Zwischenbericht der Projektgruppe Evolutinäre Algorithmen

M. Großmann
D. Ivancan
A. Leonhardi
T. Schmidt

1. August 1996

Prof. Dr. Volker Claus
Abteilung Formale Konzepte
Institut für Informatik
Universität Stuttgart

Breitwiesenstr. 20-22
D-70565 Stuttgart

Telefon:

0711-7816-300 (Prof. Dr. V. Claus)
0711-7816-301 (Sekretariat)
0711-7816-330 (FAX)

E-Mail: claus@informatik.uni-stuttgart.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Die Projektgruppe

Das Studium der Informatik vermittelt dem Studenten zwar einen großen Teil des nötigen Fachwissens, jedoch stellt das Berufsleben noch weitere Anforderungen an den Informatiker. Teamfähigkeit und Erfahrung spielen gerade bei der Mitarbeit an großen Software–Projekten eine wichtige Rolle. Hier verfolgt die Idee der Projektgruppe folgende Ausbildungsziele:

- Arbeiten im Team
- Analyse von Problemen, Strukturierung von Lösungen und gemeinsamer Entwurf geeigneter Systeme
- Selbstständige Erarbeitung von Lösungsvorschlägen und deren Vorstellung und Verteidigung in einer Gruppe
- Übernahme von Verantwortung für die Lösung von Teilaufgaben und die Erstellung von Modulen
- Mitwirkung an einer umfassenden Dokumentation
- Erstellen eines Software–Produktes, das ein Einzelner innerhalb des vorgegebenen Zeitraumes unmöglich bewältigen kann
- Projekt–Planung und Kosten/Nutzen–Analyse
- Einsatz von Werkzeugen
- Persönlichkeitsbildung (Übernahme von Verantwortung, Selbstvertrauen, Verläßlichkeit, Rücksichtnahme, Durchsetzungsfähigkeit usw.)

An der Projektgruppe nehmen in der Regel acht bis zwölf Studierende des Hauptstudiums teil. Sie erarbeiten im Laufe eines Jahres ein Software–Produkt, welches einem Zeitaufwand von mehreren Personenjahren entspricht. Hierbei sollen sämtliche Phasen eines Software–Lifecycles — von der Planung bis zur Wartung — durchlaufen werden, was in anderen Lehrveranstaltungen nicht üblich ist. Bei Software– und Fachpraktika wird zumeist eine gegebene, genau festgelegte Aufgabenstellung in ein Programm umgesetzt.

Eine Projektgruppe vereinigt die Lehrveranstaltungsformen „Hauptseminar“ (2 SWS), „Fachpraktikum“ (4 SWS) und „Studienarbeit“ (10 SWS) in sich. Demzufolge ist eine Projektgruppe mit 16 SWS einzustufen.

Der Ablauf einer Projektgruppe folgt meist folgendem Schema: Seminar–, Planungs–, Entwurfs–, Implementierungs–, Integrations–, Experimentier– und Schlußphase. Diese Phasen werden im folgenden genauer erläutert.

Seminarphase: Die Themenstellung wird gründlich analysiert. Dazu werden von den Mitgliedern Originalpublikationen durchgearbeitet und die Ergebnisse vorgetragen. Ergebnisse dieser Phase sind viel Wissen, je eine Vortragsausarbeitung und eine zusammenfassende Darstellung der Literaturauswertung.

Planungsphase: Die Projektgruppe analysiert den Problembereich, stellt Einsatzmöglichkeiten und Anwendungen zusammen, erarbeitet einen Anforderungskatalog und diskutiert Lösungsmöglichkeiten für diese Fragestellungen. Hierbei werden die in der Literatur bekannten Lösungsvorschläge und eigene Ideen gegeneinander abgewogen. Insbesondere wird frühzeitig diskutiert, welche Hard– und Software für die jeweiligen Lösungen erforderlich ist, welche sonstigen Kosten entstehen, wie hoch der Zeitaufwand sein wird, usw. Wichtig ist eine frühe Spezifizierung der Eigenschaften des Systems (Robustheit, Antwortverhalten, Flexibilität, Schutzmechanismen, Erweiterbarkeit, Verteiltheit, ...).
Inhaltliches Ergebnis ist eine möglichst eindeutige, ausschnittsweise sogar formale Spezifikation. Für jede ins Auge gefaßte Anwendung wird darüber hinaus ein Szenario bzgl. des Einsatzes, der Nutzung, der Tests und der Wartung skizziert.
Organisatorische Ergebnisse sind ein grober Zeitplan und die erste Aufteilung von Aufgabengebieten. Hier setzt auch eine Spezialisierung der Gruppenmitglieder ein.

Entwurfsphase: Voraussetzung für die Entwurfsphase ist, daß Begriffsbestimmungen, Anwendungen und Modelle weitgehend geklärt sind. Nach Festlegung des grundsätzlichen Lösungsverfahrens werden Teilprobleme und charakteristische Objekte herauskristallisiert, miteinander in Beziehung gesetzt,

auf ihre Realisierbarkeit geprüft und grundlegende Datenstrukturen und Kommunikationswege festgelegt. Dabei werden die Schnittstellen der Einzelteile des Systems untereinander genau definiert. Ergebnis ist ein Plan des zu erstellenden (oder zu modifizierenden) Systems. Stehen die einzelnen Aufgaben fest, werden sie auf die Mitglieder verteilt. Die Implementierungssprache(n) sowie die erforderliche Hardware und die zu verwendenden Werkzeuge werden festgelegt. Eine Liste von Beispielen, die das System später positiv bewältigen muß, wird für die Testphase erstellt.

In der *Implementationsphase* und *Integrationsphase* wird der Programmcode erstellt, zusammengebunden (integriert) und getestet.

Die *Experimentierphase* schließt weitere Tests mit speziellen Anwendungen ein.

Zur *Schlußphase* zählt in erster Linie der Abschluß der *Dokumentation*, die ständig parallel zur Projektgruppenarbeit erstellt und auf den neuesten Stand gebracht wird.

Das Konzept der Projektgruppe wird bereits seit Jahren an anderen Universitäten wie z.B. in Oldenburg und Dortmund erprobt und durchgeführt. Dort sind Projektgruppen z.T. schon Pflichtveranstaltungen im Rahmen des Informatikstudiums.

1.2 Die Projektgruppe Evolutionäre Algorithmen

1.2.1 Aufgabenstellung

Aufgabe dieser Projektgruppe „Evolutionäre Algorithmen“ ist es, aufbauend auf die Ergebnisse der Projektgruppe „Genetische Algorithmen“ [AJJ+94, AJK+95] und des Technischen Berichts [JW95] ein System zur Bearbeitung hartnäckiger (NP–harter) Probleme mit Hilfe von Evolutionären Algorithmen wie z.B. Evolutionsstrategien und Genetischen Algorithmen zu erstellen.

Das generelle Vorgehen sollte sich dabei in die folgenden Punkte gliedern:

- Analyse des Problems
- Spezifikation des Systems
- Realisierung und Implementierung

- Erstellung von Testbibliotheken
- Durchführung von Experimenten
- Auswertung und Restrukturierungsvorschläge
- Dokumentation

Insgesamt soll das System einem Erstellungsaufwand von ca. drei Personenjahren entsprechen. Als Arbeitsmittel wurde Zugriff auf eine Workstation gewährt und ein Terminalraum mit Besprechungstisch zur 50%–igen Nutzung bereitgestellt.

1.2.2 Ziele

Das System soll folgende Konzepte enthalten:

1. Unterscheidung zwischen Problem– und Kodierungsstruktur. Dadurch wird eine einheitliche Darstellung des Problems erreicht und damit von der Sichtweise der Algorithmen getrennt. Den Übergang zwischen Problem– und Kodierungsstruktur bilden die Kodierungs– bzw die Dekodierungsfunktionen.
2. Verwendung verschiedener Datentypen innerhalb einer Problem– bzw. Kodierungsstruktur.
3. Möglichst freie Kombinierbarkeit von Verfahren und Operatoren, um unter Rückgriff auf vorhandene Operatoren neue Verfahren ausprobieren zu können.
4. Austauschbarkeit von Individuen verschiedener Kodierungen zwischen Verfahren, um hybride Verfahren möglich zu machen.
5. Abgestufte Einstiegsmöglichkeiten für den Benutzer.
6. Nebenläufige Algorithmen sollen implementierbar sein.

Teil I

Vorbereitung

Kapitel 2

Überblick über EAGLE

In diesem Kapitel soll ein knapper Überblick über die Funktionalität des Software–Systems EAGLE gegeben werden. Die funktionale Spezifikation dieses Systems (vgl. den technischen Bericht [JW95]) dient als Grundlage der Arbeit der Projektgruppe *Evolutionäre Algorithmen*.

Dieser Überblick baut auf den Endbericht der Projektgruppe Genetische Algorithmen [AJK+95] auf, aus dem den Kapiteln 3 bis 6 eine umfassendere Darstellung von EAGLE entnommen werden kann.

2.1 Aufgabe von EAGLE

Häufig werden Probleme der folgenden Form betrachtet: es wird in einem Lösungsraum ein Punkt gesucht, der bezüglich einer Funktion einen minimalen bzw. maximalen Funktionswert besitzt. Ist ein solches Problem „schwierig“ (z.B. NP–hart) und dadurch nicht effizient mit mathematischen Methoden zu lösen, werden andere Herangehensweisen notwendig. Neben der systematischen Suche (alles durchprobieren) und verschiedenen einfachen heuristischen Verfahren (wie z.B. reine Zufallssuche) haben sich hier auch sogenannte evolutionäre Verfahren bewährt, deren Arbeitsweise an die Evolution in der Natur angelehnt ist. Diese Verfahren sind der Gegenstand des Software–Entwurfs EAGLE.

