evtl. viele Daten innerhalb kurzer Zeit übertragen werden. Als Lösung dieses Problems wäre es möglich, den Pufferplatz per Option beim Start des Programms zu erhöhen. Dies schafft das Problem aber nicht aus der Welt, sondern verschiebt es nur. Außerdem existieren keine Richtwerte, wie groß der Puffer zu wählen ist. Deshalb wurde in VEES dieselbe Lösung, wie sie schon in VEGA realisiert wurde, gewählt. Da beim Individuenaustausch die Größe der Nachrichten bekannt ist, kann der Pufferplatz explizit zur Verfügung gestellt werden, noch bevor die Nachricht abgeschickt wird. Erst wenn alle Puffer bereitgestellt sind, wird mit dem Senden begonnen. Dadurch geht zwar etwas an Zeit verloren, andererseits sollte der Individuenaustausch auch nicht nach jeder Generation stattfinden, damit er seinen Zweck erfüllt, die Konvergenz der Evolutionsstrategie zu beschleunigen.

4.6 Anbindung neuer Applikationen

Eine Evolutionsstrategie wird immer auf ein bestimmtes Optimierungsproblem angewendet. Die Funktion, die das Optimierungsproblem beschreibt, ist die Qualitäts– oder auch Fitnessfunktion. Sie gibt die Form des Qualitätsgebirges an, für das der höchste Gipfel gefunden werden soll. Die Qualitätsfunktion inklusive aller weiteren Daten, die zur Beschreibung des Optimierungsproblemes nötig sind, wird Anwendung oder Applikation genannt. In VEES sind bereits einige verschiedene Applikationen integriert, welche in Anhang D aufgelistet sind. Zur Untersuchung von anderen Problemen ist es sehr leicht möglich, neue Applikationen an VEES anzubinden. Wie dies funktioniert, wird in diesem Abschnitt beschrieben.

Wie in Abblidung 4.7 ersichtlich, wird auf die Applikationen nur über die Schnittstelle des Moduls `vees_ApplicationInterface` zugegriffen. Deshalb müssen hier auch Ergänzungen gemacht werden, um eine neue Applikation hinzuzufügen. Dies geschieht grob gesehen in drei Schritten:

1. Erstellen des Applikationsmoduls (‘.c’– und ‘.h’–Datei). Am einfachsten ist es, ein vorhandenes Modul zu kopieren und daran die nötigen Änderungen vorzunehmen.

2. Eintragen der Applikationsdaten und des `include`–Befehls in die Datei `vees_ApplicationInterface.c`.
3. Eintragen der Applikation in die Datei `veea/source/makefile.vees`

Ein Applikationsmodul muß dabei die folgenden 8 Objekte zur Verfügung stellen. Drei dieser Objekte müssen als Funktion implementiert werden, bei den restlichen Objekten handelt es sich um Konstanten.

1. Die Qualitätsfunktion.
2. Die Initialisierungsfunktion.
3. Die Beendigungsfunktion.
4. Die Art der Beschränkung der Objektvariablen.
5. Der Dimensionsbereich, für den die Applikation lauffähig ist.
6. Die Anfangsschrittweite, mit der die Individuen initialisiert werden.
7. Der Qualitätswert des Optimums, sofern bekannt.
8. Der Name der Applikation.

```
double vees_applicationF01Quality ( const int      dimensions,
                                    const double * objVars );

RetValVeeaType vees_applicationF01InitFunction (
                            const int                  dimension,
                            constrFctPtrPtrVeesType * constraintFunctions,
                            rangePtrVeesType        * rangesObjVar );

RetValVeeaType vees_applicationF01ExitFunction (
                            const int                  dimension,
                            constrFctPtrPtrVeesType * constraintFunctions,
                            rangePtrVeesType        * rangesObjVar );
```

Abbildung 4.10: Prototyp–Deklarationen der drei Applikations–Funktionen

Die Qualitätsfunktion ist der wichtigste Teil an der Applikation. Sie bestimmt das zu optimierende Problem durch die Form des „Qualitätsgebirges“. Ein Aufruf dieser Funktion dient dazu, genau ein Individuum zu bewerten. Die Funktion bekommt als Eingabe die Objektvariablen eines Individuums, welche durch ein Feld von `double`–Fließkommazahlen und die Größe dieses Feldes festgelegt sind. Daraus muß sie dann die Qualität des Individuums berechnen, welche sie in Form eines `double`–Wertes zurückgibt. Tritt bei der Qualitätsberechnung ein Fehler auf, so kann dies durch Rückgabe des Fehlerwertes `VEES_NO_QUALITY` angezeigt werden (vgl.

4.5.1). Sehr wichtig ist hierbei die Einschränkung, daß die Qualität nur positive Werte oder Null annehmen darf. Dies ist für die Roulette-Wheel–Selektion von Bedeutung, die nicht mit negativen Werten funktioniert. Die zweite wichtige Eigenschaft der Qualitätswerte ist, daß ein größerer Wert immer eine bessere Qualität bedeutet. Diese beiden Festlegungen stellen aber keine wesentliche Einschränkung dar. Kann die Funktion negative Werte annehmen, so muß sie durch Addition in den positiven Bereich verschoben werden. Genauso kann durch Subtraktion der Qualität von einem Maximalwert die Ordnung der Qualitätswerte umgedreht werden. Da der Wertebereich des Typs `double` sehr groß ist, sollten hier keine Überlaufprobleme auftreten. Der Prototyp der Qualitätsfunktion ist in Abbildung 4.10 dargestellt.

Bevor die Initialisierungs– und die Beendigungsfunktion besprochen werden, muß zum besseren Verständnis auf die Art der Beschränkung der Objektvariablen (kurz: Beschränkungstyp) eingegangen werden. Der Beschränkungstyp sorgt dafür, daß die Objektvariablen den Definitionsbereich der Qualitätsfunktion einhalten. Er kann einen von drei Werten annehmen:

- `NO_CONSTRAINTS`:
  Keine Einschränkungen, die Objektvariablen können Werte aus dem gesamten Bereich des `double`–Typs annehmen und werden mit beliebigen Werten initialisiert.
- `USE_RANGE`:
  Der Wertebereich ist für jede Objektvariable durch eine Unter– und eine Obergrenze festgelegt. Die Objektvariablen werden auch in diesem Bereich initialisiert.
- `USE_CONSTRAINT_FUNCTION`:
  Mit diesem Beschränkungstyp können beliebige andere Beschränkungsarten realisiert werden. Für jede Objektvariable existiert dabei eine Beschränkungsfunktion, die den Wert der Variablen als Eingabe nimmt und den evtl. veränderten Wert zurückgibt.

Die Beschränkung der Objektvariablen wird jedesmal durchgeführt, wenn über eine *Set*–Funktion des *genus* (s. Abschnitt 4.2.1) die Objektvariablen eines Individuums gesetzt werden.

Die Initialisierungsfunktion wird einmal aufgerufen, bevor mit der Applikation eine Evolutionsstrategie gestartet wird. Sie bekommt als Eingabe die Anzahl der Objektvariablen, mit denen die Applikation rechnen soll und zwei Zeiger auf Felder, die sie je nach gewähltem Beschränkungstyp anlegen muß. Ist der Beschränkungstyp `NO_CONSTRAINTS`, so braucht sie nichts weiter zu tun. Bei `USE_RANGE` muß sie ein Feld mit Bereichen (ranges) anlegen, das für jede Objektvariable eine Unter– und Obergrenze enthält. Bei `USE_CONSTRAINT_FUNCTION` muß sie ebenfalls ein Feld anlegen, dieses muß für jede Objektvariable eine Beschränkungsfunktion (constraint function) enthalten. Unabhängig von den Feldern liefert die Initialisierungsfunktion einen Rückgabewert vom Typ `RetValVeeaType`, der je nachdem ob ein Fehler aufgetreten ist oder nicht, den Wert `OK` oder `FAIL` annimmt (s. Abschnitt 4.5.1).

Die Beendigungsfunktion hat die gleiche Gestalt wie die Initialisierungsfunktion (s. Abb. 4.10). Ihre einzige Aufgabe ist es, die Felder mit den Bereichen oder Beschränkungsfunktionen wieder freizugeben.

Der Dimensionenbereich besteht einfach aus der Angabe der kleinsten und der größten Zahl von Objektvariablen, für die die Applikation berechnet werden kann. Das *Traveling Salesman Problem (TSP)* beispielsweise, welches als Applikation *f10* implementiert ist, ist auf genau 100 Städte ausgelegt. Deshalb ist hier der Dimensionenbereich mit {100, 100} angegeben.

Die Anfangsschrittweite ist der Standardwert, der im Menü im Parameter `deltaIndividual` vorgegeben wird. Er ist nicht bindend und kann im Menü jederzeit abgeändert werden.

Wenn das Optimum der Applikation bekannt ist, so sollte dessen Qualitätswert angegeben werden, ansonsten irgendein Wert, der nicht weiter von Bedeutung ist. Der Optimumwert wird bei entsprechender Menüeinstellung (Parameter `convergence`) als Abbruchkriterium der Evolutionsstrategie benutzt.

Als letztes muß schließlich noch ein Name für die Applikation angegeben werden. Unter diesem kann sie dann im Menü angewählt werden.

Nachdem nun die Bedeutung der anzugebenden Parameter bekannt ist, ist hier noch einmal zusammengefaßt, wie eine neue Applikation in VEES integriert wird:

Zuerst muß das Applikationsmodul erstellt werden. In der '.c'–Datei sind die drei Applikationsfunktionen implementiert: Qualitätsfunktion, Initialisierungsfunktion und Beendigungsfunktion. In der '.h'–Datei der Applikation werden diese drei Funktionen exportiert. Außerdem stellt sie alle Parameter über `define`–Anweisungen der Applikationsschnittstelle zur Verfügung. Als Beispiel sind hier die Deklarationen der Applikation f1 gezeigt:

```
#define VEES_APPLICATION_F01_QUALITY           vees_applicationF01Quality
#define VEES_APPLICATION_F01_INIT_FUNCTION     vees_applicationF01InitFunction
#define VEES_APPLICATION_F01_EXIT_FUNCTION     vees_applicationF01ExitFunction
#define VEES_APPLICATION_F01_CONSTRAINT_SWITCH USE_RANGE
#define VEES_APPLICATION_F01_DIMENSION_RANGE   {1, 3}
#define VEES_APPLICATION_F01_DELTA_INDIVIDUAL  0.1024
#define VEES_APPLICATION_F01_CONVERGENCE_AT    78.644
#define VEES_APPLICATION_F01_NAME              "f1"
```

Wie man sieht, benutzt die Funktion *f1* als Beschränkungstyp `USE_RANGE`. Der Definitionsbereich dieser Applikation ist $[-5.12 \ldots + 5.12]$ für jede Objektvariable. Dieser wird bei der Initialisierung der Applikation in das Feld `rangesObjVar` aus Abbildung 4.10 eingetragen.

Diese Applikationsdaten müssen nun noch in die Datei `vees_ApplicationInterface.c` eingebracht werden. Dazu muß zuerst die Header–Datei der neuen Applikation importiert werden. Dies geschieht durch Ergänzen einer `include`–Direktive:

```
#include "vees_Application_F01.h"
```

Die so importierten Daten müssen in der Initialisierung der Variablen `applicationData` hinzugefügt werden. Diese Variable ist ein Feld, das für jede Applikation einen Eintrag enthält. Jeder Eintrag besteht aus der Liste der vorher beschriebenen acht Parameter.

```
applicationVeesType applicationData[] = {
  {
    .
    .
    .
  },
```

```
  {
    VEES_APPLICATION_F01_QUALITY,           /* the quality function        */
    VEES_APPLICATION_F01_INIT_FUNCTION,     /* the initialisation function */
    VEES_APPLICATION_F01_EXIT_FUNCTION,     /* the finishing function      */
    VEES_APPLICATION_F01_CONSTRAINT_SWITCH, /* kind of constraints         */
    VEES_APPLICATION_F01_DIMENSION_RANGE,   /* range of allowed dimensions */
    VEES_APPLICATION_F01_DELTA_INDIVIDUAL,  /* initial indiv. steplength   */
    VEES_APPLICATION_F01_CONVERGENCE_AT,    /* convergence value           */
    VEES_APPLICATION_F01_NAME               /* name of the application     */
  }
};
```

Eine programmiertechnisch erstrebenswertere Möglichkeit, die acht Parameter der Applikation zu exportieren, wäre gewesen, sie Konstanten zuzuweisen (C–Schlüsselwort `const`) und diese dann zu exportieren. Stattdessen mußte dies über `define`–Anweisungen geschehen, da in der Zuweisung an die Variable `applicationData` nur wirklich *konstante Werte* und keine *Konstantenbezeichner* vom Compiler akzeptiert werden.

Die Stelle, an der die Einträge hinzugefügt werden müssen, ist durch einen Kommentar deutlich gekennzeichnet. Zur Erstellung einer neuen Applikation kann entweder eine vorhandene Applikation kopiert werden, oder die Schablone `vees_Application_template` benutzt werden, die sich im Verzeichnis `veea/source/applications` befindet. Dadurch ist der Aufwand zur Einbindung einer neuen Applikation sehr klein.

Als letztes muß die Applikation noch in die Datei `veea/source/makefile.vees` eingetragen werden, damit sie mitübersetzt wird. Dazu wird einfach ein Eintrag mit dem Namen der Objektdatei der Applikation bei der Variablen `APP_OBJS` hinzugefügt:

```
APP_OBJS =      ${PATH_OBJ}/vees_Application_F01.o          \
                ${PATH_OBJ}/vees_Application_F02.o          \
                .
                .
                .
                ${PATH_OBJ}/vees_Application_F24.o          \
                ${PATH_OBJ}/vees_Application_Neu.o          \
```

Nach der erneuten Übersetzung von VEES erkennt die Applikationsschnittstelle selbständig, daß eine neue Applikation vorhanden ist. Sie erscheint dann automatisch im Menü und kann dort angewählt werden.

Kapitel 5

Vergleich mit anderen Implementierungen von Evolutionsalgorithmen

In diesem Kapitel werden die Ergebnisse von Leistungsmessungen am Programm VEES präsentiert. Sie werden mit den Leistungsdaten von anderen Programmen verglichen, die ebenfalls Implementierungen von Evolutionsalgorithmen sind. Als Vergleichsfunktionen werden drei Standard–Benchmarkfunktionen verwendet, die verschiedenartige Anforderungen an die Programme stellen.

5.1 Vorstellung der Testfunktionen

In den folgenden Abschnitten werden die verwendeten Testfunktionen vorgestellt. Diese Funktionen sind unter den gleichen Namen in VEES und allen anderen Programmen des EA–Projektes implementiert. Eine Aufstellung aller vorhandenen Funktionen befindet sich in Anhang D.

Die drei vorgestellten Funktionen wurden benutzt, weil sie sehr unterschiedliche Anforderungen an die Testprogramme stellen. Außerdem wurden mit diesen Funktionen schon umfangreiche Messungen mit den Programmen VEGA, MPES und MPGA durchgeführt. Diese Messungen sind in Abschnitt 5.3 zum Vergleich verwendet worden. Die Parametereinstellungen dieser Messungen sind in [Baumann 95] (VEGA) und [Görzig 95] (MPES) zu finden. Die genauen Einstellungen, mit denen man die angegebenen Ergebnisse mit VEES erzielt, sind in Anhang E aufgelistet. Bei den einzelnen Messungen wird darauf jeweils gesondert verwiesen.

5.1.1 Testfunktion f_1

Bei der Funktion f_1 handelt es sich um eine einfache quadratische Funktion. Die von ihr beschriebene Form wird auch „sphere model“ genannt.

$$f_1(\vec{x}) = \sum_{i=1}^{n} x_i^2 \quad , \quad \text{mit} \quad n = 3, -5.12 \leq x_i \leq 5.12$$

Sie besitzt ein globales Minimum bei: $f_1(\vec{o}) = f_1(0,\ldots,0) = 0$. Da VEES nur nach Maxima sucht, wurde die Funktion invertiert. In Abbildung 5.1 ist sie für den 2-dimensionalen Fall dargestellt, die Messungen wurden jedoch mit 3 Objektvariablen durchgeführt.

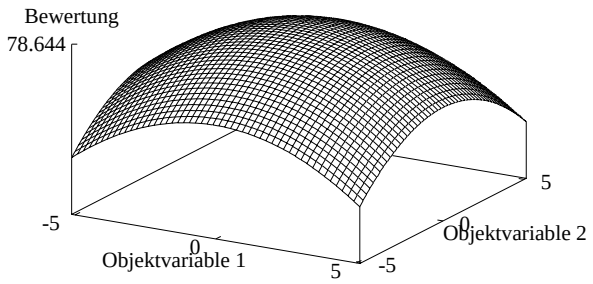


Abbildung 5.1: Die Testfunktion f_1 im 2-dimensionalen Fall (invertiert)

Da die Funktion f_1 sehr einfach ist, eignet sie sich vor allem zum Vergleich der Konvergenzgeschwindigkeit der Programme.

5.1.2 Testfunktion f_{10} (Traveling Salesman)

Bei der Testfunktion f_{10} handelt es sich um das Problem des Handlungsreisenden *(traveling salesman problem — TSP)* für 100 Städte. Die Problemstellung ist die folgende: es sind 100 Städte vorgegeben, die ein Handelsvertreter alle genau einmal besuchen muß. Gesucht ist dabei die kürzeste Rundreiseroute.

Beim Problem des Handlungsreisenden gibt es für 100 Städte $\frac{100!}{2 \cdot 100} \approx 4.67 \cdot 10^{155}$ mögliche Reiserouten. Es fällt in die Klasse der *NP–vollständigen* Probleme, d. h. daß die Berechnungszeit jedes bisher bekannten Lösungsalgorithmus, der garantiert die kürzeste Strecke findet, mit der Zahl n der Städte exponentiell ansteigt. Die Evolutionsstrategien garantieren nicht, daß sie die beste Lösung finden, sie versuchen nur möglichst gute Lösungen zu finden. Deshalb benötigen sie auch keine exponentiell ansteigende Rechenzeit.

Die hier verwendete konkrete Problemstellung wird auch *Krolak's 100 cities TSP* genannt. Es sind die Koordinaten von 100 Städten in der Ebene vorgegeben. Die Länge eine Reiseroute berechnet sich aus der Summe der Entfernungen der nacheinander besuchten Städte. Die Route endet in derselben Stadt, in der sie angefangen hat. Die kürzeste Route ist für dieses Problem bekannt und beträgt 21285 (s. Abbildung 5.3).

5.1.3 Testfunktion f_{22}

Die Testfunktion f_{22} besitzt sehr viele lokale Optima. Sie entsteht durch Überlagerung einer quadratischen Funktion mit Cosinus–Termen. In Abbildung 5.2 ist sie für den 2-dimensionalen

Fall dargestellt. Für die Messungen wurde der 10-dimensionale Fall verwendet. Hierbei besitzt sie neben dem globalen Optimum noch sehr viele Suboptima. Bei vieren davon ist der Qualitätswert nur 0.0074 von dem des globalen Optimums entfernt. Die Schwierigkeit bei dieser Funktion besteht also darin, nicht in einem der Suboptima hängenzubleiben.

$$f_{22}(\vec{x}) = \frac{1}{d}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos(\frac{x_i}{\sqrt{i}}) + 1 \quad \text{mit} \quad n = 10, d = 4000, -600 \leq x_i \leq 600$$

$$\text{Minimum: } f_{10}(\vec{o}) = f_{10}(0, \ldots, 0) = 0$$

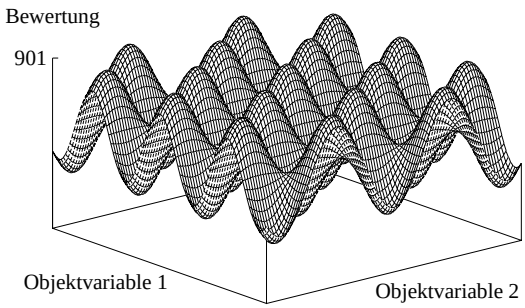


Abbildung 5.2: Die Testfunktion f_{22} im 2-dimensionalen Fall

5.2 Vorstellung der Vergleichsprogramme

In den folgenden Abschnitten werden kurz die Programme vorgestellt, mit denen die Vergleichsmessungen durchgeführt wurden.

5.2.1 Escapade

Das Programm ESC$^{\text{A}}$P$_{\text{A}}$DE stammt von Frank Hoffmeister und wurde an der Universität Dortmund entwickelt [Hoffmeister 90, Hoffmeister et al. 90]. Der Name steht dabei für *E*volution *S*trategies *cap*able of *ad*aptive evolution. Gemeint ist dabei nicht die Fähigkeit von Evolutionsstrategien, die Lösungen dem Lösungsraum anzupassen, sondern die Fähigkeit der Anpassung der Strategie selbst an die lokale Beschaffenheit des Qualitätsgebirges. Der Name zielt also auf die 'Meta'–Evolution der Strategievariablen.

Für die Messungen wurde ESC$^{\text{A}}$P$_{\text{A}}$DE Version 1.2 verwendet, welches Evolutionsstrategien vom Typ $(\mu/\rho^{+}, \lambda)^{\gamma}$ implementiert.

5.2.2 VEGA

Das Programm VEGA wurde von Roland Baumann geschrieben [Baumann 95]. Der Programmname steht für *VE*rteilte *G*enetische *A*lgorithmen. Es handelt sich dabei um eine Implementierung von Genetischen Algorithmen auf dem MIMD–Parallelrechner Intel Paragon, auf dem auch VEES implementiert ist. VEGA ist ebenso wie VEES auch auf einer UNIX–Workstation lauffähig. Der Vergleich zwischen VEES und VEGA ist sehr interessant im Hinblick auf die Unterschiede zwischen Genetischen Algorithmen und Evolutionsstrategien.

5.2.3 MPES

Das Programm MPES wurde von Steffen Görzig erstellt [Görzig 95]. Es handelt sich um eine Implementierung von *M*assiv *P*arallelen *E*volutions–*S*trategien auf dem SIMD–Parallelrechner MasPar MP-1. Es kann die beiden klassischen Arten von Evolutionsstrategien — My-Lambda– und geschachtelte My-Lambda–Strategien — ausführen. Außerdem sind noch speziell auf die SIMD–Architektur abgestimmte, massiv parallele Evolutionsstrategien implementiert. Sofern nichts anderes angegeben ist, wurden die aufgeführten Meßergebnisse mit den massiv parallelen Strategien erreicht. Ein interessanter Aspekt an dem Vergleich zwischen MPES und VEES ist der Unterschied zwischen der massiv parallelen und der verteilten Durchführung von Evolutionsstrategien.

5.3 Vergleichsmessungen

In den folgenden Abschnitten werden die Meßergebnisse vorgestellt. Dabei kommt es sowohl auf die absolute Laufzeit bis zur Konvergenz an, als auch auf die Rechenzeit pro Generation und Individuum. Die Zeit pro Generation berechnet sich als die insgesamt benötigte Zeit geteilt durch die Anzahl der Generationen. Die Zeit pro Individuum ist dabei nicht immer die reale Rechenzeit, sondern die Zeit, die effektiv für die Berechnung eines Individuums benötigt wurde. Bei nur einem Prozessor berechnet sie sich als die Zeit pro Generation, geteilt durch die Anzahl der Individuen. Wurden mehrere Prozessoren verwendet, so wurde zusätzlich durch deren Anzahl geteilt. Deswegen wirkt sich hier die parallele Durchführung aus.

Bei den Evolutionsstrategien wurde als Anzahl der berechneten Individuen die Zahl λ der Kindindividuen verwendet und nicht die Zahl μ der Elternindividuen. Der Grund dafür ist, daß in jeder Generation λ Individuen erzeugt, mutiert, bewertet und eventuell sortiert werden müssen (letzteres für die Selektion). Die Berechnungen werden also immer auf λ Individuen durchgeführt.

Es wurden die folgenden Bezeichnungen und Formeln zur Berechnung verwendet:

n	Zahl der Rechenknoten/Prozessoren	
ind	Zahl der Individuen	
gen	Zahl der Generationen	
t_{ges}	Gesamtrechenzeit	
t_{gen}	Zeit pro Generation	$t_{gen} = t_{ges}/gen$
t_{ind}	Zeit pro Individuum	$t_{ind} = t_{gen}/(n \cdot ind)$

Die aufgeführten Meßwerte sind in den meisten Fällen die Durchschnittswerte einer Meßreihe, die aus mehreren Einzelmessungen besteht. Bei jeder Einzelmessung wurde ein anderer Initialisierungswert für den Zufallszahlengenerator verwendet. Je nach Laufzeit der Messungen wurden zwischen einer und 20 Einzelmessungen gemacht. Die genauen Einstellungen jeder Reihe sind in Anhang E zu finden. Bei jeder Messung wird aber nocheinmal gesondert auf die verwendeten Einstellungen verwiesen.

5.3.1 Testfunktion f_1

Bei den Messungen mit VEES wurden jeweils die Durchschnittswerte einer Meßreihe aus 20 Einzelmessungen gebildet. Die Einstellungen von VEES bei diesen Messungen sind für die Reihe mit 16+1 Knoten in Tabelle E.1 und für 1+1 Knoten und Sparc 10 in Tabelle E.2 aufgeführt. Die Einstellungen von EsCAPADE befinden sich in Tabelle E.3.

$f_1(3)$	ind	gen	t_{ges} [s]	t_{gen} [ms]	t_{ind} [μs]
VEGA (16+1) Knoten	16×22	92	1,6	17,4	49,4
VEES (16+1) Knoten	16×22	33	0,906	27,5	78
VEGA (1+1) Knoten	1×350	69	6,2	85	257
VEES (1+1) Knoten	1×350	25	3,47	139	397
VEES (Sparc 10)	1×350	25	4,1	164	469
EsCAPADE (Sparc 10)	1×350	28	0,9	32,1	91,7
MPES[1] 16K PEs	16384×1	8	0,33	41	2,5

VEES erreicht das Optimum bei gleicher Knotenzahl grundsäzlich etwas schneller als VEGA. Der Grund dafür ist, daß VEES nur ca. ein Drittel der Generationen benötigt, um ans Ziel zu kommen. Die Berechnungszeit für eine Generation und für ein Individuum ist aber größer als bei VEGA. Der Geschwindigkeitsgewinn zwischen der Berechnung auf 16 Knoten und auf einem Knoten ist bei VEGA und VEES gleich und liegt ca. bei Faktor 3,8 für diese Funktion. EsCAPADE benötigt deutlich weniger Zeit als VEES und VEGA auf einem Knoten bzw. auf einer Sparc 10. Das Programm ist hier genauso schnell wie VEES auf 16 Knoten. MPES erzielt bei dieser Funktion das beste Ergebnis mit nur 0,33 Sekunden und benötigt auch nur 8 Generationen. Durch die gleichzeitige Berechnung von 16384 Individuen liegt die effektive Berechnungszeit für ein einzelnes Individuum weit unter der der anderen Programme.

Die Funktion f_1 besitzt nur ein Optimum, welches deshalb immer gefunden wird. Evolutionsstrategien benötigen hierfür weniger Generationen als Genetische Algorithmen, da sie über den Mechanismus der Schrittweitenanpassung verfügen. Dadurch können sie sich in den ersten Generationen mit großen Schritten dem Ziel nähern. Gegen Ende wird sich die Schrittweite immer mehr verkleinern.

EsCAPADE ist bei dieser Funktion deutlich schneller als VEES auf einem Prozessor. Die Gesamtausführungszeit bei dieser Funktion liegt allerdings nur im Sekundenbereich. Es kann durchaus sein, daß sich der Geschwindigkeitsunterschied bei anderen Funktionen und längeren

[1] mit My-Lambda– oder geschachtelten My-Lambda–Strategien das gleiche Ergebnis

Berechnungen verkleinert. Dies konnte allerdings nicht getestet werden, da in der verwendeten Version von ESCAPADE die Funktionen f_{10} und f_{22} nicht implementiert waren.