In EAGLE lassen sich nahezu beliebige Probleme mittels der Struktur des Lösungsraums und der zu optimierenden Funktion über diesem Lösungsraum beschreiben. Zusätzlich lassen sich nahezu beliebige evolutionäre Verfahren,

die mit einer Menge von Lösungen aus einem Lösungsraum arbeiten, eingeben, mit denen das Optimum gesucht werden kann. Häufig ist es notwendig, den Lösungsraum an das jeweilige Verfahren anzupassen. Dazu gibt es die Möglichkeit, diesen zu kodieren.

Die Aufgabe von EAGLE ist nun die Anwendung des eingegebenen Verfahrens auf das eingegebene Problem. Zu Beginn einer solchen Simulation können noch verschiedene Einstellungen am zu simulierenden Verfahren in der Laufinitialisierung vorgenommen werden. Neben dem durch dieses Verfahren gefundenen „besten“ Element des Lösungsraums sind als Ausgabe auch noch weitere spezielle Bildschirmausgaben und gefilterte Ausgaben in Dateien notwendig. Dadurch soll insbesondere ermöglicht werden, daß die Arbeitsweise des Verfahrens detailliert untersucht werden kann und verschiedene Verfahren miteinander verglichen werden können. Interaktiv kann dabei auch in Simulationen eingegriffen werden, und das laufende Verfahren modifiziert werden. Diese Funktionsweise von EAGLE wird auch schematisch in Abbildung 2.1 dargestellt.

2.2 Problem

Ein zu untersuchendes Problem besteht aus einer Beschreibung des Lösungsraum und der sogenannten Fitneßfunktion darüber.

Der Lösungsraum wird durch ein beliebig langes Tupel aus den verschiedenen Datentypen

- Bit
- Integer (Intervall mit unterer und oberer Grenze)
- Real (Intervall mit unterer und oberer Grenze)
- Permutation mit Angabe der Größe

angegeben.

Die Fitneßfunktion wird als spezieller Fitneßoperator eingegeben. Sein Aufbau und seine Syntax entspricht im wesentlichen den Operatoren für das evolutionäre Verfahren und ist in den Abschnitten 2.5 und 2.6 dargestellt.

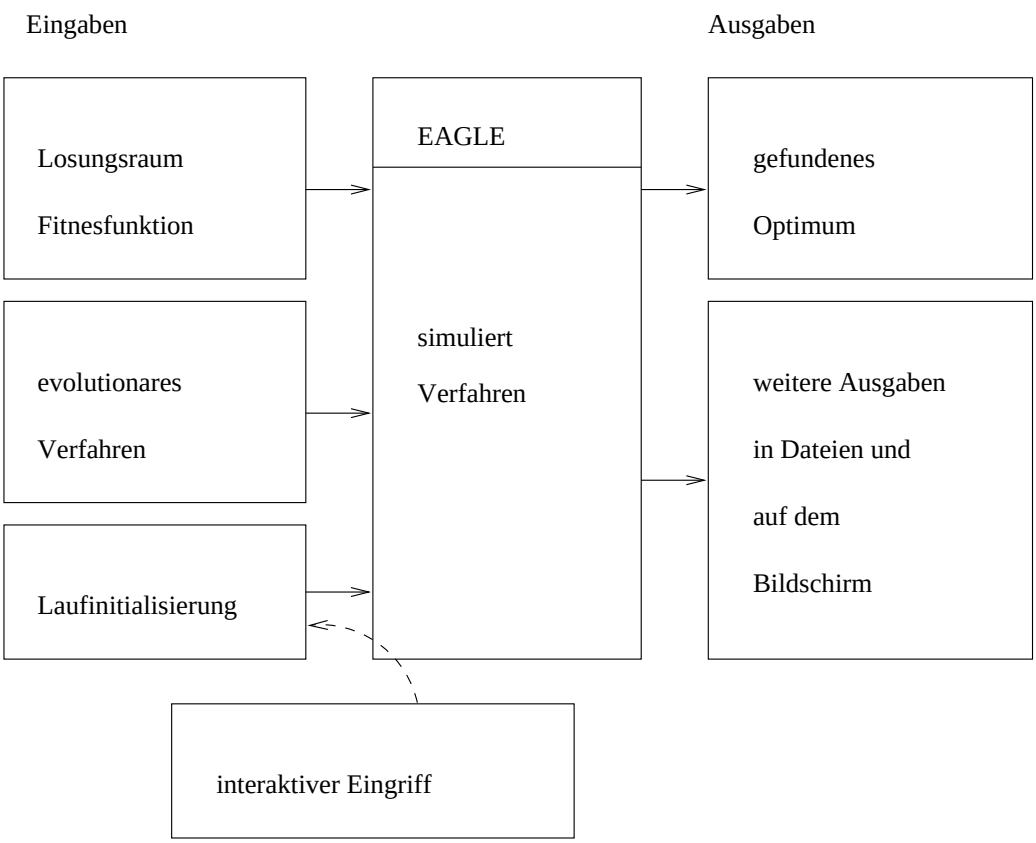


Abbildung 2.1: Funktionsweise von EAGLE

2.3 Kodierung und evolutionäres Verfahren

Damit ein Verfahren im Lösungsraum eines Problems nach einem optimalen Element suchen kann, wird der Lösungsraum durch eine Kodierung an das Verfahren angepaßt. Diese Kodierung wird verwirklicht durch eine Kodierung jedes einzelnen Datentyps im Tupel der Problemstruktur, das Einfügen von weiteren Strukturen, die im Verfahren als Strategieparametern genutzt werden können, und eine Umsortierung. Dadurch ergibt sich die Kodierungsstruktur. Auf ihr arbeitet das evolutionäre Verfahren. Der Zusammenhang zwischen Problem- und Kodierungsstruktur ist in Abbildung 2.2 dargestellt.

Zur binären Kodierung einzelner Atome wird die Anzahl der Bits eingegeben, durch welche sie kodiert werden sollen. D.h. das reelle oder ganzzahlige Intervall wird durch äquidistante Stützstellen repräsentiert, wobei sich die Anzahl der Stützstellen aus der Anzahl der verschiedenen Bitbelegungen berechnet.

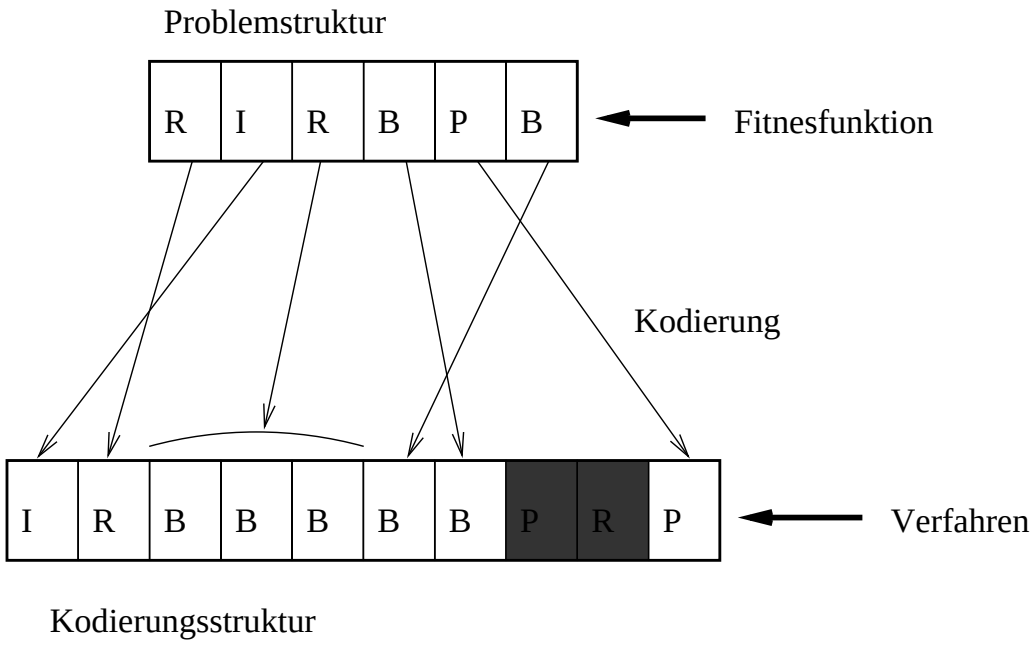


Abbildung 2.2: Problemstruktur und Kodierungsstruktur
Schematische Darstellung der Überführung einer Problemstruktur in eine Kodierungsstruktur, bei der ein reelles Atom in einen Bitstring kodiert wird, zwei zusätzliche Parameter (grau schraffiert) eingefügt und alle Atome umsortiert werden.

Dadurch entstehen keine ungültigen Belegungen im Bitstring, allerdings ist es möglich, daß bei einem kleineren ganzzahligem Intervall die Anzahl der verschiedenen Bitbelegungen größer ist als die Anzahl der möglichen Stützstellen. Dadurch kann ein ganzzahliger Wert durch mehrere verschiedene binäre Strings dargestellt werden. Dies ist im Beispiel in Tabelle 2.1 dargestellt.

Die verschiedenen Kodierungsmöglichkeiten der einzelnen Datentypen können Tabelle 2.2 entnommen werden.

Die Kodierungsstruktur bestimmt das Aussehen eines Individuums, welches in Evolutionären Algorithmen ja eine mögliche Lösung im Suchraum darstellt. D.h. jedes Individuum, das im evolutionären Verfahren verwendet wird, hat als Wert eine mögliche Wertebelegung der Kodierungsstruktur. Diese Belegung des Individuums wird während der evolutionären Suche vom Verfahren z.B. durch Mutation oder Crossover jeweils geändert. Soll die Fitneß eines Individuums bestimmt werden, werden zunächst die entsprechenden Werte der Problemstruktur aus den Werten der Kodierungsstruktur des Individuums bestimmt und anschließend wird die Fitneß durch den Fitneßoperator

Werte 1 ... 3		Werte 1 ... 7		Werte 1 ... 12		Werte 1 ... 20	
000	1	000	1	000	1	000	1
001	1	001	2	001	3	001	4
010	2	010	3	010	4	010	6
011	2	011	4	011	6	011	9
100	2	100	4	100	7	100	12
101	2	101	5	101	9	101	15
110	3	110	6	110	10	110	17
111	3	111	7	111	12	111	20

Tabelle 2.1: Beispiele zur binären Kodierung

Datentyp	**Kodierung**
Bit	Bit (keine Kodierung)
Integer	Integer (keine Kodierung)
	Bitstring (standardbinär kodiert durch Angabe der Anzahl der Bits)
	Bitstring (Gray–binär kodiert durch Angabe der Anzahl der Bits)
Real	Real (keine Kodierung)
	Bitstring (standardbinär kodiert durch Angabe der Anzahl der Bits)
	Bitstring (Gray–binär kodiert durch Angabe der Anzahl der Bits)
Permutation	Permutation (keine Kodierung)
	String reeller Zahlen (Anzahl entspricht der Größe der Permutation)
	Bitstring (durch standardbinäre Kodierung des reellen Zahlenstrings unter zusätzlicher Angabe der Bitanzahl)
	Bitstring (dito mit Gray-binärer Kodierung)

Tabelle 2.2: Übersicht über die verschiedenen Kodierungsmöglichkeiten

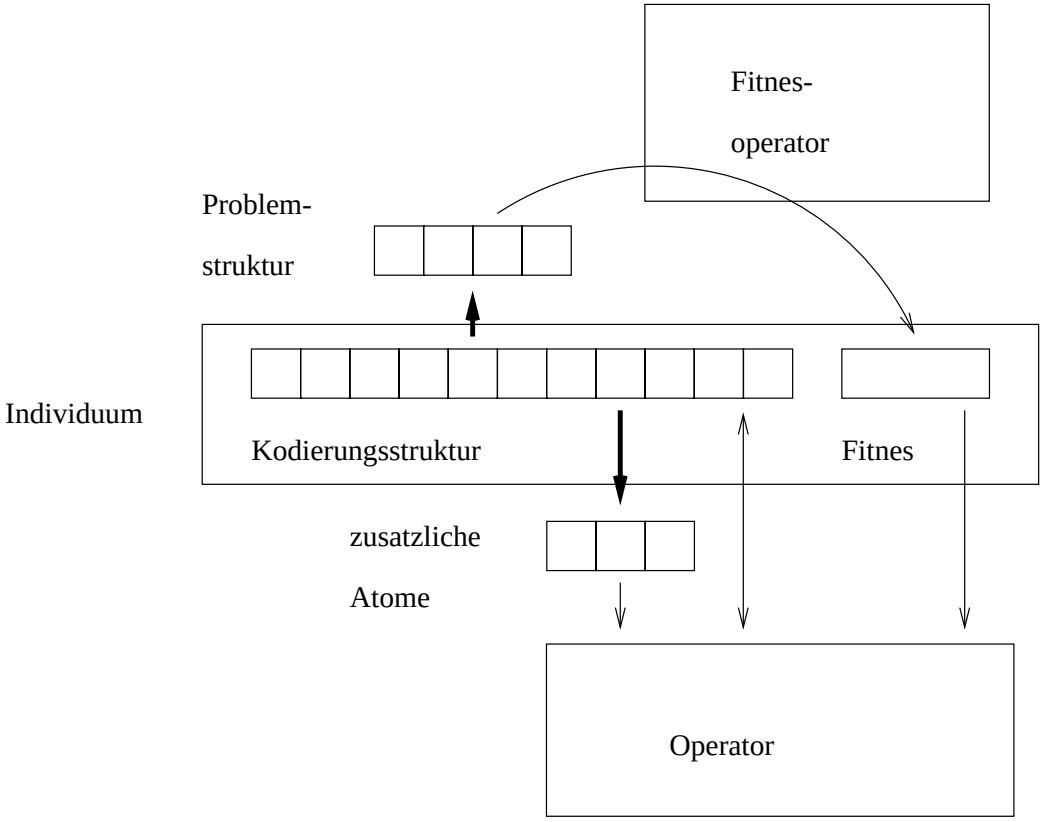


Abbildung 2.3: Schematische Darstellung des Individuums bestehend aus einer Kodierungsstruktur und einem Fitneßwert. Aus der Belegung der Kodierungsstruktur können die Belegungen der Problemstruktur und der zusätzlichen Parameter dekodiert werden. Der Fitneßoperator berechnet aus der Belegung der Problemstruktur den aktuellen Fitneßwert und der Operator greift auf die Belegung der zusätzlichen Atome zu.

berechnet, welcher ausschließlich auf den Werten der Problemstruktur arbeitet. Die zuletzt berechnete Fitneß wird immer zusätzlich im Individuum gespeichert. Das Zusammenspiel zwischen Individuum, Verfahren und Fitneßfunktion ist in Abbildung 2.3 schematisch dargestellt. Das evolutionäre Verfahren wird durch einen Operator, den sogenannten Hauptoperator, bestimmt. Dieser kann noch weitere Operatoren benutzen (siehe Abschnitt 2.4).

2.4 Konzept der Operatoren

Alle Eingaben des Benutzers, die Berechnungsvorschriften enthalten, werden in EAGLE als Operatoren eingegeben. Hierzu zählen insbesondere die Fitneßfunktion als Fitneßoperator und das evolutionäre Verfahren als Hauptoperator. Hierbei kann der Hauptoperator noch beliebige weitere Unteroperatoren verwenden und aufrufen. Dieses Operatorenkonzept ist beispielhaft in Abbildung 2.4 verdeutlicht.

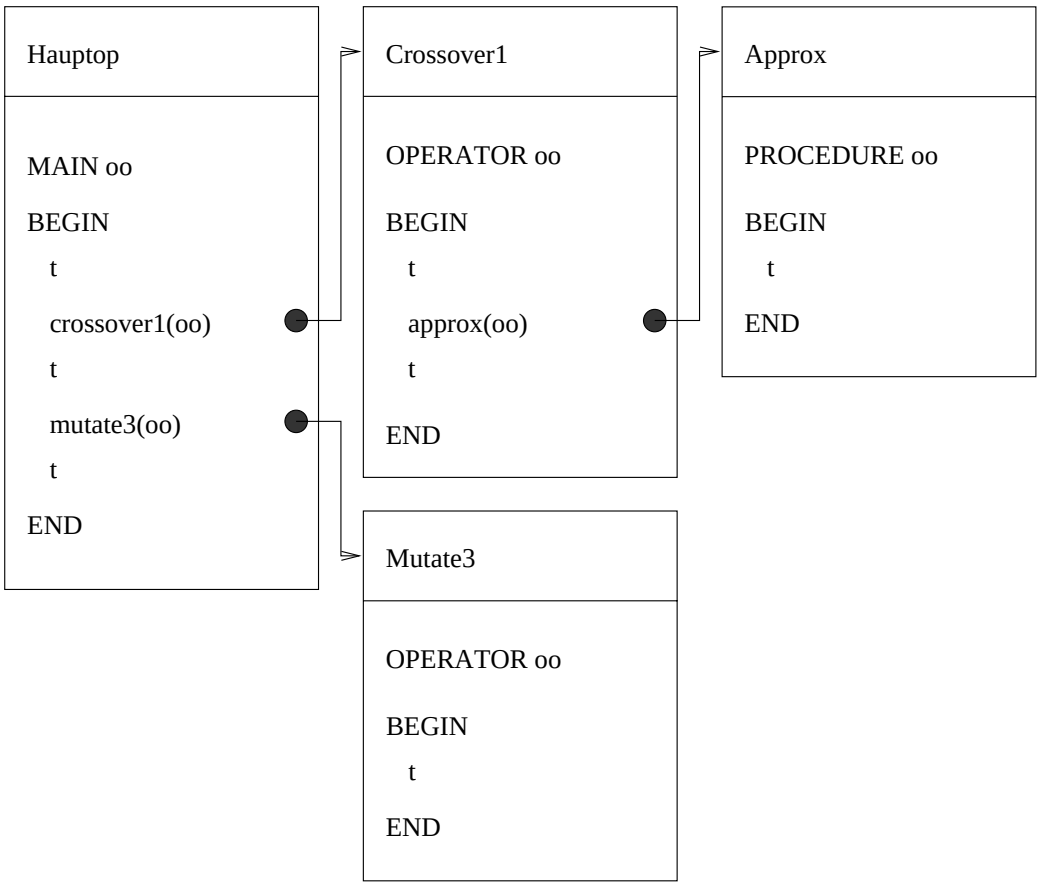


Abbildung 2.4: Operatorenkonzept

Die Algorithmen der Operatoren werden mittels der Programmiersprache LEA (siehe Abschnitt 2.6) eingegeben. Der allgemeine Aufbau der Operatoren ist in Abschnitt 2.5 erläutert.

Durch das Operatorkonzept soll eine hohe Wiederverwertbarkeit der Operatoren auch in weiteren Verfahren gewährleistet werden. Dabei ergibt sich

das Problem, daß bei unterschiedlichen Kodierungen der Teil der Individuen, der manipuliert werden soll, sich an verschiedenen Stellen in der Kodierungsstruktur befindet. Daher wurde eine Möglichkeit eingeführt, mit der beim Aufruf eines Unteroperators die Sichtbarkeit der Kodierungsstruktur eingeschränkt werden kann. Damit läßt sich ein Operator allgemein für den betroffenen Teil einer Kodierungsstruktur formulieren, während beim Aufruf des Operators die anderen Teile der Struktur einfach ausgeblendet werden, womit der Operator automatisch an der richtigen Stelle im Individuum arbeitet. Diese Funktionsweise ist beispielhaft auch in Abbildung 2.5 dargestellt.