Wegen der sehr großen Anzahl an Individuen, ist MPES bei dieser Funktion am schnellsten. Es werden nur sehr wenige Generationen benötigt, da sich bei so vielen Individuen immer genügend gute Mutationen ergeben, die sich sehr genau in Richtung Optimum bewegen.

5.3.2 Testfunktion f_{10} (Traveling Salesman)

Bei den ersten Vergleichen mit der Funktion f_{10} kommt es auf die Rechengeschwindigkeit an. Deshalb wurden durchweg 1000 Generationen berechnet, ohne die Konvergenz zu beachten. Die Meßreihen bestanden bei 64+1 Knoten aus 3 Messungen, bei 36+1 Knoten aus 2 und bei 1+1 Knoten und Sparc 10 wegen der langen Rechenzeit aus einer Einzelmessung mit VEES. Die dafür verwendeten Einstellungen sind in den Tabellen E.4, E.5 und E.6 zu finden.

$f_{10}(100)$	ind	gen	min. Route	t_{ges} [s]	t_{gen} [s]	t_{ind} [μs]
VEGA (64+1) Knoten	64×256	1000	29498	1320	1,32	80,56
VEES (64+1) Knoten	64×256	1000	23935	2063	2,063	125,9
VEGA (36+1) Knoten	36×100	1000	32890	700	0,7	194
VEES (36+1) Knoten	36×100	1000	22293	614	0,614	170,6
VEES (1+1) Knoten	1×7200	1000	22583	33907	33,907	4709
VEES (Sparc 10)	1×7200	1000	22474	42623	42,623	5920
VEGA (Sparc 10)	1×7200	1000	34813	19802	19,8	2750

Die Zeiten liegen bei VEES auf 64+1 Knoten ca. 50% höher als bei VEGA. Auf 36+1 Knoten ist dagegen VEES etwas schneller. Auf einer Sparc 10 benötigt VEES ca. 25% länger als auf einem Knoten der Paragon. VEGA ist auf einem Knoten doppelt so schnell wie VEES. Ein Vergleich zwischen den Messungen auf verschiedenen Knotenzahlen ist hier nicht angebracht, da unterschiedliche Anzahlen von Individuen verwendet wurden.

Die von VEGA gefundenen Routen liegen im Bereich von ca. 29.500 bis fast 35.000, was deutlich länger ist, als die von VEES ermittelten, die im Bereich von 22.300 bis 24.000 liegen. Zwischen der besten Lösung von VEGA und der schlechtesten von VEES besteht ein Unterschied von ca. 5.500, was eine Verbesserung von ungefähr 20% gegenüber der Lösung von VEGA darstellt.

Je größere Populationen verwendet werden, desto langsamer wird VEES im Vergleich zu VEGA. Dies deutet darauf hin, daß die Verwaltung eines einzelnen Individuums in VEES grundsätzlich länger dauert. Wird dagegen eine kleinere Population verwendet, so überwiegt wahrscheinlich der Verwaltungsaufwand für Populationen in VEGA.

Die obigen Messungen dienten dem reinen Rechenzeitvergleich. Bei optimalen Einstellungen benötigen VEES und VEGA wesentlich weniger Zeit. Hinzu kommt, daß die Programme meist schon nach weniger als der Hälfte der 1000 Generationen auf eine bestimmte Lösung konvergiert waren. Bei Evolutionsstrategien werden üblicherweise in einer Population nur wenige hundert

Individuen verwendet, da sich ein Vorteil eher durch die Berechnung von mehr Generationen, als durch die Verwendung von mehr Individuen ergibt.

Deshalb sind in der nächsten Tabelle Vergleichswerte bei annähernd optimalen Einstellungen angegeben. Die Einstellungen der Messungen mit einem Knoten und mit 32+1 Knoten sind in den Tabellen E.7 und E.8 zu finden. Die Messungen mit 64+1 Knoten und 36+1 Knoten sind Einzelmessungen, die aus den Meßreihen der vorigen Tabelle herausgegriffen wurden.

$f_{10}(100)$	ind	gen	min. Route	t_{ges} [s]	t_{gen} [ms]	t_{ind} [μs]
VEES 1 Knoten	1×50	450	26140	148	329	6580
VEES (32+1) Knoten	32×30	430	23359	95	221	230
VEES (36+1) Knoten	36×100	320	22280	200	625	174
VEES (64+1) Knoten	64×256	360	22235	664	1844	112
MPES 16K PEs	16384×1	580	28497	621	1071	65,4

Schon mit nur einem Rechenknoten und nur 50 Individuen kommt VEES nach verhältnismäßig kurzer Zeit zu einem Ergebnis. Die gefundene Route ist kürzer, als die mit VEGA erreichten, selbst wenn dort 64 Knoten verwendet wurden. Durch Verwendung von mehr Prozessoren findet VEES immer bessere Lösungen. Die Anzahl der benötigten Generationen nimmt dabei ab. Eine Ausnahme hiervon bildet die Messung mit 64 Knoten. Hier wurden allerdings auch mehr Individuen verwendet, wodurch sie sich nicht mehr ganz mit den vorherigen vergleichen läßt. Die effektive Berechnungszeit für ein einzelnes Individuum verringert sich erwartungsgemäß bei der Steigerung der Knotenanzahl. Sie nimmt allerdings nicht exakt linear mit der Anzahl der Prozessoren zu, sondern verläuft wegen des hinzukommenden Kommunikationsaufwandes leicht sublinear.

MPES benötigt bei 16384 Individuen etwas weniger Zeit, als VEES mit der gleichen Anzahl Individuen auf 64 Prozessoren verteilt. Dabei berechnet MPES in dieser Zeit mehr Generationen, wodurch die Zeit pro Generation und Individuum unter der von VEES liegt. Die von VEES gefundene Route ist aber wesentlich kürzer, sogar bei Verwendung von nur einem Prozessor.

Für das Traveling Salesman Problem scheinen verteilte Evolutionsstrategien besser geeignet zu sein, als die massiv parallele Implementierung. Das Insel–Modell mit durch Migrationspfade verbundenen Populationen bringt hier scheinbar größere Fortschritte, als einzelne Individuen mit Nachbarschaftsverbindungen. Dies liegt vielleicht daran, daß ein Individuum in einer Population mehr direkte Nachbarindividuen besitzt, als bei massiv parallelen Strategien. Dadurch kann es mit Individuen rekombiniert werden, die sich stärker von ihm selbst unterscheiden. Durch den Individuenaustausch kommen neue Individuen Schubweise in die Population. Bei massiv parallelen Strategien dagegen bewegen sich die neuen Informationen stetiger fließend durch die Population.

Bei der Funktion f_{10} konnte ein sehr deutlicher Einfluß der Rekombination auf die erreichbaren Lösungen festgestellt werden. Ohne Rekombination wurden nur Routen gefunden, die um ca. 10.000 Längeneinheiten größer waren. Durch Verwendung von 2 Rekombinationspartnern wurden die gefundenen Routen deutlich kürzer. Die Steigerung auf 3 Rekombinanten brachte nur eine leichte, aber noch deutlich erkennbare Verbesserung. Darüber hinaus konnte keine wesentliche Steigerung mehr erzielt werden.

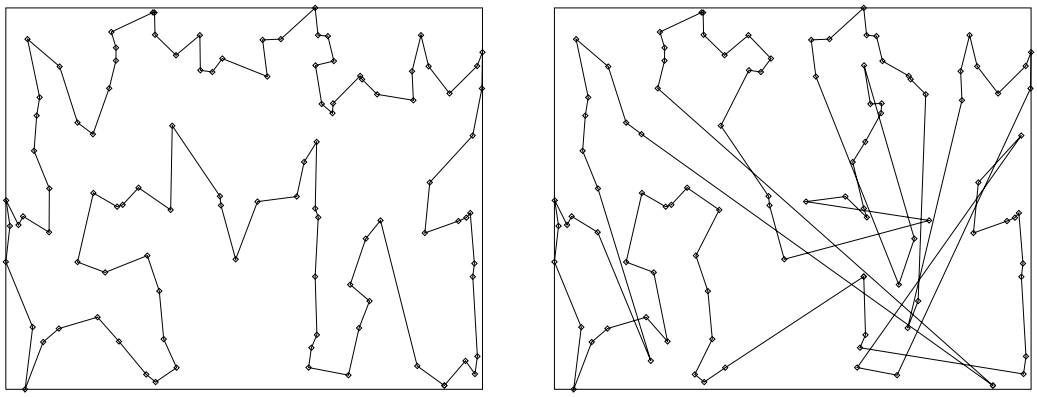


Abbildung 5.3: Die kürzeste Route beim TSP (links) mit Länge 21285 und die beste von VEES gefundene Lösung mit Länge 22235

Die kürzeste Route, die von VEES gefunden wurde, beträgt 22235 und trat in obiger Messung auf 64 Knoten auf. Sie ist nur 950 vom Optimum mit 21285 entfernt. In Abbildung 5.3 ist links die global kürzeste Route und rechts die von VEES gefundene dargestellt. Einige kurze Teilstrecken der Route von VEES stimmen mit der optimalen Lösung überein. Ansonsten weist sie noch einige große 'Zacken' auf, deren Beseitigung eine wesentliche Verbesserung bringen würde. Die verwendete Codierung des Problems in den Objektvariablen erklärt aber, warum die Route nicht weiter verbessert werden konnte:

Jede Objektvariable legt durch ihren Wert die relative Position einer Stadt zu den anderen Städten fest. Soll ein Individuum bewertet werden, so wird jeder Objektvariable die Nummer einer Stadt fest zugeordnet. Dann werden die Objektvariablen der Größe nach sortiert und dabei die Nummern der Städte mitgeführt. Die Reihenfolge, die die Nummern der Städte nach der Sortierung einnehmen, ist die Reihenfolge, in der die Städte besucht werden. Durch Verändern der Werte von Objektvariablen bewegen sich die Städte also relativ zu ihren Vorgängern bzw. Nachfolgern in der Reihenfolge.

Betrachtet man z. B. die quer über das Bild gehende, spitz zulaufende Verbindung der Stadt ganz unten rechts mit den beiden Städten oben links, so wird klar, warum sich hier keine Verbesserung ergeben kann: Um eine kürzere Strecke zu ergeben, müßte die Stadt an der Spitze erst sehr viele Vorgänger– bzw. Folgestädte 'überspringen'. Das ist aber in diesem Falle nicht mehr möglich, weil sich die Schrittweite bereits in einen Bereich angepaßt hat, in dem sie keine so großen Veränderungen mehr bewirken kann. Die Verwendung von Einzelschrittweiten schafft hier keine Abhilfe, da mit ihr solche Lösungen erst gar nicht erreicht werden.

5.3.3 Testfunktion f_{22}

Bei der Funktion f_{22} wurden mit VEES 10 Einzelmessungen mit einem Knoten und 20 Einzelmessungen mit 64 Knoten gemacht. Die verwendeten Einstellungen sind in den Tabellen E.9 (1 Knoten und Sparc 10) und E.10 (64 Knoten) zu finden.

$f_{22}(10)$	ind	gen	t_{ges} [s]	t_{gen} [ms]	t_{ind} [μs]
VEGA (Sparc 10)	1×3200	164	407,2	2483	776
VEES (Sparc 10)	1×3200	106	268,4	2532	791
VEGA (1+1) Knoten	1×3200	290	570,4	1967	615
VEES 1 Knoten	1×3200	106	300,3	2833	885
VEGA (64+1) Knoten	64×50	304	30,61	100,7	31,5
VEES (64+1) Knoten	64×50	108	13,6	126	39,4
MPES 16K[2] PEs	16384×2	51	10,02	196,5	6

Bei VEES konnte durch Aufteilen der 3200 Individuen von einem auf 64 Knoten ein Geschwindigkeitsgewinn um den Faktor 23 erzielt werden. Daß die Steigerung nicht gleich der Erhöhung der Zahl der Prozessoren ist, läßt sich durch den hinzukommenden Aufwand für Kommunikation und Synchronisation erklären. Bei nur 50 Individuen pro Prozessor fällt der Verwaltungsaufwand für die kleine Population auch mehr ins Gewicht, als bei 3200 Individuen.

VEES benötigt bei dieser Funktion nur wenig mehr Zeit zur Berechnung eines Individuums als VEGA. Allerdings waren in VEGA bis zu dreimal so viele Generationen bis zum Erreichen des Optimums notwendig, wodurch VEES effektiv ca. doppelt so schnell wie VEGA ist. Bei Verwendung von 64 Knoten braucht VEES doppelt so viele Generationen wie MPES, benutzt aber weniger Individuen. Deshalb ist VEES hier nur ein klein wenig langsamer. Durch den hohen Grad der Parallelität von MPES ist die effektive Zeit pro Individuum wesentlich kleiner.

Die Schwierigkeit dieser Funktion ist, wie bereits erwähnt, daß viele lokale Optima existieren, die es vom globalen zu unterscheiden gilt. Bei Verwendung vieler Knoten stellt dies kein großes Problem dar, normalerweise konvergieren alle Läufe zum globalen Optimum. Bei VEGA lag die Konvergenzrate mit wenigen Knoten teilweise bei nur 15%. Für VEES wurden Einstellungen verwendet, bei denen auf nur einem Knoten sämtliche Läufe das globale Optimum fanden. Das ist bei dieser Funktion eine sehr gute Konvergenzrate. Für MPES bereitet es ebenfalls kein Problem, sehr sicher das globale Optimum zu finden. Dies ist einerseits wieder mit der großen Zahl an Individuen zu begründen, die den Problemraum weitläufiger durchsuchen, andererseits durch die Populationsmutation, die bei den geschachtelten My-Lambda–Strategien angewendet wird. Sie verhindert das Verfangen in einem lokalen Optimum.

5.4 Zusammenfassung der Vergleiche

Die verteilten Evolutionsstrategien von VEES fallen, nach der absoluten Rechenzeit beurteilt, auf entsprechend vielen Rechenknoten durchaus in die gleiche Leistungsklasse, wie die massiv parallelen Evolutionsstrategien von MPES. MPES ist dabei architekturbedingt in der Lage, innerhalb derselben Zeit mehr Individuen als VEES zu berechnen. Allerdings konnte gezeigt werden, daß es bei Evolutionsstrategien meistens ausreichend ist, nur kleine Populationen zu verwenden.