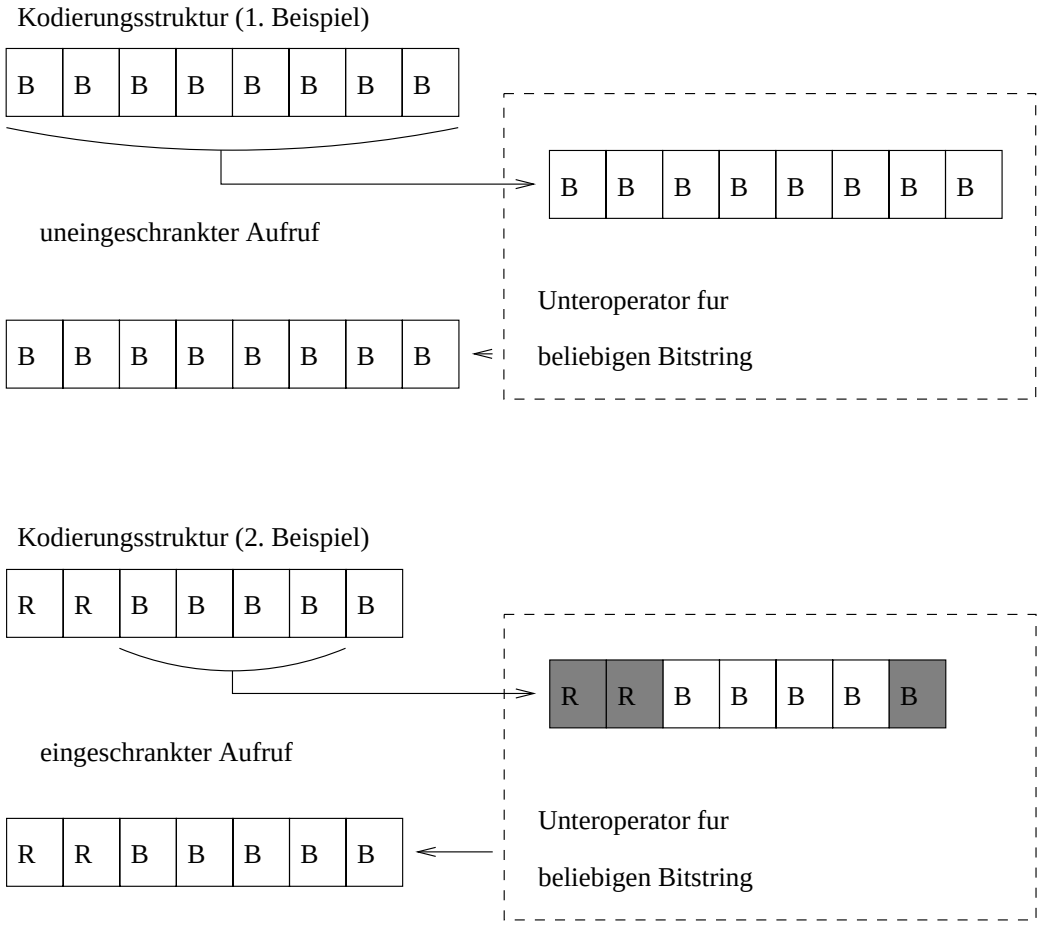


Abbildung 2.5: Funktionsweise des eingeschränkten Operatoraufrufs

2.5 Aufbau der Operatoren

Operatoren bestehen im wesentlichen aus drei Bestandteilen:

- der Kopfzeile des Operators — sie gibt den Typ des Operators (Fitneßoperator, Hauptoperator, etc.), die Aufrufparamter einschließlich der Art des Aufrufs (call–by–value oder call–by–reference) und den Rückgabetyp an.
- den lokalen Deklarationen für den Algorithmenteil — hier können Konstanten, Variablen und Parameter deklariert werden. Parameter sind Konstanten, deren Werte vor jeder Simulation in der Laufinitialisierung eingegeben werden. Außerdem wird hier angegeben, welche weiteren Unteroperatoren benötigt werden.
- dem Algorithmenteil — in ihm wird mittels der Sprache LEA die Berechnungsvorschrift für diesen Operator angegeben.

Als Datentypen für die Variablen, die Aufrufparameter und den Rückgabetyp stehen Bit, Integer, reelle Zahlen, Permutation, Individuum und (nur bei Verfahrensoperatoren) Population zur Verfügung. Konstanten und Parameter können nur als Bit, Integer oder reelle Zahlen deklariert werden. Für Variablen stehen laut [AJK+95] auch Felder (Arrays) der Datentypen zur Verfügung. Hier wird allerdings aus Gründen der Übersichtlichkeit auf Felder verzichtet.

2.6 Sprache LEA zur Eingabe der Operatoren

LEA (Language for Evolutionary Algorithms) wurde direkt als Eingabesprache für die Operatoren in EAGLE entwickelt. Sie lehnt sich stark an Sprachen wie Pascal ([JW74]) oder Modula–2 ([Wir82]) an. In diesem Abschnitt soll nur ein äußerst knapper Überblick über LEA gegeben werden. Eine wesentlich detailliertere Beschreibung befindet sich in Kapitel 6 von [AJK+95]. Deshalb werden hier hauptsächlich die Unterschiede, die zwischen [AJK+95] und [JW95] bestehen, vorgestellt. In dieser Kurzbeschreibung wird z.B. die Trennung zwischen dem Fitneßoperator und den Verfahrensoperatoren berücksichtigt. D.h. es gibt bestimmte Befehle, die nur in einem der beiden Operatortypen verwendet werden dürfen.

Es stehen folgende Sprachelemente als Anweisungen zur Verfügung:

- Wertzuweisungen in der Form "`<Variable> := <Ausdruck>`"
- IF–THEN–ELSE–END– bzw. IF–THEN–END–Verzweigung
- WHILE–Schleife
- FOR–Schleife
- FOREACH–Schleife mit der Syntax
  "`FOREACH <Variable> IN <Ausdruck> DO <Anweisungen> END`",
  wobei die Variable vom Typ Individuum und der Ausdruck vom Typ Population ist – dann werden für jedes Individuum in der Population die Anweisungen ausgeführt.
- RETURN–Anweisung zur Rückgabe des gesuchten Werts – sie darf nur am Ende eines Operators stehen.
- WRITE–Anweisung zur Ausgabe von Werten oder Text auf dem Bildschirm
- FILTER–Anweisungen zur Filterung von Daten in verschiedene Dateien — die Dateien werden den Filtern allerdings erst in der Laufinitialisierung zugeordnet.

Die FOREACH–Schleife und die WRITE– und FILTER–Anweisungen stehen nicht in Fitneßoperatoren zur Verfügung.

Daneben gibt es auch noch weitere spezielle evolutionäre Anweisungen, die zur Manipulation von Individuen, Populationen und Permutationen dienen. Sie können den Tabellen 2.3 bis 2.8 entnommen werden. In diesen Tabellen sind immer in der ersten Hälfte die Befehle angegeben, die in Ausdrücken verwendet werden können und in der zweiten Hälfte die Anweisungen. In Tabelle 2.9 sind noch weitere Funktionen aufgeführt.

Neben diesen Standardbefehlen können auch die importierten Operatoren aufgerufen werden. Falls hier Individuen als Argument übergeben werden, kann bei diesen die Sichtbarkeit wie in Abschnitt 2.4 beschrieben eingeschränkt werden. Hier wurde allerdings die Syntax im Vergleich zu [AJK+95] vollständig geändert. Die Einschränkung wird jetzt allgemein für alle Individuen, die in diesem Operator vorkommen, eingegeben, und es wird zudem auf eine genaue Angabe der Datentypen im Individuum verzichtet. Die Syntax lautet jetzt

```
Operator "{" Sichtbarkeit "}""(" Aufrufparameter ")".
```

Die Sichtbarkeit besteht dabei aus Zahlen zwischen 1 und der Länge der Kodierungsstruktur, welche durch "`,`" für Aufzählungen bzw. "`-`" für Bereichs-

angaben getrennt werden. So sind z.B. mit `{1,3-7}` nur das erste Atom und die Atome zwischen einschließlich dem dritten und dem siebten Atom im aufgerufenen Operator sichtbar.

Funktion	**Beschreibung**
`replength()`	liefert die Anzahl der Atome in der Problemstruktur
`numberrepbit(pos)`	liefert die Anzahl der zusammenhängenden Bits in der Problemstruktur ab der Position `pos`
`numberrepint(pos)`	dito mit Integern
`numberrepreal(pos)`	dito mit reellen Datentypen
`numberrepperm(pos)`	dito mit Permutationen

Tabelle 2.3: Funktionen zum Zugriff auf Informationen über Problemstruktur in Fitneßoperatoren

Zusätzlich können in Verfahrensoperatoren auch `LABEL` gesetzt werden. Ihnen wird vor jedem Experiment ihre Bedeutung zugeordnet. Sie dienen vor allem als definierte Halte– oder Abbruchpunkte.

2.7 Experiment

Ein Experiment bzw. eine Simulation wird durch die Eingabe der Problemstruktur, der Fitneßfunktion, der Kodierung, des evolutionären Verfahren und der Laufinitialisierung bestimmt.

In der Laufinitialisierung werden den offenen Parametern der Operatoren Werte zugewiesen, die Filter können aktiviert werden, indem sie auf eine Datei gelenkt werden, und die Label können als Halte– oder Abbruchlabel gesetzt werden. Zusätzlich wird ein Random Seed für den Zufallszahlengenerator eingegeben, und es ist möglich als neue Anfangspopulation eine schon bestehende Population aus einem anderen Experiment zu wählen.

Anschließend beginnt die Simulation des Verfahrens für das eingegebene Problem. Es wird während des Experiments laufend die derzeitige Generationsnummer, die beste und die mittlere Fitneß der derzeitigen Population und die Textausgaben aus den Operatoren angezeigt. Die gefilterten Daten werden in den Dateien der Laufinitialisierung abgelegt. Während der Simulation hat

der Benutzer jederzeit die Möglichkeit, den Lauf für Modifikationen der Laufinitialisierung anzuhalten, dann stoppt EAGLE beim Erreichen des nächsten aktiven Haltelabels, oder das Experiment zu beenden. Wird während der Simulation ein aktives Abbruchlabel erreicht, wird die Simulation ebenfalls sofort beendet.

Funktion	**Beschreibung**
`length()`	liefert die Anzahl der Atome in der Kodierungsstruktur
`numberbit(pos)`	liefert die Anzahl der zusammenhängenden Bits in der Kodierungsstruktur ab der Position `pos`
`numberint(pos)`	dito mit Integern
`numberreal(pos)`	dito mit reellen Datentypen
`numberperm(pos)`	dito mit Permutationen
`stratlength()`	liefert die Anzahl der zusätzlichen Atome (Strategieparameter)
`numberstratbit(pos)`	liefert die Anzahl der zusammenhängenden Bits in den zusätzlichen Atomen ab der Position `pos`
`numberstratint(pos)`	dito mit Integern
`numberstratreal(pos)`	dito mit reellen Datentypen
`numberstratperm(pos)`	dito mit Permutationen

Tabelle 2.4: Funktionen zum Zugriff auf Informationen über Kodierungsstruktur und die zusätzlichen Atome

Funktion	**Beschreibung**
`getbit(ind,pos)`	liefert das Atom an der Position `pos` im Individuum `ind` aus der Sicht der Kodierungsstruktur, falls dort ein Bit steht
`getint(ind,pos)`	dito mit Integer
`getreal(ind,pos)`	dito mit reellen Datentypen
`getperm(ind,pos)`	dito mit Permutationen
`getstratbit(ind,pos)`	liefert das Atom an der Position `pos` im Individuum `ind` aus der Sicht der zusätzlichen Atome, falls dort ein Bit steht
`getstratint(ind,pos)`	dito mit Integer
`getstratreal(ind,pos)`	dito mit reellen Datentypen
`getstratperm(ind,pos)`	dito mit Permutationen
`fitness(ind)`	liefert die zuletzt berechnete Fitneß des Individuums `ind`
`setbit(ind,pos,wert)`	setzt das Atom an der Position `pos` der Kodierungsstruktur im Individuum `ind` auf den Wert `wert`, falls dort ein Bit steht
`setint(ind,pos,wert)`	dito mit Integer
`setreal(ind,pos,wert)`	dito mit reellen Datentypen
`setperm(ind,pos,wert)`	dito mit Permutationen
`setstratbit(ind,pos,wert)`	setzt das Atom an der Position `pos` der zusätzlichen Atome im Individuum `ind` auf den Wert `wert`, falls dort ein Bit steht
`setstratint(ind,pos,wert)`	dito mit Integer
`setstratreal(ind,pos,wert)`	dito mit reellen Datentypen
`setstratperm(ind,pos,wert)`	dito mit Permutationen
`evaluate(ind)`	berechnet die Fitneß des Individuums `ind` mit dem Fitneßoperator und speichert sie beim Individuum

Tabelle 2.5: Funktionen für Individuen in Verfahrensoperatoren

Funktion	Beschreibung
`getrepbit(ind,pos)`	liefert das Atom an der Position `pos` im Individuum `ind` aus der Sicht der Problemstruktur, falls dort ein Bit steht
`getrepint(ind,pos)`	dito mit Integer
`getrepreal(ind,pos)`	dito mit reellen Datentypen
`getrepperm(ind,pos)`	dito mit Permutationen
`setrepbit(ind,pos,wert)`	setzt das Atom an der Position `pos` der Problemstruktur im Individuum `ind` auf den Wert `wert`, falls dort ein Bit steht
`setrepint(ind,pos,wert)`	dito mit Integer
`setrepreal(ind,pos,wert)`	dito mit reellen Datentypen
`setrepperm(ind,pos,wert)`	dito mit Permutationen

Tabelle 2.6: Funktionen für Individuen in Fitneßoperatoren

Funktion	Beschreibung
`getpermvalue(perm,pos)`	liefert den Wert der Permutation `perm` an der Stelle `pos`
`setpermvalue(perm,pos,wert)`	setzt den Wert der Permutation `perm` an der Stelle `pos` auf den Wert `wert`, die anderen Werte werden dabei verschoben
`xchangeperm(perm,pos1,pos2)`	vertauscht die Werte der Permutation `perm` an den Stellen `pos1` und `pos2`
`reverseperm(perm,pos1,pos2)`	spiegelt das Teilstück der Permutation `perm` zwischen den Stellen `pos1` und `pos2` einschließlich

Tabelle 2.7: Funktionen für Permutationen

Funktion	Beschreibung
`sizeofpop(pop)`	liefert Anzahl der Individuen in der Population `pop`
`getavgfitness(pop)`	liefert die durchschnittliche Fitneß der Individuen in der Population `pop`
`getbestfitness(pop)`	liefert die Fitneß des derzeit besten Individuums in der Population `pop`
`getworstfitness(pop)`	liefert die Fitneß des derzeit schlechtesten Individuums in der Population `pop`
`getind(pop,nr)`	liefert das Individuum mit der Nummer `nr` in der Population `pop`
`getbest(pop)`	liefert das Individuum mit der besten Fitneß in der Population `pop`
`getworst(pop)`	liefert das Individuum mit der schlechtesten Fitneß in der Population `pop`
`clearpop(pop)`	löscht alle Individuen in der Population `pop`
`mergepop(pop1,pop2)`	alle Individuen aus Population `pop2` werden zusätzlich in die Population `pop1` kopiert
`insertind(pop,ind,nr)`	das Individuum `ind` wird in die Population `pop` an der Position `nr` eingefügt, die weitere Numerierung verschiebt sich hierbei
`killinpop(pop,nr)`	in der Population `pop` wird das Individuum mit der Nummer `nr` gelöscht, die weitere Numerierung verschiebt sich hierbei
`evaluate(pop)`	für alle Individuen in der Population `pop` wird die Fitneß neu berechnet

Tabelle 2.8: Funktionen für Populationen in Verfahrensoperatoren

Funktion	**Beschreibung**
`getrandomreal()`	liefert eine reelle Zufallszahl zwischen 0 und 1
`getrandomint(ug,og)`	liefert eine ganzzahlige Zufallszahl zwischen `ug` und `og` einschließlich
`gen()`	liefert den Wert des globalen Generationenzählers
`incgen()`	erhöht den globalen Generationenzähler um 1
`write(out)`	schreibt `out` auf den Bildschirm, dabei kann `out` sowohl ein beliebiger Ausdruck als auch ein Textstring sein

Tabelle 2.9: Weitere Funktionen in Verfahrensoperatoren

Kapitel 3

Seminarvorträge

3.1 Algebraische Spezifikation und Typ–Polymorphismus

3.1.1 Einleitung

Eine erste Entscheidung war die Wahl der Programmiersprache ML für die Projektgruppe. Weil einer der Ausgangspunkte der Projektgruppe die formale Spezifikation von EAGLE war, lag es nahe, auch eine funktionale Programmiersprache zu verwenden. Da ML zusätzlich eine relativ kompakte Schreibweise des Quelltextes unterstützt, besteht die Hoffnung, damit relativ schnell Prototypen und erste eingeschränkte Versionen des Programms erstellen zu können. So unterstützt ML z.B. bei der Deklaration von Funktionen ein *pattern matching* der Parameter. Da ML polymorphe Typen verwendet, können Funktionen und Datentypen so geschrieben werden, daß sie auf möglichst viele verschiedene Typen anwendbar sind und gut wiederverwendet werden können. Als weiteren Vorteil, insbesonders gegenüber vielen anderen funktionalen Sprachen, besitzt ML, trotz der Typpolymorphie, eine strenge Typprüfung, mit der Typfehler schon vor Programmstart entdeckt werden können. Der Zusammenhang zwischen dieser Typprüfung und der Typpolymorphie in ML soll in diesem Abschnitt erläutert werden.

3.1.2 Einführung in den Lambda–Kalkül

Zum Verständnis der nachfolgenden Abschnitte erfolgt hier eine kurze Einführung in die Idee des funktionalen Programmierens und den Lambda–Kalkül.

3.1.2.1 Funktionales Programmieren

Beim funktionalen Programmieren bestehen das Programm und die dazugehörenden Daten aus einem Ausdruck E. Eine Reduktionsmaschine wandelt diese Eingabe mittels der Ersetzungsregeln solange um, bis keine dieser Regeln mehr anwendbar ist. Die Ersetzungsregeln haben die Form $P \to P'$ und geben an, daß der Teilausdruck P aus E durch P' ersetzt werden soll. Wenn in einem so gewonnenen Ausdruck E' kein Teilausdruck P mehr vorkommt, der auf der linken Seite einer Ersetzungsregel steht, nennt man ihn die Normalform von E. Dieser Ausdruck kann nicht mehr weiter umgewandelt werden. Er ist die Ausgabe zu dem funktionalen Programm E.

3.1.2.2 Lambda–Kalkül

Die Menge der λ –Terme (Λ) baut auf den Mengen der Konstanten $C = \{c, c', c'', \ldots\}$ und Variablen $V = \{v, v', v'', \ldots\}$ mittels folgender Regeln auf:

1. $c \in C \Rightarrow c \in \Lambda$
2. $x \in V \Rightarrow x \in \Lambda$
3. $M, N \in \Lambda \Rightarrow (MN) \in \Lambda$ (M auf N anwenden)
4. $M \in \Lambda, x \in V \Rightarrow (\lambda x.M) \in \Lambda$ (Abstraktion)

Abkürzungen: Zum Sparen von Klammern wird beim Anwenden einer Funktion Rechtsassoziativität und bei der Abstraktion Linksassoziativität verwendet:

$$FM_1M_2 \ldots M_n \text{ entspricht dann } (\ldots((FM_1)M_2)\ldots M_n) \text{ und}$$
$$\lambda x_1 \ldots x_n.M \text{ entspricht } \lambda x_1.(\ldots(\lambda x_n.M)\ldots).$$

Freie Variablen: Freie Variablen sind alle Variablen x, die nicht von einem entsprechenden λx eingeschlossen werden. Die Variablen, die zu einem umschließenden λx gehören, werden durch diese Abstraktion gebunden und demnach

als gebundene Variablen bezeichnet. Die Menge $FV(M)$ der freien Variablen von M kann man dann induktiv definieren als:

$$\begin{aligned} &FV(x) = \{x\}; \\ &FV(MN) = FV(M) \cup FV(N); \\ &FV(\lambda x.M) = FV(M) \Leftrightarrow \{x\}. \end{aligned}$$

Der Ausdruck $x(\lambda x.xy)$ enthält z.B. die freie Variable y und die Variable x einmal als gebundene und einmal als freie Variable. Ein Lambda-Term M heißt geschlossen, wenn er keine freien Variablen enthält, d. h. $FV(M) = \emptyset$.