Im Vergleich mit den verteilten Genetischen Algorithmen von VEGA konvergiert VEES nach weniger Generationen. Oftmals werden nur ungefähr ein Drittel der Generationen benötigt.

[2] mit geschachtelten My-Lambda–Strategien

Dadurch wird die etwas höhere Bearbeitungszeit pro Individuum mehr als ausgeglichen. Im Vergleich mit Genetischen Algorithmen sind bei den Evolutionsstrategien auch keine so großen Populationen notwendig, weil die Fortschrittsgeschwindigkeit weniger von der Diversität der Individuen abhängt, als bei Genetischen Algorithmen. Durch Verwendung kleinerer Populationen wird die Rechenzeit bei den Evolutionsstrategien weiter verringert. Beim Traveling Salesman Problem war VEES in der Lage, wesentlich bessere Lösungen als VEGA zu finden.

Mit dem Programm ESCAPADE konnte ein Vergleich nur bei der einfachen Funktion f_1 stattfinden, da die beiden anderen Testfunktionen in der verwendeten Version nicht implementiert waren. Hierbei hat sich gezeigt, daß mit ESCAPADE bessere Zeiten erzielt werden konnten als mit VEES auf einem Prozessor. Wurden mehr Prozessoren verwendet, so konnte mit VEES ungefähr die gleiche Zeit erreicht werden. Einen aussagekräftigeren Vergleich würden allerdings erst komplexere Testfunktionen erbringen.

Mit zunehmendem Parallelisierungsgrad konnte nicht nur die Gesamtrechenzeit deutlich verringert werden, sondern es erhöhte sich auch die Zahl der zum globalen Optimum konvergierten Läufe. Der wichtigste Gewinn durch Einsatz von parallelen Evolutionsstrategien hat sich beim Traveling Salesman Problem gezeigt. Hierbei steigerte sich die Qualität der gefundenen Lösungen deutlich durch den Einsatz von mehr Prozessoren.

Kapitel 6

Zusammenfassung und Ausblick

In dieser Diplomarbeit wurde das Programm VEES entworfen und auf dem MIMD–Parallelrechner Intel Paragon implementiert. Es ist aber nicht nur auf diesem Parallelrechner lauffähig, sondern kann auch von Einprozessor–Rechnern, wie z. B. Workstations, ausgeführt werden. In VEES sind parallele Evolutionsstrategien nach dem Insel–Modell mit Individuenaustausch und geschachtelte My-Lambda–Strategien verwirklicht. Es kann übersichtlich durch ein textuelles Menü bedient werden. Außerdem steht durch die Integration der Kommando–Sprache Tcl ein Satz leistungsfähiger Befehle zur Steuerung von VEES zur Verfügung. Im Programm sind bereits 24 Standard–Benchmarkfunktionen vorhanden, es kann aber leicht um neue Anwendungen erweitert werden.

Zur Beurteilung der Leistung der verteilten Evolutionsstrategien wurden zahlreiche Vergleichsmessungen mit anderen Implementierungen von Evolutionsalgorithmen durchgeführt. Bei Verwendung von entsprechend vielen Prozessoren konnten Rechenzeiten der gleichen Größenordnung wie bei massiv parallelen Evolutionsstrategien erreicht werden. Im Vergleich mit verteilten Genetischen Algorithmen erzielte VEES meist bessere Resultate. Durch die Messungen konnte gezeigt werden, daß die parallele Durchführung von Evolutionsstrategien nicht nur einen Geschwindigkeitsgewinn bringt. Es wird auch mit größerer Wahrscheinlichkeit das globale Optimum gefunden. Bei einem schwierigen Problem steigert sich auch die Qualität der erreichbaren Lösungen.

Für eine Weiterentwicklung des Programmes wäre denkbar, die VeES–Strategien durch eine Populationsmutation zu erweitern, um die Konvergenzrate zum globalen Optimum weiter zu erhöhen. Die geschachtelten My-Lambda–Strategien könnten derart erweitert werden, daß auf Populationsebene andere Variablen optimiert werden, als auf Individuenebene. Dies würde sich zur Lösung von Strukturierungsproblemen einsetzen lassen, die so nur schwer zugänglich sind. Der Mechanismus der separaten Schrittweitenregelung könnte zu korrelierten Schrittweiten ergänzt werden. Dadurch wird die völlig unabhängige Entwicklung der einzelnen Schrittweiten verhindert, was sich als Problem herausgestellt hat. Weiterhin können in einigen Teilen des Codes von VEES noch Optimierungen vorgenommen werden, wodurch die Ausführungsgeschwindigkeit gesteigert werden kann.

Anhang A

Erzeugung von Zufallszahlen

In Evolutionsstrategien werden an vielen Stellen Zufallszahlen gebraucht, z. B. bei der Auswahl von Eltern oder der Mutation von Objektvariablen. Für letztere werden Zufallszahlen einer speziellen Verteilung (Normalverteilung) benötigt, die oft auch bei natürlichen Prozessen anzutreffen ist. Da in Computern aber kein Zufallsgenerator eingebaut ist und Programme deterministisch ablaufen, behilft man sich mit sogenannten „Pseudo"–Zufallszahlen. Pseudo heißen sie deshalb, weil sie nicht wirklich zufällig sind, sondern aus einer Berechnungsvorschrift hervorgehen und somit jederzeit bei bekanntem Algorithmus und Anfangswert reproduziert werden können. Die Reproduzierbarkeit ist bei den Evolutionsstrategien aber ein Vorteil. Sie impliziert auch die Reproduzierbarkeit eines kompletten ES–Laufes, wenn er mit exakt denselben Parametern gestartet wird. Das bedeutet, daß auch genau dieselbe Lösung wieder gefunden wird. Somit können Meßergebnisse bei genauer Angabe der verwendeten Einstellungen nachvollzogen werden.

A.1 Gleichverteilte Zufallszahlen

Eine Zufallsvariable X heißt im Intervall $[a, b]$ gleichverteilt, wenn ihre Dichte in $[a, b]$ konstant und sonst 0 ist. Ihre Verteilungsfunktion hat die Gestalt A.1.

$$P(X < x) = F(x) = \frac{x - a}{b - a} \tag{A.1}$$

Ist X eine diskrete gleichverteilte Zufallsvariable, die n verschiedene Werte annehmen kann, so ist die Wahrscheinlichkeit, daß sie einen beliebigen Wert x_i annimmt $P(X = x_i) = \frac{1}{n}$.

Auf Computern gibt es viele verschiedene Verfahren, mit denen gleichverteilte „Pseudo–Zufallszahlen" erzeugt werden können. Einfache Verfahren haben zwar den Vorteil, daß sie schnell sind, aber oftmals haben sie eine relativ kleine Periode, nach der sich die Folge wiederholt. Außerdem neigen bestimmte Bitstellen dazu, sich sehr regelmäßig zu verändern. Ein Beispiel hierfür ist die Methode der linearen Kongruenz, bei der ein Nachfolgewert x_{i+1} in der Zufallsfolge aus dem direkten Vorgängerwert x_i nach Formel A.2 berechnet wird. Begonnen wird mit einem Startwert x_0, auch Initialisierungswert des Generators genannt.

$$x_{i+1} = (ax_i + b) \mod c \tag{A.2}$$

Mit dieser Formel können gleichverteilte Folgen von ganzen Zahlen erzeugt werden. Die Parameter a, b und c sind Konstanten, die die Folge beeinflussen. Der Bereich, in dem die erzeugten Zahlen x_i liegen, wird durch c bestimmt: $0 \leq x_i < c$. Die Konstanten a und b beeinflussen die Periode der Folge.

Durch günstige Wahl der Parameter und einige Ergänzungen können solche Generatoren aber wesentlich verbessert werden. Der in VEES verwendete Generator benutzt ein Verfahren, das in [Press et al. 88] vorgeschlagen wird. Dabei werden drei dieser einfachen Generatoren benutzt, um zusammen einen verbesserten zu bilden. Zwei Generatoren erzeugen jeweils den höherwertigen und niederwertigen Anteil einer Zahl, der dritte Generator wird dazu verwendet, eine Folge dieser Zahlen durcheinander zu mischen, damit sie in einer anderen Reihenfolge ausgegeben werden wie sie erzeugt wurden.

Aus den ganzzahligen Werten x_i lassen sich leicht reellwertige Zahlen z_i erzeugen:

$$z_i = \frac{x_i}{c} \qquad 0 \leq x_i < c \quad , \quad 0 \leq z_i < 1 \tag{A.3}$$

Diese liegen dann im normierten Bereich $[0, 1)$, der sich durch Anwenden von Formel A.4 in beliebige Bereiche $[r, s)$ umrechnen lassen.

$$z_i' = z_i(s - r) + r \tag{A.4}$$

A.2 Normalverteilte Zufallszahlen

Eine Zufallsgröße X heißt (μ, σ)–normalverteilt, wenn sie ihre Verteilungsfunktion die Gestalt A.5 hat. Dabei ist μ der Mittelwert und σ die Streuung oder Standardabweichung. σ^2 wird als Varianz bezeichnet.

$$P(x < X) = \int_{-\infty}^{x} f(t) \quad dt = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{(t-\mu)^2}{2\sigma^2}} dt \tag{A.5}$$

Normalverteilte Zufallsgrößen treten sehr häufig in der Natur auf, z. B. ist die Körpergröße von Menschen normalverteilt. Die Größe der meisten Menschen ist nahe einem Mittelwert. Entfernt man sich von diesem Mittelwert, so wird man immer weniger Menschen finden, die eine solche Größe haben.
Da Evolutionsstrategien aus der Natur nachgebildete Prozesse sind, treten auch hier normalverteilte Zufallsgrößen auf, z. B. bei der Mutation.

Die Dichtefunktion $f(x)$ (Formel A.6) der Normalverteilung ist die *Gausssche Glockenkurve* (Abbildung A.1). Sie hat ihr Maximum an der Stelle $x = \mu$ und besitzt Wendepunkte bei $x = \pm\sigma$. Die Fläche unter der Kurve beträgt 1.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{A.6}$$

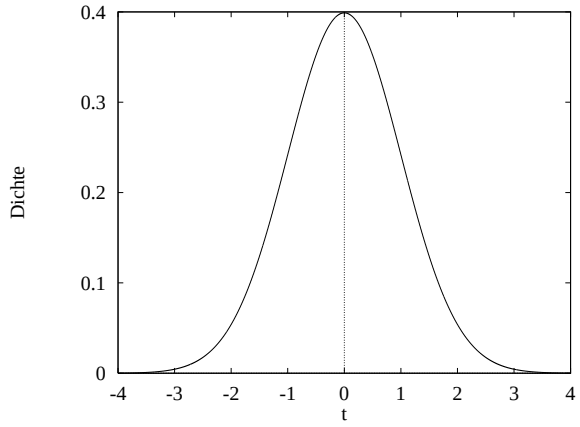


Abbildung A.1: Die Dichtefunktion der $N(0,1)$–Normalverteilung

Auch für normalverteilte Zufallsgrößen gibt es eine Normierung. Dafür werden die Werte $\mu = 0$ und $\sigma = 1$ verwendet. Die Zufallszahl heißt dann $N(0,1)$– verteilt. Eine $N(0,1)$–verteilte Zufallszahl Z läßt sich durch folgende Umformung in eine $N(\mu,\sigma)$–verteilte Zahl Z' umrechnen:

$$Z' = Z\sigma + \mu \tag{A.7}$$

Es existieren verschiedene Verfahren, um aus einer Folge von gleichverteilten Zufallszahlen eine Folge von normalverteilten Zahlen zu erzeugen [Knuth 69, Press et al. 88, Jansson 66]. Das effizienteste davon ist die *Box-Muller–Transformation*. Mit ihr können aus nur zwei $[0,1)$–gleichverteilten Zufallszahlen w_1 und w_2, mit Formel A.8 zwei $N(0,1)$–verteilte Zufallszahlen z_1 und z_2 erzeugt werden. Alle anderen Verfahren benötigen mehr gleichverteilte Zufallszahlen bei der Erzeugung, erfüllen allerdings auch spezielle stochastische Tests für Zufallszahlen. Da in Evolutionsstrategien sehr viele Zufallszahlen benötigt werden, eignet sich das schnelle Box-Muller–Verfahren hier besonders.

$$\begin{aligned} z_1 &= \sqrt{-2\ln w_1}\cos 2\pi w_2 \\ z_2 &= \sqrt{-2\ln w_1}\sin 2\pi w_2 \end{aligned} \tag{A.8}$$

In dieser Form treten aber noch die beiden trigonometrischen Funktionen Sinus und Cosinus auf. Diese sind auf dem Computer nur recht aufwendig zu berechnen. Durch eine Umwandlung von Sibuya und Marsaglia-Bray lassen sie sich aber durch schneller berechenbare Funktionen ersetzen:

$$\begin{aligned} z_1 &= y_1\sqrt{\frac{-2\ln(y_1^2+y_2^2)}{y_1^2+y_2^2}} \\ z_2 &= y_2\sqrt{\frac{-2\ln(y_1^2+y_2^2)}{y_1^2+y_2^2}} \\ y_1, y_2 &= [-1,1]\text{–gleichverteilte Zufallszahlen} \\ &\text{mit } y_1^2+y_2^2 \le 1 \quad \text{und} \quad y_1^2+y_2^2 \neq 0 \end{aligned} \tag{A.9}$$

Anhang B

Menü–Einträge

Hier folgt eine Auflistung sämtlicher Parameter, die in VEES vom Benutzer eingestellt werden können. Zu jedem Parameter wird der volle Name, die Abkürzung, der Typ (ganze Zahl, reele Zahl, Aufzählung usw.), der erlaubte Bereich, sinnvolle Werte und eine Beschreibung des Parameters aufgelistet. Da unter bestimmten Bedingungen manche Menüpunkte nicht aktiv sind, ist wenn nötig unter dem Stichwort 'Abhängigkeiten' angegeben, wann dieser Menüpunkt aktiv ist. Beispielsweise ist es nicht sinnvoll, wenn das Selektionsverfahren 'best' gewählt ist, den Ranking–Gradient anzugeben, der nur für das Selektionsverfahren 'ranking' nötig ist. Unter der textuellen Oberfläche erscheinen dann solche Menüpunkte nicht, wenn der Befehl 'm' für das Menü eingegeben wird. Unter der graphischen Oberfläche werden inaktive Menüpunkte grau dargestellt.