Die Substitution $M[x := N]$ ersetzt die freien Vorkommen der Variable x in dem Term M durch N. Die freien Vorkommen werden dabei alle simultan ersetzt. Daher ist die Substitution auch nicht rekursiv. Gebundene Variablen x werden nicht ersetzt.

$$\begin{aligned} xy(\lambda x.xy)[x := N] &= Ny(\lambda x.xy) \\ xy(\lambda x.xy)[y := N] &= xN(\lambda x.xN) \end{aligned}$$

Mit Hilfe der Substitution kann man dann die Wirkungsweise des Lambda–Kalküls definieren.

- $(\lambda x.M)N = M[x := N]\ \forall M, N \in \Lambda$
- Die Regeln, die angeben, wann zwei Term gleich sein sollen:
  $M = M$, $\quad M = N \Rightarrow N = M$,
  $M = N, N = L \Rightarrow M = L$, $\quad M = N \Rightarrow MZ = NZ$,
  $M = N \Rightarrow ZM = ZN$, $\quad M = N \Rightarrow \lambda x.M = \lambda x.N$.

Beim Anwenden der ersten Regel wird das Vorkommen eines entsprechenden λx–Ausdrucks durch die rechte Seite der Regel ersetzt. Dies wird solange gemacht, wie in der Formel noch λx–Ausdrücke auf andere Ausdrücke angewendet werden. Wenn sich die Gleichheit zweier Ausdrücke mit Hilfe von diesen Regeln herleiten läßt ($M = N$), dann heißen sie *β–convertible*, und man schreibt $\lambda \vdash M = N$. Da sich die Bedeutung von zwei Ausdrücken M und N nicht unterscheidet, wenn nur die gebundenen Variablen unterschiedlich sind, schreibt man dann $M \equiv N$. So ist z.B.: $(\lambda x.x)z \equiv (\lambda y.y)z$.

Beispiel für das Ersetzen von λx–Ausdrücken:

$$\begin{aligned} &((\lambda x.(\lambda y.(x^2 + y^2)))4)3 \qquad (Abk.: \ \lambda xy.(x^2 + y^2)4\,3) \\ &\quad (\lambda y.(4^2 + y^2))3 \\ &\quad\quad 4^2 + 3^2 \end{aligned}$$

Der letzte Ausdruck wird hier nicht weiter ausgewertet. Er stellt die Normalform des ersten Ausdrucks dar. Erst wenn man noch zusätzliche Ersetzungsregeln für + und ^{2} einführt, bekommt man als Ergebnis dann auch 25 (siehe [Bar90]).

3.1.3 Einfache Typisierung

In diesem Abschnitt erfolgt die kurze Darstellung einer Möglichkeit, Terme des Lambda–Kalküls mit Typen zu versehen.

Die Menge der Typen (*Type*) wird dazu induktiv definiert durch:

- $\iota_0, \iota_1, \ldots \in \mathit{Type}$ (Grundtypen),
- $\alpha, \beta, \ldots \in \mathit{Type}$ (Typvariablen),
- $\sigma \in \mathit{Type}, \tau \in \mathit{Type} \Rightarrow (\sigma \to \tau) \in \mathit{Type}$.

Eine *Basis* ist eine Menge von Typzuweisungen zu Termvariablen (z.B. $B = \{x : \sigma, \ldots\}$, wobei der Variable x hier der Typ σ zugewiesen wird). Das folgende Axiom und die folgenden Regeln beschreiben induktiv, wann die Typisierung eines Ausdrucks M des Lambda–Kalküls mit einem Typ σ ($M : \sigma$) aus einer Basis B herleitbar ist.

Axiom:

- $B \vdash x : \sigma$, falls $x : \sigma \in B$

Regeln:

1. $$\frac{B \vdash M : \sigma \to \tau \quad B \vdash N : \sigma}{B \vdash MN : \tau}$$
2. $$\frac{B \cup \{x : \sigma\} \vdash M : \tau}{B \vdash \lambda x.M : \sigma \to \tau}$$

Das folgende Beispiel zeigt die Herleitung des Typs für den Ausdruck $\lambda x.yx$ mit der Basis $B = \{y : \sigma \to \sigma\}$:

$$\frac{\dfrac{B \vdash y : (\sigma \to \sigma) \Rightarrow}{B \cup \{x : \sigma\} \vdash y : (\sigma \to \sigma) \quad B \cup \{x : \sigma\} \vdash x : \sigma} \quad (1)}{\dfrac{B \cup \{x : \sigma\} \vdash (yx) : \sigma}{B \vdash \lambda x.yx : (\sigma \to \sigma)} \quad (2)}$$

$$\{y : \sigma \to \sigma\} \vdash \lambda x.yx : \sigma \to \sigma$$

Mit dieser Vorgehensweise kann jedoch nur die Korrektheit einer Typisierung nachgewiesen werden. So muß in diesem Beispiel der Typ für die gebundene Variable x selbst korrekt bestimmt werden, da sonst die Korrektheit der Typisierung nicht gezeigt werden kann. Ein Algorithmus, der zu einem Programm den korrekten Typ liefert, folgt im nächsten Abschnitt.

3.1.4 Typpolymorphismus

Für eine einfache Sprache *Exp* soll die Wirkungsweise des Algorithmus $\mathcal{W}$ beschrieben werden. Er liefert eine korrekte, aber noch möglichst freie Typisierung (falls vorhanden) zu einem beliebigen Ausdruck dieser Sprache. Da er nicht wie andere Typprüfungsalgorithmen verlangt, daß der zu prüfende Ausdruck in Normalform vorliegt, kann er noch vor Beginn der Reduktion des Ausdrucks eingesetzt werden und muß nicht bis zur Laufzeit des Programms warten. Allerdings ist er daher auch stark vom Aufbau der entsprechenden Sprache abhängig.

3.1.4.1 Die Sprache Exp

Im Folgenden wird eine einfache Sprache Exp vorgestellt, für die dann die Arbeitsweise des Algorithmus $\mathcal{W}$ gezeigt wird. Die Sprache und mit ihr der Algorithmus können dann später auf komplizierte Konstrukte und komplexere Typkombinationen erweitert werden.

Die Sprache Exp baut auf folgenden Elementen auf, wobei x eine Variable und e, e' und e'' Ausdrücke aus Exp sind:

1. x: Variable,
2. (ee'): e auf e' angewendet,
3. $if\, e\, then\, e'\, else\, e''$: bedingte Verzweigung,
4. $\lambda x.e$: Abstraktion (siehe oben),
5. $fix\, x.e$: kleinster Fixpunkt von $\lambda x.e$ und
6. $let\, x = e\, in\, e'$: Binden von e an x innerhalb von e'.

Da Exp recht einfach aufgebaut ist, können zur Laufzeit nur zwei Arten von Fehlern auftreten. Zum einen muß bei der Verzweigung $if\, e\, then\, e'\, else\, e''$ der Ausdruck e ein Ergebnis vom Typ *boolean* haben. Als zweites muß ein

Ausdruck e, der auf einen anderen angewendet wird (ee'), eine Funktion sein. Dies kann durch eine Typprüfung erzwungen werden.

Die für die Typisierung von Exp verwendeten Typen werden folgendermaßen aufgebaut, wobei beliebige Typen im Folgenden durch ρ, σ und τ dargestellt werden:

1. $\iota_0, \iota_1, \ldots$ sind die Grundtypen, wobei ι_0 der Typ für boolsche Werte ist.
2. Weiterhin gibt es eine aufzählbare Menge von Typvariablen. Diese sind ebenfalls Typen. $\alpha, \beta, \gamma, \ldots$ bezeichnen diese Typvariablen.
3. Der einzige Typoperator ist hier $\rightarrow$, der den Typ für eine Funktion erzeugt. Wenn ρ und σ Typen sind, dann ist auch $\rho \rightarrow \sigma$ ein Typ.

Alle Typen, die keine Typvariablen enthalten, werden als Monotypen (*monotypes*) bezeichnet. Alle, die eine oder mehrere Typvariablen enthalten, werden als Polytypen (*polytypes*) bezeichnet. Diese sind es, die die Polymorphie in einem Programm darstellen. Die Typvariablen in einem Polytyp stehen dabei für einen beliebigen Monotyp. So bedeutet z.B. $\alpha \rightarrow \alpha\ \forall\alpha.\alpha \rightarrow \alpha$, wobei α jeder beliebige Monotyp sein kann, aber nicht ein beliebiger Typ.

Das Folgende könnte leicht auch auf weitere Typoperatoren ausgebaut werden, wie sie z.B. auch in ML verwendet werden. So könnte man zusätzlich noch die binären Operatoren $\times$ für ein kartesisches Produkt und $\cup$ für Vereinigung einführen, sowie den unären Operator *list* für Listen.

3.1.4.2 Typkorrektheit

Hier wird zuerst die Eigenschaft *well–typed* (wt) vorgestellt, die aussagt, daß alle Teilausdrücke eines Programms die korrekten Typen haben. Dazu sind erst einmal einige Definitionen nötig.

Ein *Präfix* ist eine Folge von λx, *fix* x und *let* x, getrennt durch Punkte. Eine *prefixed expression* (pe) besteht aus einem Präfix p und einem Ausdruck e (geschrieben $p|e$), wobei alle freien Variablen aus e in p vorkommen. Jeder pe setzt sich nach folgenden Regeln aus Teilausdrücken mit dem dazugehörenden Präfix (sub–pe's) zusammen. Dabei gilt der transitive Abschluß.

1. $p|(ee')$ hat die sub–pe's $p|e$ und $p|e'$.
2. $p|(\mathit{if}\ e\ \mathit{then}\ e'\ \mathit{else}\ e'')$ hat die sub–pe's $p|e$, $p|e'$ und $p|e''$.
3. $p|(\lambda x.e)$ hat die sub–pe $p.\lambda x|e$.

4. $p|(\mathit{fix}\, x.e)$ hat die sub–pe $p.\mathit{fix}\, x|e$.

5. $p|(\mathit{let}\, x\ =\ e\ \mathit{in} e')$ hat die sub–pe's $p|e$ und $p.\mathit{let}\, x|e'$.

Diese Regeln zeigen, welche Definitionen für die Teilausdrücke gelten. So gilt grundsätzlich die Gesamtdefinition p für alle Teilausdrücke der Anweisung. Bei λ, *fix* und *let* gelten zusätzlich die lokalen Definitionen für den Teilausdruck. Bei *let* gilt aber die Definition von x nicht für den Teilausdruck e.

Bsp.:

$$\lambda x.(\lambda y.(\mathit{let}\, g = (\lambda x.(y(yx)))\, \mathit{in}\, (gx))) \text{ hat u. a. die sub–pe's}$$
$$\lambda x.\lambda y.\lambda x|(y(yx)) \text{ und } \lambda x.\lambda y.\lambda x.\mathit{let}\, g|(gx).$$

Eine Definition $\mathit{let}\, x$, $\mathit{fix}\, x$ oder λx ist *aktiv* im Präfix, wenn rechts davon keine Definition mit x mehr vorkommt. Im Ausdruck $\lambda x.\lambda y.\lambda x|(y(yx))$ ist z.B. das zweite λx aktiv, das erste aber nicht. Diese Unterscheidung wird später wichtig, wenn die beiden x verschiedene Typen haben.

Beim Typisieren eines Ausdrucks $p|e$ wird jeder Definition λx, $\mathit{fix}\, x$ und $\mathit{let}\, x$ und jedem Teilausdruck und jeder Definition in e ein Typ zugewiesen. Dargestellt als $\bar{p}|\bar{e}_\sigma$, wenn dem Ausdruck e der Typ σ zugewiesen wurde.

Das folgende Beispiel zeigt die Typisierung eines Exp–Ausdrucks:

$$(\lambda y_{\iota_1}.\mathit{let}\, f_{\alpha\to\alpha} = (\lambda x_\alpha.x_\alpha)_{\alpha\to\alpha}\, \mathit{in}\, f_{\iota_1\to\iota_1} y_{\iota_1})_{\iota_1\to\iota_1}$$

Die Typen für die Teilausdrücke sind so gewählt, daß sich entsprechende Ausdrücke den gleichen Typ haben (oder bei *let* einen Typ in dem die Typvariablen konkrete Typen angenommen haben). Ein so typisierter Ausdruck wird später dann als *well–typed* bezeichnet.

Generische Variablen: Eine Typvariable im Typ σ eines $\mathit{let}\, x_\sigma$ aus dem Präfix oder dem Ausdruck einer pe ist generisch, wenn sie in keinem Typ τ eines umschließenden λx_τ oder $\mathit{fix}\, x_\tau$ vorkommt. In dem Ausdruck $\lambda y_\alpha.\mathit{let}\, x_{\alpha\to\beta} = \ldots\, \mathit{in}(x_{\alpha\to\beta} y_\alpha)_\beta$ ist für das *let* die Typvariable β generisch, die Typvariable α jedoch nicht. Die generischen Variablen drücken den Polymorphismus aus, der in einem mit *let* definierten Ausdruck steckt.

Eine *generische Instanz* von σ ist eine Instanz von σ, wobei nur die generischen Typvariablen konkrete Typen annehmen.

Standard: Eine pe $\bar{p}|\bar{e}$ ist standard, wenn für alle sub–pe's $\bar{p}'|\bar{e}'$ die generischen Typvariablen jedes $\mathit{let}\, x_\sigma$ aus $\bar{p}'$ sonst nirgends in $\bar{p}'|\bar{e}'$ vorkommen. Dies verhindert, daß es zu Konflikten mit gleichnamigen Typen kommt, die anderswo verwendet werden.

Eine *prefixed expression* (pe) wird als *well–typed* (wt) bezeichnet, wenn folgendes gilt:

1. $\bar{p}|x_\tau$ ist genau dann wt, wenn er standard ist und entweder
   (a) λx_τ oder $\mathit{fix}\, x_\tau$ ist aktiv in $\bar{p}$, oder
   (b) $\mathit{let}\, x_\sigma$ ist aktiv in $\bar{p}$ und τ ist eine generische Instanz von σ.
2. $\bar{p}|(\bar{e}_\rho \bar{e}'_\sigma)_\tau$, ist genau dann wt, wenn sowohl $\bar{p}|\bar{e}$ als auch $\bar{p}|\bar{e}'$ wt sind und $\rho = \sigma \to \tau$.
3. $\bar{p}|\,(\mathit{if}\, \bar{e}_\rho\, \mathit{then}\, \bar{e}'_\sigma\, \mathit{else}\, \bar{e}''_{\sigma'})_\tau$ ist genau dann wt, wenn $\bar{p}|\bar{e}$, $\bar{p}|\bar{e}'$ und $\bar{p}|\bar{e}''$ wt sind, $\rho = \iota_0$ und $\sigma = \sigma' = \tau$.
4. $\bar{p}|(\lambda x_\rho.\bar{e}_\sigma)_\tau$ ist genau dann wt, wenn $\bar{p}.\lambda x_\rho|\bar{e}$ wt ist und $\tau = \rho \to \sigma$.
5. $\bar{p}|(\mathit{fix}\, x_\rho.\bar{e}_\sigma)_\tau$ ist genau dann wt, wenn $\bar{p}.fix\, x_\rho|\bar{e}$ wt ist und $\rho = \sigma = \tau$.
6. $\bar{p}|(\mathit{let}\, x_\rho = \bar{e}_{\rho'}\ \mathit{in}\, \bar{e}'_\sigma)_\tau$ ist genau dann wt, wenn $\bar{p}|\bar{e}$ und $\bar{p}.\mathit{let}\, x_\rho|\bar{e}'$ wt sind und $\rho = \rho'$, $\sigma = \tau$.

Grundsätzlich gilt für alle Regeln, daß die Teilausdrücke, aus denen ein Ausdruck besteht, ebenfalls erst einmal wt sein müssen. Punkt 1 sorgt zusätzlich dafür, daß jede Variable mit dem Typ ihrer Deklaration übereinstimmt, und daß jede Variable deklariert wurde. Punkt 2 stellt sicher, daß bei der Parameterübergabe ein Parameter mit dem richtigen Typ übergeben wird und der Typ, den die Funktion zurückliefert, stimmt. Bei $\mathit{if}\, d\, \mathit{then}\, e\, \mathit{else}\, e'$ muß der Typ der beiden Zweige e und e' mit dem Ergebnistyp übereinstimmen und der Bedingungsteil d muß vom Typ *boolean* sein. Bei den Deklarationen λx, $\mathit{fix}\, x$ und $\mathit{let}\, x$ muß der Typ des Ausdruck mit dem gesuchten Typ für das Ergebnis übereinstimmen. Für die Untersuchung des Ausdrucks e gilt dann zusätzlich noch die (lokale) Deklaration für die Variable x. Eine Besonderheit ist hier die unterschiedliche Behandlung von λx (bzw. $\mathit{fix}\, x$) und $\mathit{let}\, x$. So kann eine mit $\mathit{let}\, x_\sigma = \ldots\ \mathit{in}\, e$ definierte Variable x bei jedem Auftreten in e einen anderen Typ haben, solange alle generische Instanzen von σ sind. Bei einem Ausdruck $(\lambda x_\sigma.e)e'$ (bzw. $\mathit{fix}\, x_\sigma.e$) müssen jedoch alle Vorkommen von x in e' (bzw. bei $\mathit{fix}\, x$ in e) den Typ σ haben.

Bsp.:

$$\lambda f_{\rho\to(\sigma\to\tau)}.\mathit{let}\, x_\alpha = \ldots\ \mathit{in}((f\, x_\rho)x_\sigma)_\tau \text{ ist daher wt.}$$

$$\lambda f_{\rho\to(\sigma\to\tau)}.\lambda x_\alpha.((f\, x_\rho)x_\sigma)_\tau \text{ ist jedoch nicht wt.}$$

Man kann mit einer strukturellen Induktion über die obigen Regeln zeigen, daß ein Programm, das nach dieser Definition *well–typed* ist, während des

Ablaufs keine Ausdrücke mit falschen Typen als Operatoren oder Operanden verwendet (siehe [Mil78]).

3.1.4.3 Der Algorithmus $\mathcal{W}$

Es folgt nun der Algorithmus $\mathcal{W}$, der zu einem Deklarationsteil (mit Typen) $\bar{p}$ und einem Programm f das typisierte Programm $\bar{f}$ und eine Substitution T liefert, die Typen miteinander vereinbart. Der Algorithmus benötigt den Unifikationsalgorithmus $\mathcal{U}$. Dieser liefert für $\mathcal{U}(\sigma, \tau)$ ein allgemeinstes U mit Substitutionen für die in σ und τ enthaltenen Typvariablen, das σ und τ unifiziert, d. h. $U\sigma = U\tau$.