Alle Parameter sind in der textuellen und der graphischen Oberfläche unter exakt den gleichen Namen vorhanden.

Parameter zur Festlegung der Strategieart
In dieser Klasse existiert nur ein Menüpunkt, mit dem man die Strategieart einstellen kann. Je nach gewählter Strategieart wird die Klasse der VeES–Parameter oder die Klasse für geschachtelte My-Lambda–Strategien aktiviert. Da es sich hierbei um komplette Klassen handelt, wird diese Abhängigkeit bei den einzelnen Parametern dieser Klassen nicht mehr extra aufgeführt.

esStrategy	**esst**
Klasse:	Strategietyp
Typ:	Aufzählung
Werte:	`vees, nested_my_lambda`
Beschreibung:	Mit diesem Parameter kann die Art der ausgeführten Evolutionsstrategie bestimmt werden. Eine `vees`–Strategie ist nach dem Insel–Modell auf die Parallelarchitektur abgestimmt. Eine `nested_my_lambda`–Strategie ist eine geschachtelte Standard–Evolutionsstrategie.

VeES-Parameter
Diese sind nur verfügbar, wenn der Parameter `esStrategy` auf `vees` eingestellt ist. Alle Parameter dieser Klasse beginnen mit den beiden Buchstaben `ve`.

veMy	**vemi**
Klasse:	VeES–Parameter
Typ:	ganze Zahl
Werte:	$\mu \geq 1$
sinnvoller Bereich:	1 . . . max ca. 10000
Beschreibung:	Dies ist der Parameter μ in einer $(\mu/\rho^{+}, \lambda)^{\gamma}$ Evolutionsstrategie. Er bestimmt die Anzahl der Eltern–Individuen pro Prozessor.

veRho	**veri**
Klasse:	VeES–Parameter
Typ:	ganze Zahl
Werte:	$1 \leq \rho \leq \mu$
sinnvoller Bereich:	1 . . .10
Beschreibung:	Dies ist der Parameter ρ in einer $(\mu/\rho^{+}, \lambda)^{\gamma}$ Evolutionsstrategie. Er bestimmt die Anzahl der Individuen, die zu einem neuen Individuum rekombiniert werden.

veLambda	**veli**
Klasse:	VeES–Parameter
Typ:	ganze Zahl
Werte:	$\lambda \geq 1$ beim Strategietyp Plus (s. `veType`) $\lambda \geq \mu$ beim Strategietyp Komma
sinnvoller Bereich:	1 . . .10000
Beschreibung:	Dies ist der Parameter λ in einer $(\mu/\rho^{+}, \lambda)^{\gamma}$ Evolutionsstrategie. Er bestimmt die Anzahl der Kind–Individuen pro Prozessor.

veGamma(generations)	**vegi**
Klasse:	VeES–Parameter
Typ:	ganze Zahl
Werte:	$\gamma \geq 1$
Beschreibung:	Dies ist der Parameter γ in einer $(\mu/\rho^{+}, \lambda)^{\gamma}$ Evolutionsstrategie. Er bestimmt die Anzahl der zu berechnenden Generationen.

veType	**veti**
Klasse:	VeES–Parameter
Typ:	Aufzählung
Werte:	`plus, comma`
Beschreibung:	Selektions–Typ der Evolutionsstrategie.

veExchangeInterval	**veexi**
Klasse:	VeES–Parameter
Typ:	ganze Zahl
Werte:	≥ 1
Beschreibung:	Das Austauschintervall bestimmt, nach wievielen Generationen ein Individuenaustausch zwischen den Prozessoren stattfindet.

veExchangeRate	**veexr**
Klasse:	VeES–Parameter
Typ:	reelle Zahl
Werte:	$0 \leq$ veExchangeRate ≤ 1
Beschreibung:	Die Austauschrate bestimmt den relativen Anteil der Individuen, die ausgetauscht werden. Sie bezieht sich auf die Zahl `veMy` der Eltern–Individuen.

veExchangeTopology	**veext**
Klasse:	VeES–Parameter
Typ:	Aufzählung
Werte:	`ring, grid, xnet, hypercube, full-connected`
Beschreibung:	Die Austauschtopologie bestimmt, zwischen welchen Prozessoren Austauschverbindungen bestehen.

veDelta	**vedi**
Klasse:	VeES–Parameter
Typ:	reelle Zahl
Beschreibung:	Anfangswert der Mutationsschrittweite von Individuen. Wird beim Anwählen einer Applikation (`app`) vorbelegt.

veStepAdaption	**vesa**
Klasse:	VeES–Parameter
Typ:	Schalter
Werte:	`on, off`
Beschreibung:	Mutations-Schrittweiten-Regelung (MSR) an– oder ausschalten.

veStepControl	**vesc**
Klasse:	VeES–Parameter
Typ:	Aufzählung
Werte:	`uniform, separate`
Beschreibung:	Einzelne Schrittweite für ein ganzes Individuum oder getrennte Schrittweiten für jede Objektvariable.
Abhängigkeiten:	Nur verfügbar, wenn `veStepAdaption` auf `on`.

veAlpha	**veai**
Klasse:	VeES–Parameter
Typ:	reelle Zahl
Werte:	> 1.0
sinnvoller Bereich:	$1.0 \ldots 5.0$
Beschreibung:	Modifizierungsfaktor der Mutationsschrittweite.
Abhängigkeiten:	Nur verfügbar, wenn `veStepAdaption` auf `on`.

veRhoRecombObjVars	**veroi**
Klasse:	VeES–Parameter
Typ:	Aufzählung
Werte:	`dominant, intermediate, continuous`
Beschreibung:	Rekombinationsmethode für die Objektvariablen.
Abhängigkeiten:	Nur verfügbar, wenn `veRho` > 1.

veRhoRecombStratVars	**versi**
Klasse:	VeES–Parameter
Typ:	Aufzählung
Werte:	`dominant, intermediate`
Beschreibung:	Rekombinationsmethode für die Strategievariable(n).
Abhängigkeiten:	Nur verfügbar, wenn `veRho` > 1.

veSelectionMethod	**vesmi**
Klasse:	VeES–Parameter
Typ:	Aufzählung
Werte:	`best, ranking, roulette_wheel`
Beschreibung:	Selektionsmethode für Individuen.

veRankingGradient	**vergi**
Klasse:	VeES–Parameter
Typ:	reelle Zahl
Werte:	$0.0 \leq$ vergi ≤ 1.0
Beschreibung:	Ranking–Gradient.
Abhängigkeiten:	Nur verfügbar, bei Ranking–Selektion (`veSelectionMethod = ranking`)

Parameter für geschachtelte My-Lambda–Strategien
Die Parameter dieser Klasse sind nur bei geschachtelten My-Lambda–Strategien verfügbar (`esStrategy = nested_my_lambda`). Diese Abhängigkeit ist bei den einzelnen Parametern nicht mehr besonders vermerkt. Alle Parameter dieser Klasse beginnen mit den drei Buchstaben `nml`, was für *nested my-lambda* (geschachtelte My-Lambda–Strategie) steht. In den folgenden Parameterbeschreibungen wird für die Anzahl der Rechenprozessoren die Bezeichnung n_{proz} benutzt. Wenn VEES im Master-Slave–Modus arbeitet, ist dies die Anzahl der Prozessoren, auf denen VEES gestartet wurde, minus eins.

nmlMyPopulation	**nmlmp**
Klasse:	geschachtelte My-Lambda–Strategien
Typ:	ganze Zahl
Werte:	$1 \leq \mu' \leq n_{proz}$
Beschreibung:	Dies ist der Parameter μ' in einer $[\mu'/\rho'^{+}, \lambda'(\mu/\rho^{+}, \lambda)^{\gamma}]^{\gamma'}$ Evolutionsstrategie. Er bestimmt die Anzahl der Eltern–Populationen.

nmlRhoPopulation	**nmlrp**
Klasse:	geschachtelte My-Lambda–Strategien
Typ:	ganze Zahl
Werte:	$1 \leq \rho' \leq \mu'$
Beschreibung:	Dies ist der Parameter ρ' in einer $[\mu'/\rho'^{+}, \lambda'(\mu/\rho^{+}, \lambda)^{\gamma}]^{\gamma'}$ Evolutionsstrategie. Er bestimmt die Anzahl der zu rekombinierenden Populationen.

nmlLambdaPopulation	**nmllp**
Klasse:	geschachtelte My-Lambda–Strategien
Typ:	ganze Zahl
Werte:	$\lambda' = n_{proz}$ bei laufendem Programm nicht veränderbar
Beschreibung:	Dies ist der Parameter λ' in einer $[\mu'/\rho'^{+}, \lambda'(\mu/\rho^{+}, \lambda)^{\gamma}]^{\gamma'}$ Evolutionsstrategie. Er bestimmt die Anzahl der Kind–Populationen.

nmlGammaPopulation(cycles)	**nmlgp**
Klasse:	geschachtelte My-Lambda–Strategien
Typ:	ganze Zahl
Werte:	≥ 1
Beschreibung:	Dies ist der Parameter γ' in einer $[\mu'/\rho'^{+}, \lambda'(\mu/\rho^{+}, \lambda)^{\gamma}]^{\gamma'}$ Evolutionsstrategie. Er bestimmt die Anzahl der zu berechnenden Zyklen auf Populationsebene. Insgesamt werden $\gamma' \cdot \gamma$ Generationen berechnet.

nmlTypePopulation	**nmltp**
Klasse:	geschachtelte My-Lambda–Strategien
Typ:	Aufzählung
Werte:	`plus, comma`
Beschreibung:	Selektions–Typ auf Populationsebene.

nmlDeltaPopulation	**nmldp**
Klasse:	geschachtelte My-Lambda–Strategien
Typ:	reelle Zahl
Werte:	> 0
Beschreibung:	Anfangswert der Mutationsschrittweite von Populationen. Wird beim Anwählen einer Applikation (`app`) vorbelegt.

nmlStepAdaptionPopulation	**nmlsp**
Klasse:	geschachtelte My-Lambda–Strategien
Typ:	Schalter
Werte:	`on, off`
Beschreibung:	MSR für Populationen an– oder ausschalten.

nmlAlphaGenus	**nmlag**
Klasse:	geschachtelte My-Lambda–Strategien
Typ:	reelle Zahl
Werte:	> 1.0
Beschreibung:	Modifizierungsfaktor der Mutationsschrittweite von Populationen.
Abhängigkeiten:	Nur verfügbar, wenn `nmlStepAdaptionPopulation` auf `on`.

nmlRhoRecombPopulation	**nmlrmp**
Klasse:	geschachtelte My-Lambda–Strategien
Typ:	Aufzählung
Werte:	`dominant`
Beschreibung:	Rekombinationsmethode für Populationen.
Abhängigkeiten:	Nur verfügbar, wenn `nmlRhoPopulation` > 1.

nmlSelectionMethodPopulation	**nmlsmp**
Klasse:	geschachtelte My-Lambda–Strategien
Typ:	Aufzählung
Werte:	`best_average`
Beschreibung:	Selektionsmethode für Populationen; es werden diejenigen mit der besten durchschnittlichen Qualität ihrer Individuen gewählt.

Die Klasse der Parameter für geschachtelte My-Lambda–Strategien umfaßt noch weitere Parameter. Es handelt sich dabei um die Parameter auf Individuenebene. Sie unterscheiden sich meist nur im Präfix des Names von den entsprechenden Parametern bei VeES–Strategien, haben jedoch grundsätzlich dieselbe Bedeutung. Deshalb wird an dieser Stelle auf eine Wiederholung verzichtet. Die folgende Tabelle gibt kurz die restlichen Namen der Parameter für geschachtelte My-Lambda–Strategien, deren Abkürzung und den Namen des entsprechenden Parameters der VeES–Strategien, bei denen dann genauere Informationen nachgeschlagen werden kann. Schließlich ist noch eine Stichwort für die Bedeutung aufgeführt.

Abk.	Parametername	entspricht	Kurzbeschreibung
nmlmi	nmlMyIndividual	veMy	Anz. Elternind.
nmlri	nmlRhoIndividual	veRho	Rekombinationszahl
nmlli	nmlLambdaIndividual	veLambda	Anz. Kindindividuen
nmlgi	nmlGammaIndividual	veGamma	Anz. Generationen
nmlti	nmlTypeIndividual	veType	Selektionstyp
nmldi	nmlDeltaIndividual	veDelta	Anfangsschrittweite
nmlsi	nmlStepAdaptionIndividual	veStepAdaption	MSR an oder aus
nmlsc	nmlStepControl	veStepControl	Schrittweitenart
nmlap	nmlAlphaPopulation	veAlpha	Modifikationsfaktor
nmlroi	nmlRhoRecombObjVarsIndi	veRhoRecombObjVars	Rekombinationsmeth.
nmlrsi	nmlRhoRecombStratVarsIndi	veRhoRecombStratVars	Rekombinationsmeth.
nmlsmi	nmlSelectionMethodIndividual	veSelectionMethod	Selektionsmethode
nmlrgi	nmlRankingGradientIndividual	veRankingGradient	Ranking–Gradient

Parameter zur Rauschunterdrückung

Die folgende Klasse enthält Parameter, mit denen die Kappa-Ka–Rauschunterdrückung beeinflußt werden kann.

noiseTreatment	**noise**
Klasse:	Rauschunterdrückungs–Parameter
Typ:	Schalter
Werte:	`on, off`
Beschreibung:	Kappa-Ka Rauschbehandlung an– oder ausschalten.

noiseKappa	**kappa**
Klasse:	Rauschunterdrückungs–Parameter
Typ:	reelle Zahl
Werte:	> 1.0
Beschreibung:	Verstärkungsexponent für die Modifikation der Mutationsschrittweite.
Abhängigkeiten:	Nur verfügbar, wenn die Rauschunterdrückung an ist (`noise = on`).

noiseKa	**ka**
Klasse:	Rauschunterdrückungs–Parameter
Typ:	reelle Zahl
Werte:	> 1.0
Beschreibung:	Verstärkungsfaktor der Mutationsschrittweite
Abhängigkeiten:	Nur verfügbar, wenn die Rauschunterdrückung an ist (`noise = on`).

Applikationsparameter
Die Applikationsparameter umfassen das Einstellen der gewünschten Applikation, der Dimensionszahl des Problems und des Konvergenzkriteriums.

application	**app**
Klasse:	Applikations–Parameter
Typ:	Aufzählung
Werte:	`f1, f2, ..., f24`, u. a.
Beschreibung:	Wahl der Applikation (Bewertungsfunktion). Neu eingebundene Applikationen werden automatisch der Auswahlmenge hinzugefügt.

objectVariables	**objvar**
Klasse:	Applikations–Parameter
Typ:	ganze Zahl
Werte:	≥ 1
Beschreibung:	Anzahl der Objektvariablen pro Individuum (Dimension des Problems). Wird durch die Applikation vorbelegt und in ihren Grenzen festgelegt (`app`).

convergence	**con**
Klasse:	Applikations–Parameter
Typ:	Schalter
Werte:	`on, off`
Beschreibung:	An– und Ausschalten der Konvergenzprüfung. Ist sie ausgeschaltet, so stoppt die Strategie bei Erreichen der eingestellten Zahl von Generationen.

convergenceLimit	**conlim**
Klasse:	Applikations–Parameter
Typ:	reelle Zahl
Werte:	> 0
Beschreibung:	Gibt den (vermuteten) optimalen Qualitätswert der Applikation an.
Abhängigkeiten:	Nur verfügbar, wenn die Konvergenzprüfung an ist (`convergence`).

convergenceApproxGrade	**conapp**
Klasse:	Applikations–Parameter
Typ:	reelle Zahl
Werte:	0 < conapp <= 100
Beschreibung:	Grad der gewünschten Annäherung an das Optimum (`conlim`) in Prozent. Die Strategie wird abgebrochen, wenn die maximale Qualität eines Individuums größer oder gleich $\frac{\text{'conapp'}}{100} \cdot$ 'conlim' ist.
Abhängigkeiten:	Nur verfügbar, wenn die Konvergenzprüfung an ist (`convergence`).

applicationPopulation	**popapp**
Klasse:	Applikations–Parameter
Typ:	Aufzählung
Werte:	`best_average`
Beschreibung:	Bewertungsfunktion auf Populationsebene.
Abhängigkeiten:	Nur verfügbar, bei geschachtelten My-Lambda–Strategien.

Zufallszahlengenerator
Zur Durchführung von Versuchsreihen mit gleichen Einstellungen sollte der Initialisierungswert des Zufallszahlengenerators verändert werden.

seedRandomGenerator	**seed**
Klasse:	Zufallszahlengenerator
Typ:	ganze Zahl
Werte:	≥ 1
Beschreibung:	Initialisierungswert für den Zufallszahlengenerator.

Statistikparameter
Diese Klasse enthält alle Parameter zur Steuerung der textuellen und graphischen Ausgabe von Statistikdaten auf den Bildschirm und in Dateien.

statisticGnuPlot	**statg**
Klasse:	Statistik
Typ:	Schalter
Werte:	`on, off`
Beschreibung:	Graphische Statistikausgabe mit Gnuplot an oder aus.

statisticVerbose	**statv**
Klasse:	Statistik
Typ:	Schalter
Werte:	`on, off`
Beschreibung:	Statistikausgabe auf Bildschirm und in Dateien an oder aus.

statisticInterval	**stati**
Klasse:	Statistik
Typ:	ganze Zahl
Werte:	≥ 1
Beschreibung:	Ausgabeintervall in Generationen für alle Statistikdaten (Bildschirm, Dateien und Gnuplot).
Abhängigkeiten:	Nur verfügbar, wenn mindestens eine Statiskausgabe aktiviert ist (`statg on` oder `statv on`).

filenameProtocolIndividual	**filepi**
Klasse:	Statistik
Typ:	Dateiname
Beschreibung:	Dateiname der Protokoll– und Individuendatei. Es werden automatisch die beiden Endungen '.ESpro' und '.ESind' angehängt.
Abhängigkeiten:	Nur verfügbar, wenn Statistikausgabe an (`statv on`).

filenameCollection	**filec**
Klasse:	Statistik
Typ:	Dateiname
Beschreibung:	Dateiname der Sammeldatei. Es wird automatisch die Endung '.ESsum' angehängt.
Abhängigkeiten:	Nur verfügbar, wenn Statistikausgabe an (`statv on`).

Befehle

In dieser Klasse sind die drei Befehle zum Starten und Fortsetzen einer Evolutionsstrategie und die Kontrollbefehle für das Menü enthalten.

run	**r**
Klasse:	Steuerung
Typ:	Befehl
Beschreibung:	Start einer Evolutionsstrategie. Abbruch erfolgt bei Erreichen der Zahl der Generationen (`veGamma` bzw. `nmlGammaIndividual` mal `nmlGammaPopulation`) oder bei Erreichen des Konvergenzkriteriums.

step	**s**
Klasse:	Steuerung
Typ:	Befehl
Beschreibung:	Durchführen einer Schrittes (Generation) einer Evolutionsstrategie.

continueEs	**c**
Klasse:	Steuerung
Typ:	Befehl
Beschreibung:	Eine Evolutionsstrategie an der Stelle fortsetzen, an der sie beendet wurde. Vorher sollte die Zahl der Generationen (`veGamma` bzw. `nmlGammaPopulation`) vergrößert oder das Konvergenzkriterium verändert werden.

info	**i**
Klasse:	Steuerung
Typ:	Befehl
Beschreibung:	Ausgeben von Konfigurationsinformationen.

menu	**m**
Klasse:	Steuerung
Typ:	Befehl
Beschreibung:	Anzeigen des Menüs. Inaktive Parameter werden nicht angezeigt.

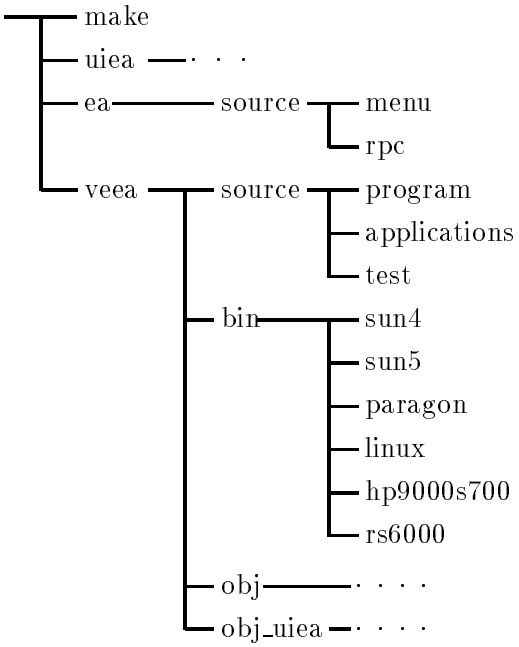
help	**h**
Klasse:	Steuerung
Typ:	Befehl
Beschreibung:	Ausgeben eines Hilfetextes über einen Parameter.

quit	**q**
Klasse:	Steuerung
Typ:	Befehl
Beschreibung:	Beenden des Programmes.

Anhang C

Dateien

In diesem Anhang wird ein vollständiger Überblick über alle Dateien von VEES gegeben.
Die Verzeichnisstruktur des Programms sieht folgendermaßen aus:



Auf der höchsten Ebene befinden sich die Verzeichnisse aller EA–Programme, der textuellen (`ea`) und graphischen (`uiea`) Oberfläche und des Compilierungsskriptes (`make`). Von den EA–Programmverzeichnissen ist hier allerdings nur `veea` dargestellt, welches die beiden Programme für Verteilte Evolutionäre Algorithmen — VEES und VEGA — enthält. Weiterhin gibt es die Verzeichnisse `mpea` für die beiden massiv parallelen Programme MPES und MPGA und das Verzeichnis `cnga` für Genetische Algorithmen auf dem Neurocomputer CNAPS. Diese sind aber unabhängig von VEES und werden zur Compilierung nicht benötigt.

Das Verzeichnis make

Hier befindet sich das Skript, welches die Compilierung steuert. Die Makefiles, die es aufruft, befinden sich allerdings im Programm–Verzeichnis `veea`.

Dateien im Verzeichnis make	
Name	**Beschreibung**
`build`	Skript zum compilieren eines beliebigen Programms aus dem EA–Projekt. Per Menü kann VEES wahlweise mit oder ohne graphische Oberfläche angewählt werden.
`build.doing`	Skript, welches von `build` eingelesen wird und die default–Werte für das `build`–Menü enthält. Kann vom Benutzer einfach an die eigenen Wünsche angepaßt werden.
`build.input`	Hilfsskript für die Benutzereingabe. Wird von `build` benutzt.

Das Verzeichnis ea

Hier befinden sich Dateien für die allgemeine Menüverwaltung aller EA–Programme und die Schnittstelle zur Graphischen Oberfläche, welche über einen RPC–Mechanismus[1] realisiert wurde. Das Verzeichnis `ea` ist deshalb weiter unterteilt in `ea/source/menu` und `ea/source/rpc`. Auf den Inhalt von `ea/source/rpc` wird hier nicht näher eingegangen, da diese Dateien nur benötigt werden, wenn VEES mit der graphischen Benutzeroberfläche bedient werden soll. In der Tabelle ist bei jeder Datei noch ein Kürzel für den Namen des Autors angegeben (G - Steffen Görzig, H - Alexander Hasel). Diese Dateien wurden nicht in der Arbeit VEES erstellt.

In den nun folgenden Tabellen sind bei Dateien, von denen eine C-Code–Datei (Endung '.c') und eine Header–Datei (Endung '.h') existiert, die Endungen weggelassen. Existiert nur eine der beiden Dateien, oder besitzt eine Datei eine andere Endung, so ist die Endung angegeben. Eine Ausnahme von dieser Regelung bilden nur die `makefile`– und `depend`–Dateien, welche genauso heißen, wie angegeben ist.

Dateien im Verzeichnis ea/source/menu		
Name	**Autor**	**Beschreibung**
`ea.h`	H	allgemeine Typdefinitionen
`ea_Debug`	H	Makros für Debugging der EA–Schnittstelle
`ea_Menu.x`	H	RPC-fähige Datentypen und Funktionen
`ea_MenuParam.h`	H	Definition der EA–Schnittstelle
`ea_MenuParam.c`	H	Funktionen zur Verwaltung der Menüstruktur
`ea_MenuParamClass.c`	H	Funktionen zur Verwaltung von Parameterklassen
`ea_MenuParamGetSet.c`	H	Get– und Set–Funktionen für die Menüparameter
`ea_MenuParamModuleInfo.c`	H	Ausgabe von Information über geladene Module
`ea_MenuParamLoad.c`	H	Laden von Parametereinstellungen aus Dateien
`ea_MenuParamSave.c`	H	Speichern von Parametereinstellungen in Dateien
`ea_MenuParamErrorCodes.h`	H	Definitionen für die Fehlerbehandlung
`ea_MenuRPCMessage`	H	Definition von Ausgabefiltern
`ea_info_msg`	H	Ausgabe von Informationstexten
`ea_error_msg`	G/H	Ausgabe von Fehlermeldungen
`ea_warning_msg`	G/H	Ausgabe von Warnungen
`ea_menu_tcl`	G	Implementierung der kompletten Textoberfläche mit Tcl–Anbindung

[1] *R*emote *P*rocedure *C*all

Das Verzeichnis uiea

Hier befinden sich die Dateien der graphischen Benutzeroberfläche (*U*ser *I*nterface für *E*volutionäre *A*lgorithmen). Dieses Verzeichnis ist auf die gleiche Art weiter unterteilt, wie das `veea`–Verzeichnis. Auf den Inhalt soll hier nicht weiter eingegangen werden. Die graphische Oberfläche wird in [Hasel 95] detailliert beschrieben.

Das Verzeichnis veea

Hier befinden sich die Programme VEES und VEGA. Das Verzeichnis ist weiter unterteilt in `source` für Quellcode–Dateien, `bin` für ausführbare Programme und `obj` und `obj_uiea` für Objektdateien. Bei der Übersetzung von VEES ohne Anbindung der graphischen Oberfläche, werden alle Objekt–Dateien im Verzeichnis `obj` abgelegt, bei Anbindung an die graphische Oberfläche im Verzeichnis `obj_uiea`. Das Binär- und die Objektverzeichnisse sind noch weiter unterteilt in die unterstützten Rechnerarchitekturen `sun4`, `sun5`, `paragon`, `linux`, `hp9000s700` und `rs6000`.

Im Verzeichnis `veea` selbst befinden sich Makefiles und depend–Dateien:

Dateien im Verzeichnis veea/source	
Name	**Beschreibung**
`makefile.vees`	Makefile für die Erstellung von VEES; wird von `build` aufgerufen
`makefile.vega`	Makefile für die Erstellung von VEGA; wird von `build` aufgerufen
`makefile.veea`	gemeinsame Definitionen für VEES und VEGA; wird von `makefile.vees` und `makefile.vega` benutzt.
`makefile.rules`	Compilierungs–Regeln; wird ebenfalls von `makefile.vees` und `makefile.vega` benutzt.
`depend.VEES`	Compilierungs–Abhängigkeiten für VEES
`depend.VEGA`	Compilierungs–Abhängigkeiten für VEGA

Dateien im Verzeichnis veea/source/program	
Name	**Beschreibung**
`vees_main.c`	Hauptprogramm von VEES
`vees_menu_init`	Initialisierung des Menüs; enthält alle Menüpunkte und ihre Anordnung
`vees_menu_init_get_set`	Set– und Get–Prozeduren für besondere Menü–Parameter
`vees_Parameters`	Schnittstelle zwischen Menü und und allen anderen Modulen, die auf die Menüparameter zugreifen müssen
`vees_Master`	Master–Modul; enthält die drei Befehle `run`, `step` und `continueEs`, die vom Menü aufgerufen werden
`vees_Slave`	Slave–Modul; empfängt Befehle vom Master–Modul und verzweigt in die spezielleren Slave–Module
`vees_VeesMaster`	spezielles Master–Modul für Evolutionsstrategien vom Typ VeES; enthält die Steuerung des Algorithmus
`vees_NmlMaster`	spezielles Master–Modul für Evolutionsstrategien vom Typ nested-my-lambda (geschachtelte My-Lambda–Strategien)

vees_VeesSlave	spezielles Slave–Modul für Evolutionsstrategien vom Typ VeES; enthält den Kern des Algorithmus für eine Generation
vees_NmlSlave	spezielles Slave–Modul für Evolutionsstrategien vom Typ nested-my-lambda (geschachtelte My-Lambda–Strategien)
vees_Individual	Verwaltung von Individuen
vees_Population	Verwaltung von Populationen; benutzt vees_Individual
vees_Genus	Verwaltung des kompletten genus–Datentyps (s. Abschnitt 4.2.1); benutzt vees_Population und vees_Individual
vees_GenusTypes	gemeinsame Typen von vees_Genus, vees_Population und vees_Individual
vees_Recombination	Rekombinationsmethoden
vees_Mutation	Mutation mit und ohne Kappa–Ka Rauschbehandlung
vees_Selection	alle Selektionsmethoden für Plus– und Komma–Strategien; Zurücksetzung der Kapp–Ka Rauschbehandlung
vees_IndividualExchange	Austausch von Individuen zwischen Prozessoren
vees_Statistics	Berechnung und Versenden von Statistikdaten
vees_TimeStatistics	Verwaltung der Timer (Uhren) zur Erfassung der Zeit–Statistiken
vees_Random	Zufallszahlengeneratoren für gleich– und normalverteilte Zufallszahlen
vees_ErrorHandling.h	Makros zur Fehlerbehandlung (s. 4.5.1) und sämtliche Fehlermeldungen
veea_defines.h	Makros für zur Bestimmung von Master– und Slave–Knoten; fehlende Prototypen von Systemmodulen; Fehlerbehandlungsmakros für VEGA
veea_types	allgemeine Typen und Konstanten für VEES und VEGA
veea_TraceHandling	Ausgabe von Trace–Informationen
veea_TimerHandling	Abstrakter Datentyp für Zeitmessungen
veea_MessageHandling	Alle Prozeduren zum Erstellen, Senden und Empfangen von Nachrichten verschiedenen Typs; Kapselt die darunterliegende *nx*–Bibliothek der Paragon
veea_GnuplotHandling	Prozeduren zur Erzeugung von Gnuplot–Schaubildern in Fenstern zur Visualisierung von Statistikdaten
veea_ErrorHandling	Ausgabe von Fehlermeldungen mit Angabe von Prozedurname und Knotennummer, auf dem der Fehler auftrat
vega_..	Quellcodedateien des Programms VEGA (s. [Baumann 95])

Dateien im Verzeichnis veea/source/applications	
Name	**Beschreibung**
`vees_ApplicationInterface`	Schnittstelle zwischen den Applikationen und dem restlichen Programm; hier müssen neue Applikationen eingetragen werden (s. Abschnitt 4.6)
`vees_Application_..`	Applikationsmodule von VEES; bisher existieren die Module F01 bis F24 (s. Anhang D)
`vees_Application_template`	Vorlage für die Erstellung einer neuen Applikation
`vega_...`	Applikations– und Hilfsmodule von VEGA (s. [Baumann 95])

Während der Entwicklung von VEES sind einige Programme entstanden, die gezielt bestimmte Module und Funktionen testen. Diese werden durch das `build`–Skript automatisch mitübersetzt und befinden sich danach im `bin`–Verzeichnis.

Dateien im Verzeichnis veea/source/test	
Name	**Beschreibung**
`bit2ind.c`	Test der Umwandlung eines Individuums in einen Bitstring; dies ist nötig, um die verzeigerte Struktur eines Individuums in einer Nachricht verschicken zu können
`individualSelection.c`	Test der Selektionsmethode `best` für Plus– und Komma–Strategien; auf der Kommandozeile muß eine Zahl zur Initialisierung des Zufallszahlengenerators angegeben werden;
`testGenus.c`	Test des Moduls `vees_Genus`, welches den zentralen Datentyp von VEES zur Durchführung einer Evolutionsstrategie implementiert; falls Fehler auftreten, so wird dies automatisch erkannt und gemeldet
`populationSort.c`	Test der Sortier–Prozedur für Populationen; auf der Kommandozeile muß eine Zahl zur Initialisierung des Zufallszahlengenerators angegeben werden; falls Fehler auftreten, so wird dies automatisch erkannt und gemeldet
`recombination.c`	Test der Rekombinationmethoden `dominant` und `intermediate` für separate und Einzelschrittweiten
`testIndOutput.c`	Test der Ausgabeprozeduren für Individuen
`test_gnuplot.c`	Test der Gnuplotausgabe und Skalierung der Achsen

`MsgTest.c`
`TopologyTest.c`
`alkan.c`
`geno2real.c`
`genotype.c`
`hello.c`
`individual.c`
`random.c`
`trace.c`
	Testprogramme für Module von VEGA (s. [Baumann 95])

Anhang D

Applikationen

Hier sind die Applikationen (Test–Funktionen) aufgelistet, die in VEES implementiert sind. Die Funktionen f_1 bis f_{24} wurden vom Programm *GENEsYs 1.0* [Bäck 92] übernommen.

1. Sphere model:

$$f_1(\vec{x}) = \sum_{i=1}^{n} x_i^2$$
$$n = 3$$
$$-5.12 \leq x_i \leq 5.12$$
$$\min(f_1) = f_1(0, \ldots, 0) = 0$$

2. Generalized Rosenbrock's function:

$$f_2(\vec{x}) = \sum_{i=1}^{n-1} (100 \cdot (x_{i+1} - x_i^2)^2 + (x_i - 1)^2)$$
$$n = 2$$
$$-5.12 \leq x_i \leq 5.12$$
$$\min(f_2) = f_2(1, \ldots, 1) = 0$$

3. Step function:

$$f_3(\vec{x}) = 6 \cdot n + \sum_{i=1}^{n} \lfloor x_i \rfloor$$
$$n = 5$$
$$-5.12 \leq x_i \leq 5.12$$
$$\min(f_3) = f_3([-5.12, -5), \ldots, [-5.12, -5)) = 0$$

4. Quartic function with noise:

$$f_4(\vec{x}) = \sum_{i=1}^{n} i x_i^4 + \text{gauss}(0,1)$$

$$n = 30$$

$$-1.28 \leq x_i \leq 1.28$$

$$\min(f_4) = f_4(0,\ldots,0) = 0$$

5. Shekel's foxholes:

$$\frac{1}{f_5(\vec{x})} = \frac{1}{K} + \sum_{j=1}^{25} \frac{1}{c_j + \sum_{i=1}^{2}(x_i - a_{ij})^6}$$

$$(a_{ij}) = \begin{pmatrix} -32 & -16 & 0 & 16 & 32 & -32 & \cdots & 0 & 16 & 32 \\ -32 & -32 & -32 & -32 & -32 & -16 & \cdots & 32 & 32 & 32 \end{pmatrix}$$

$$K = 500 \quad ; \quad f_5(a_{1j}, a_{2j}) \approx c_j = j$$

$$-65.536 \leq x_i \leq 65.536$$

$$\min(f_5) = f_5(-32,-32) \approx 1$$

6. Schwefel's function 1.2:

$$f_6(\vec{x}) = \sum_{i=1}^{n} \left(\sum_{j=1}^{i} x_j \right)^2 = x^{\mathrm{T}} \mathbf{A} x + b^{\mathrm{T}} x$$

$$n = 5$$

$$-65.536 \leq x_i \leq 65.536$$

$$\min(f_6) = f_6(0,\ldots,0) = 0$$

7. Generalized Rastrigin's function:

$$f_7(\vec{x}) = nA + \sum_{i=1}^{n} x_i^2 - A\cos(\omega x_i)$$

$$A = 10 \quad ; \quad \omega = 2\pi \quad ; \quad n = 5$$

$$-5.12 \leq x_i \leq 5.12$$

$$\min(f_7) = f_7(0,\ldots,0) = 0$$

8. Sphere model, changing environment:

$$f_8(\vec{x}(t)) = \begin{cases} \sum_{i=1}^{n} x_i^2(t) & : \; t \bmod a \text{ even} \\ \sum_{i=1}^{n} (x_i - b)^2 & : \; t \bmod a \text{ odd} \end{cases}$$

$$a = 250 \text{ generations} \quad ; \quad b = 4 \quad ; \quad n = 5$$

$$-5.12 \leq x_i \leq 5.12$$

$$\min(f_8) = \left\{ \begin{array}{ll} f_8(0,\ldots,0) & : \; t \bmod a \text{ even} \\ f_8(b,\ldots,b) & : \; t \bmod a \text{ odd} \end{array} \right\} = 0$$

9. Ackley's function:

$$f_9(\vec{x}) = -a \cdot \exp\left(-b\sqrt{\frac{1}{n}\cdot\sum_{i=1}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\cdot\sum_{i=1}^{n}\cos(c\cdot x_i)\right) + a + e$$

$$a = 20 \quad ; \quad b = 0.2 \quad ; \quad c = 2\pi \quad ; \quad n = 5$$

$$-32.768 \le x_i \le 32.768$$

$$\min(f_9) = f_9(0,\ldots,0) = 0$$

10. Krolak's 100 city TSP:

$$f_{10}(\vec{x}) = \sum_{i=1}^{n} d(c_{\pi(i \bmod n)}, c_{\pi((i+1) \bmod n)})$$

$$n = 100$$

$$\min(f_{10}) = 21285$$

11. Low autocorrelation binary sequences:

$$f_{11}(\vec{a}) = \sum_{k=1}^{l-1}\left(\sum_{i=1}^{l-k}\beta_i\beta_{i+k}\right)^2 \quad ; \quad \beta_i = \begin{cases} -1 & , \ \alpha_i = 0 \\ 1 & , \ \alpha_i = 1 \end{cases}$$

12. Hamming distance to 0^l:

$$f_{12}(\vec{a}) = \sum_{i=1}^{l}\alpha_i$$

$$\min(f_{12}) = f_{12}(0,\ldots,0) = 0$$

13. Weierstrass-Mandelbrot fractal function:

$$f_{13}(\vec{x}) = \sum_{i=1}^{n}\left(\frac{C(x_i)}{C(1)\cdot|x_i|^{2-D}} + x_i^2 - 1\right)$$

$$C(x) = \sum_{j=-\infty}^{\infty}\frac{1-\cos\left(b^j x\right)}{b^{(2-D)j}}$$

$$1 \le D \le 2 \quad ; \quad b > 1$$

$$-5.12 \le x_i \le 5.12$$

14. Fully deceptive function:

$$f_{14}(\vec{a}) = \begin{cases} 2^{-l} & , \ f_{12}(a) = 0 \\ (1 + f_{12}(a))/l & , \ 0 < f_{12}(a) < l \\ 0 & , \ f_{12}(a) = l \end{cases}$$

15. Weighted sphere model:

$$f_{15}(\vec{x}) = \sum_{i=1}^{n} i \cdot x_i^2$$

$$n = 3$$

$$-5.12 \leq x_i \leq 5.12$$

$$\min(f_{15}) = f_{15}(0, \ldots, 0) = 0$$

16. Fletcher and Powell:

$$f_{16}(\vec{x}) = \sum_{i=1}^{n} (A_i - B_i)^2$$

$$A_i = \sum_{j=1}^{n} (a_{ij} \sin \alpha_j + b_{ij} \cos \alpha_j)$$

$$B_i = \sum_{j=1}^{n} (a_{ij} \sin x_j + b_{ij} \cos x_j)$$

$$a_{ij}, b_{ij} \in [-100, 100]$$

$$\alpha_j \in [-\pi, \pi]$$

$$(a_{ij}) = \begin{pmatrix} -25.5668 & 96.1559 & -15.2661 & -23.247 & 1.15129 \\ -97.5908 & -82.9516 & 50.3364 & -96.552 & 29.4616 \\ 36.3096 & 54.3934 & 76.9586 & 64.3481 & 0.66188 \\ -58.7916 & 30.4881 & -22.9354 & 36.1643 & 62.2798 \\ 66.582 & -18.2818 & 2.30184 & -38.6698 & -80.3784 \end{pmatrix}$$

$$(b_{ij}) = \begin{pmatrix} -38.9994 & -49.1546 & -52.2403 & 66.4195 & -36.6538 \\ -6.49429 & -97.6865 & 39.6734 & -91.4542 & 17.2329 \\ -67.1681 & -42.8325 & -42.9307 & 36.9705 & -18.3463 \\ -53.1271 & -2.74637 & 27.6244 & -72.7537 & 46.1798 \\ -1.47036 & -95.2547 & 78.6214 & -54.3113 & 84.6737 \end{pmatrix}$$

$$(\alpha_j) = (1.98961, -1.39376, -2.03048, -0.621016, 3.12736)$$

17. Fletcher and Powell:

$$f_{17}(\vec{x}) = \sum_{i=1}^{n} (A_i - B_i)^2$$

$$A_i = \sum_{j=1}^{n} (a_{ij} \sin \alpha_j + b_{ij} \cos \alpha_j)$$

$$B_i = \sum_{j=1}^{n} (a_{ij} \sin x_j + b_{ij} \cos x_j)$$

$$a_{ij}, b_{ij} \in [-100, 100]$$

$$\alpha_j \in [-\pi, \pi]$$

18. Shekel-5:

$$f_{18}(\vec{x}) = -\sum_{i=1}^{m} \frac{1}{(\vec{x} - A(i))(\vec{x} - A(i))^T + c_i}$$

$$n = 4 \quad ; \quad m = 5$$

$$0 \leq x_i \leq 10$$

i	$A(i)$				c_i
1	4	4	4	4	0.1
2	1	1	1	1	0.2
3	8	8	8	8	0.2
4	6	6	6	6	0.4
5	3	7	3	7	0.4

$$\min(f_{18}) = f_{18}(0, \ldots, 0) = 0$$

19. Shekel-7:

$$f_{19}(\vec{x}) = -\sum_{i=1}^{m} \frac{1}{(\vec{x} - A(i))(\vec{x} - A(i))^T + c_i}$$

$$n = 4 \quad ; \quad m = 7$$

$$0 \leq x_i \leq 10$$

i	$A(i)$				c_i
1	4	4	4	4	0.1
2	1	1	1	1	0.2
3	8	8	8	8	0.2
4	6	6	6	6	0.4
5	3	7	3	7	0.4
6	2	9	2	9	0.6
7	5	5	3	3	0.3

$$\min(f_{19}) = f_{19}(0, \ldots, 0) = 0$$

20. Shekel-10:

$$f_{20}(\vec{x}) = -\sum_{i=1}^{m} \frac{1}{(\vec{x} - A(i))(\vec{x} - A(i))^T + c_i}$$

$$n = 4 \quad ; \quad m = 10$$

$$0 \leq x_i \leq 10$$

i	$A(i)$				c_i
1	4	4	4	4	0.1
2	1	1	1	1	0.2
3	8	8	8	8	0.2
4	6	6	6	6	0.4
5	3	7	3	7	0.4
6	2	9	2	9	0.6
7	5	5	3	3	0.3
8	8	1	8	1	0.7
9	6	2	6	2	0.5
10	7	3.6	7	3.6	0.5

$$\min(f_{20}) = f_{20}(0, \ldots, 0) = 0$$

21. Griewank:

$$f_{21}(\vec{x}) = \frac{1}{d}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

$$d = 200 \quad ; \quad n = 2$$

$$-100.0 \leq x_i \leq 100.0$$

$$\min(f_{21}) = f_{21}(0, \ldots, 0) = 0$$

22. Griewank:

$$f_{22}(\vec{x}) = \frac{1}{d}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

$$d = 4000 \quad ; \quad n = 10$$

$$-600.0 \leq x_i \leq 600.0$$

$$\min(f_{22}) = f_{22}(0, \ldots, 0) = 0$$

23. Galar:

$$f_{23}(\vec{x}) = (\exp(-5x_1^2) + 2\exp(-5(1-x_1)^2)) \cdot \exp\left(-5\sum_{i=2}^{n} x_i^2\right)$$

$$n = 5$$

$$-5.0 \leq x_i \leq 5.0$$

24. Kowalik:

$$f_{24}(\vec{x}) = \sum_{i=1}^{11}\left(a_i - \frac{x_1(b_i^2 + b_i x_2)}{b_i^2 + b_i x_3 + x_4}\right)^2$$

$$n = 4$$

$$-5.0 \leq x_i \leq 5.0$$

i	a_i	b_i^{-1}
1	0.1957	0.25
2	0.1947	0.5
3	0.1735	1
4	0.1600	2
5	0.0844	4
6	0.0627	6
7	0.0456	8
8	0.0342	10
9	0.0323	12
10	0.0235	14
11	0.0246	16

$$\min(f_{24}) \approx f_{24}(0.1928, 0.1908, 0.1231, 0.1358) \approx 0.0003075$$

Anhang E

Parametereinstellungen der Messungen

Hier sind die Einstellungen von VEES aufgelistet, die für die Messungen in Kapitel 5 verwendet wurden. In den Tabellen wurden die Abkürzungen der Parameter benutzt. Nach dem Parameterwert folgt, bedingt durch die Tcl–Syntax, jeweils ein Semikolon, das durch ein Leerzeichen vom vorhergehenden und nachfolgenden Parameter getrennt ist. Die Befehle können so direkt hintereinander eingegeben werden. Für das Programm ESCAPADE sind die Kommandozeilenparameter angegeben, mit denen das Programm gestartet wurde.

esst vees ;	vesa on ;	veexr 0.1 ;	conapp 100 ;
vemi 7 ;	vesc uniform ;	veext full-connected ;	seed 1 ;
veri 2 ;	veai 2.0 ;	noise off ;	statg off ;
veli 22 ;	veroi intermediate ;	app f1 ;	statv on ;
vegi 100 ;	versi dominant ;	objvar 3 ;	stati 100 ;
veti comma ;	vesmi best ;	con on ;	
vedi 0.1024 ;	veexi 5 ;	conlim 78.644 ;	

Tabelle E.1: Einstellungen für f_1 bei 16+1 Knoten

esst vees ;	vedi 0.1024 ;	vesmi best ;	conapp 100 ;
vemi 95 ;	vesa on ;	noise off ;	seed 1 ;
veri 2 ;	vesc uniform ;	app f1 ;	statg off ;
veli 350 ;	veai 2.0 ;	objvar 3 ;	statv on ;
vegi 100 ;	veroi intermediate ;	con on ;	stati 100 ;
veti comma ;	versi dominant ;	conlim 78.644 ;	

Tabelle E.2: Einstellungen für f_1 bei 1+1 Knoten und Sparc 10

-K 1	-M 0	-r -1	-C 5e-16	-V 1
-E 95	-g 300	-X 1	-v 1	-n 1
-O 350	-Q 2	-Y 0.1	-f f1	
-R 322	-T 15	-x 3	-k 1	

Tabelle E.3: Einstellungen für f_1 mit ESCAPADE

esst vees ;	vesa on ;	veexr 0.05 ;	con off ;
vemi 70 ;	vesc uniform ;	veext hypercube ;	seed 1 ;
veri 3 ;	veai 1.1 ;	noise on ;	statg off ;
veli 256 ;	veroi dominant ;	kappa 2.0 ;	statv on ;
vegi 1000 ;	versi dominant ;	ka 2.0 ;	stati 10 ;
veti comma ;	vesmi best ;	app f10 ;	
vedi 0.3 ;	veexi 5 ;	objvar 100 ;	

Tabelle E.4: Einstellungen für f_{10} bei 64+1 Knoten

esst vees ;	vesa on ;	veexr 0.15 ;	con off ;
vemi 27 ;	vesc uniform ;	veext grid ;	seed 5 ;
veri 3 ;	veai 1.07 ;	noise on ;	statg off ;
veli 100 ;	veroi dominant ;	kappa 2.0 ;	statv on ;
vegi 1000 ;	versi intermediate ;	ka 2.0 ;	stati 10 ;
veti comma ;	vesmi best ;	app f10 ;	
vedi 0.5 ;	veexi 5 ;	objvar 100 ;	

Tabelle E.5: Einstellungen für f_{10} bei 36+1 Knoten

esst vees ;	vedi 0.5 ;	vesmi best ;	con off ;
vemi 1944 ;	vesa on ;	noise on ;	seed 218 ;
veri 3 ;	vesc uniform ;	kappa 2.0 ;	statg off ;
veli 7200 ;	veai 1.07 ;	ka 2.0 ;	statv on ;
vegi 1000 ;	veroi dominant ;	app f10 ;	stati 50 ;
veti comma ;	versi intermediate ;	objvar 100 ;	

Tabelle E.6: Einstellungen für f_{10} bei 1+1 Knoten und Sparc 10

```
esst vees ;     vedi 0.5 ;              vesmi best ;            con off ;
vemi 14 ;       vesa on ;               noise on ;              seed 1 ;
veri 3 ;        vesc uniform ;          kappa 2.0 ;             statg off ;
veli 50 ;       veai 1.07 ;             ka 2.0 ;                statv on ;
vegi 1000 ;     veroi dominant ;        app f10 ;               stati 10 ;
veti comma ;    versi intermediate ;    objvar 100 ;
```

Tabelle E.7: Einstellungen für f_{10} bei 1 Knoten

```
esst vees ;     vesa on ;               veexr 0.1 ;             con off ;
vemi 8 ;        vesc uniform ;          veext grid ;            seed 1 ;
veri 3 ;        veai 1.07 ;             noise on ;              statg off ;
veli 30 ;       veroi dominant ;        kappa 2.0 ;             statv on ;
vegi 600 ;      versi intermediate ;    ka 2.0 ;                stati 10 ;
veti comma ;    vesmi best ;            app f10 ;
vedi 0.5 ;      veexi 5 ;               objvar 100 ;
```

Tabelle E.8: Einstellungen für f_{10} bei 32+1 Knoten

```
esst vees ;     vedi 6.0 ;              vesmi best ;            conapp 100 ;
vemi 864 ;      vesa on ;               noise off ;             seed 1 ;
veri 2 ;        vesc uniform ;          app f22 ;               statg off ;
veli 3200 ;     veai 2.5 ;              objvar 10 ;             statv on ;
vegi 300 ;      veroi dominant ;        con on ;                stati 50 ;
veti comma ;    versi intermediate ;    conlim 901 ;
```

Tabelle E.9: Einstellungen für f_{22} bei 1 Knoten und Sparc 10

```
esst vees ;     vesa on ;               veexr 0.1 ;             conapp 100 ;
vemi 14 ;       vesc uniform ;          veext full-connected ;  seed 1 ;
veri 2 ;        veai 2.5 ;              noise off ;             statg off ;
veli 50 ;       veroi dominant ;        app f22 ;               statv on ;
vegi 300 ;      versi intermediate ;    objvar 10 ;             stati 1 ;
veti comma ;    vesmi best ;            con on ;
vedi 6.0 ;      veexi 10 ;              conlim 901 ;
```

Tabelle E.10: Einstellungen für f_{22} bei 64+1 Knoten

Literaturverzeichnis

[Bäck 92] Thomas Bäck. *A User's Guide to GENEsYs 1.0.* Technical report, University of Dortmund, Department of Computer Science, System Analysis Research Group, 1992.

[Baumann 95] Roland Baumann. *Verteilte Genetische Algorithmen auf dem MIMD–Parallelrechner Intel Paragon.* Diplomarbeit Nr. 1227, Universität Stuttgart, IPVR, 1995.

[Ebner 95] Marc Ebner. *Parallele Genetische Algorithmen auf einem Neurocomputer.* Studienarbeit Nr. 1478, Universität Stuttgart, IPVR, 1995.

[Flynn 66] M. J. Flynn. *Very High Speed Computing Systems.* In *Proceedings of the IEEE*, Band 54, Seiten 1901–1909, 1966.

[Goldberg 89] David E. Goldberg. *Genetic algorithms in search, optimization and machine learning.* Addison–Wesley, 1989.

[Gray 53] F. Gray. *Pulse Code Communication.* U.S. Patent 2 632 058, 17. März 1953.

[Görzig 95] Steffen Görzig. *Massiv Parallele Evolutionsstrategien auf dem Parallelrechner MasPar MP-1.* Diplomarbeit Nr. 1291, Universität Stuttgart, IPVR, 1995.

[Hasel 95] Alexander Hasel. *Eine graphische Oberfläche für Parallele Genetische Algorithmen und Evolutionsstrategien.* Diplomarbeit Nr. 1301, Universität Stuttgart, IPVR, 1995.

[Hoffmeister 90] Frank Hoffmeister. *The User's Guide to* EscApade *1.0 — A Runtime Environment for Evolution Strategies.* University of Dortmund, Department of Computer Science XI, P.O.Box 500 500, D-4600 Dortmund 50, Germany, November 1990.

[Hoffmeister et al. 90] Frank Hoffmeister, Hans-Paul Schwefel. KORR *2.1 — An Implementation of a* (μ^+, λ) *– Evolution Strategy.* University of Dortmund, Department of Computer Science XI, P.O.Box 500 500, D-4600 Dortmund 50, Germany, November 1990.

[Holland 75] John H. Holland. *Adaption in Neural an Artificial Systems.* Ann Arbor The University of Michigan Press, 1975.

[Holland 93] John H. Holland. *Adaption in natural and arificial systems; an introduction analysis with applications to biology, control and artificial intelligence*. MIT Press, 1993.

[Hummler 95] Alex Hummler. *Massiv parallele Genetische Algorithmen auf dem Parallelrechner MasPar MP-1*. Diplomarbeit Nr. 1240, Universität Stuttgart, IPVR, 1995.

[Int94] Intel Corporation. *Intel Supercomputer Systems Division: „Paragon^{TM} User's Guide“*, June 1994. Order Number 312489-003.

[Jansson 66] Birger Jansson. *Random Number Generators*. Victor Pettersons Bokindustri Aktiebolag, 1966.

[Knuth 69] Donald E. Knuth. *The art of computer programming*. Seminumerical algorithms, Volume 2. Addison–Wesley, Massachusetts, California, London, Ontario, 1969.

[Ousterhout 95] John K. Ousterhout. *Tcl und Tk. Entwicklung grafischer Benutzerschnittstellen für das X Window System*. Professional Computing. Addison–Wesley, 1995.

[Press et al. 88] William H. Press, Brian P. Flannery, Saul A. Teukolsky, William T. Vetterling. *Numerical recipes in C*. The Art of Scientific computing. Cambridge university press, Cambridge, New York, Port Chester, Melbourne, Sydney, 1988.

[Rechenberg 73] Ingo Rechenberg. *Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Doktorarbeit, TU Berlin, F. f. Maschinenwesen, Oktober 1973. Ersch. auch in: Schriften zur Informatik 1971.

[Rechenberg 94] Ingo Rechenberg. *Evolutionsstrategie '94*, Band 1 von *Werkstatt Bionik und Evolutionstechnik*. frommann–holzboog, Stuttgart, 1994.

[Schwefel 95] Hans-Paul Schwefel. *Evolution and optimum seeking*. Sixth-Generation Computer Technology Series. John Wiley & Sons, INC., 1995.

[Zell et al. 95a] Andreas Zell, Roland Baumann. *Distributed Genetic Algorithms on the Intel Paragon, a large MIMD Computer*. Universität Stuttgart, IPVR, 1995.

[Zell et al. 95b] Andreas Zell, Alex Hummler. *Massively Parallel Genetic Algorithms on the SIMD–Computer MasPar MP-1*. Universität Stuttgart, IPVR, 1995.