Beispiel: Unifikation von $(\alpha \to \beta) \to \iota_0$ und $(\beta \to \alpha) \to \gamma$:

$$\mathcal{U}(((\alpha \to \beta) \to \iota_0), ((\beta \to \alpha) \to \gamma)) = U = [\beta := \alpha][\gamma := \iota_0]$$
es ist dann: $U(((\alpha \to \beta) \to \iota_0)) = U(((\beta \to \alpha) \to \gamma)) = ((\alpha \to \alpha) \to \iota_0)$

Algorithmus $\mathcal{W}$: $\mathcal{W}(\bar{p}, f) = (T, \bar{f})$:

Ist f:

1. $f = x$:
   (a) Wenn λx_σ oder $\mathit{fix}\, x_\sigma$ in $\bar{p}$ aktiv ist:
   $T = I, \bar{f} = x_\sigma$. ($I$ ist dabei die leere Substitution [ ])

   (b) Wenn $\mathit{let}\, x_\sigma$ in $\bar{p}$ aktiv ist:
   $T = I, \bar{f} = x_\tau$, wobei $\tau = [\alpha_i := \beta_i]\sigma$ ist.
   β_i sind neue Variablen und α_i sind die generischen Variablen von σ.

2. $f = (de)$:
   Führe den Algorithmus $\mathcal{W}$ für d und e aus:
   $(R, \bar{d}_\rho) = \mathcal{W}(\bar{p}, d)$, $(S, \bar{e}_\sigma) = \mathcal{W}(R\bar{p}, e)$.
   $U = \mathcal{U}(S\rho, \sigma \to \beta)$, wobei β eine neue Variable ist.
   $T = USR$, $\bar{f} = U(((S\bar{d})\bar{e})_\beta)$.

3. $f = (\mathit{if}\, d\, \mathit{then}\, e\, \mathit{else}\, e')$:
   $(R, \bar{d}_\rho) = \mathcal{W}(\bar{p}, d)$ und $U_0 = \mathcal{U}(\rho, \iota_0)$.
   $(S, \bar{e}_\sigma) = \mathcal{W}(U_0 R\bar{p}, e)$, $(S', \bar{e}'_{\sigma'}) = \mathcal{W}(SU_0 R\bar{p}, e')$ und $U = \mathcal{U}(S'\sigma, \sigma')$.
   $T = US'SU_0R$, $\bar{f} = U((\mathit{if}\, S'SU_0\bar{d}\, \mathit{then}\, S'\bar{e}\, \mathit{else}\, \bar{e}')_\sigma)$.

4. $f = (\lambda x.d)$:
   $(R, \bar{d}_\rho) = \mathcal{W}(\bar{p}.\lambda x_\beta, d)$, wobei β eine neue Variable ist.
   $T = R$, $\bar{f} = (\lambda x_{R\beta}.\bar{d}_\rho)_{R\beta\to\rho}$.

5. $f = (\mathit{fix}\, x.d)$:
   $(R, \bar{d}_\rho) = \mathcal{W}(\bar{p}.\mathit{fix}\, x_\beta, d)$, wobei β wieder eine neue Variable ist.
   $U = \mathcal{U}(R\beta, \rho)$.
   $T = UR$, $\bar{f} = (\mathit{fix}\, x_{UR\beta}.U\bar{d})_{UR\beta}$.

6. $f = (\mathit{let}\, x = d\, \mathit{in}\, e)$:
   $(R, \bar{d}_\rho) = \mathcal{W}(\bar{p}, d)$ und
   $(S, \bar{e}_\sigma) = \mathcal{W}(R\bar{p}.\mathit{let}\, x_\rho, e)$.
   $T = SR$, $\bar{f} = (\mathit{let}\, x_{S\rho} = S\bar{d}\, \mathit{in}\, \bar{e})_\sigma$.

Man kann zeigen, daß ein Programm, das mit diesem Algorithmus typisiert wurde, auch *well–typed* ist (siehe [Mil78]). Der Beweis erfolgt mit struktureller Induktion über f, wobei die rekursive Definition von $\mathcal{W}$ ausgenützt wird. Ein Exp–Programm, das diesen Algorithmus erfolgreich durchlaufen hat, wird also nicht mehr an einem falschen Typ scheitern.

Indem der Algorithmus allen noch nicht eingeschränkten Ausdrücken eine neue Typvariable β_n zuweist, garantiert er anfangs eine größtmögliche Freiheit für die Typen dieser Ausdrücke. Falls dann während des weiteren Ablaufs eine Einschränkung für diese Typen erkannt wird (z.B. muß der erste Teilausdruck einer if–Anweisung einen boolschen Typ ι_0 haben), werden sie, wenn möglich, durch die Substitutionen entsprechend eingeschränkt.

Am folgenden Beispiel soll die Arbeitsweise des Algorithmus für einen kürzeren Ausdruck gezeigt werden.

$$\text{f ist } \lambda x.(\lambda y.(\mathit{let}\, g = (\lambda x.(y(yx)))\, \mathit{in}\, gx))$$

$\mathcal{W}(\emptyset, f)$:

1. Wegen 4. (λ–Ausdruck):
   $(R_1, \bar{d}_1) = \mathcal{W}(\lambda x_{\beta_1}, \lambda y.(\mathit{let}\, g = (\lambda x.y(yx))\, \mathit{in}\, gx))$

2. Wegen 4. (λ–Ausdruck):
   $(R_2, \bar{d}_2) = \mathcal{W}(\lambda x_{\beta_1}.\lambda y_{\beta_2}, \mathit{let}\, g = (\lambda x.y(yx))\, \mathit{in}\, gx)$

3. Wegen 6. (*let*–Ausdruck):

   - $(R_3, \bar{d}_3) = \mathcal{W}(\lambda x_{\beta_1}.\lambda y_{\beta_2}, \lambda x.(y(yx)))$

     Wegen 4. (λ–Ausdruck): $(R_4, \bar{d}_4) = \mathcal{W}(\lambda x_{\beta_1}.\lambda y_{\beta_2}.\lambda x_{\beta_3}, (y(yx)))$

Wegen 2. (y angewendet auf (xy)):

- $(R_5, \bar{d_5}) = \mathcal{W}(\lambda x_{\beta_1}.\lambda y_{\beta_2}.\lambda x_{\beta_3}, y)$
  Wegen 1.(a) (λy aktiv im Präfix): $R_5 = I, \bar{d_5} = y_{\beta_2}$

- $(R_6, \bar{d_6}) = \mathcal{W}(\lambda x_{\beta_1}.\lambda y_{\beta_2}.\lambda x_{\beta_3}, (yx))$

  Wegen 2. (y angewendet auf x):

  * $(R_7, \bar{d_7}) = \mathcal{W}(\lambda x_{\beta_1}.\lambda y_{\beta_2}.\lambda x_{\beta_3}, y)$
    Wegen 1.(a) (λy aktiv im Präfix): $R_7 = I, \bar{d_7} = y_{\beta_2}$

  * $(R_8, \bar{d_8}) = \mathcal{W}(\lambda x_{\beta_1}.\lambda y_{\beta_2}.\lambda x_{\beta_3}, x)$
    Wegen 1.(a) (λx aktiv im Präfix): $R_8 = I, \bar{d_8} = x_{\beta 3}$

  * $U_1 = \mathcal{U}(\beta_2, \beta_3 \to \beta_4) = [\beta_2 := \beta_3 \to \beta_4]$

  $R_6 = [\beta_2 := \beta_3 \to \beta_4], \bar{d_6} = (y_{\beta_3 \to \beta 4} x_{\beta_3})_{\beta_4}$
  Hier sind die Typen für den Parameter und das Ergebnis der Funktion y noch verschieden ($\beta_3 \to \beta_4$).

- $U_2 = \mathcal{U}(R_6 \beta_2, \beta_4 \to \beta_5) = \mathcal{U}(\beta_3 \to \beta_4, \beta_4 \to \beta_5) = [\beta_3 := \beta_5][\beta_4 := \beta_5]$

$R_4 = [\beta_3 := \beta_5][\beta_4 := \beta_5][\beta_2 := \beta_5 \to \beta_5]$,
$\bar{d_4} = (y_{\beta_5 \to \beta_5}(y_{\beta_5 \to \beta_5} x_{\beta_5})_{\beta_5})_{\beta_5}$
Da die Funktion y jetzt auf den Ausdruck (yx) angewendet wird, muß der Typ des Parameters x von y gleich dem Ergebnistyp von y sein. Weil für das Ergebnis der Funktion y der neue Typ β_5 angenommen wurde, werden die benutzten Typvariablen durch Unifizieren entsprechend eingeschränkt. Man erhält als Typ für die Funktion y jetzt $\beta_5 \to \beta_5$.

$R_3 = R_4, \bar{d_3} = (\lambda x_{\beta_5}.(y_{\beta_5 \to \beta_5}(y_{\beta_5 \to \beta_5} x_{\beta_5})_{\beta_5})_{\beta_5 \to \beta_5}$

- $(R_9, \bar{d_9}) = \mathcal{W}(\lambda x_{\beta_1}.\lambda y_{\beta_5 \to \beta_5}.let\, g_{\beta_5 \to \beta_5}, (ge))$

Wegen 2. (g angewendet auf x):

- $(R_{10}, \bar{d}_{10}) = \mathcal{W}(\lambda x_{\beta_1}.\lambda y_{\beta_5 \to \beta_5}.let\, g_{\beta_5 \to \beta_5}, g)$
  Wegen 1.(b) ($let\, x$ aktiv im Präfix):
  $R_{10} = I, \bar{d}_{10} = g_{\beta_5 \to \beta_5}$
  Es werden hier keine neuen Variablen eingeführt, da $(\beta_5 \to \beta_5)$ keine generischen Variablen enthält.

- $(R_{11}, \bar{d}_{11}) = \mathcal{W}(\lambda x_{\beta_1}.\lambda y_{\beta_5 \to \beta_5}.let\, g_{\beta_5 \to \beta_5}, x)$
  Wegen 1.(a) (λx aktiv im Präfix):
  $R_{11} = I, \bar{d}_{11} = x_{\beta_1}$

- $U_3 = \mathcal{U}(\beta_5 \to \beta_5, \beta_1 \to \beta_6) = [\beta_1 := \beta_6][\beta_5 := \beta_6])$

$$R_9 = U_3, \bar{d}_9 = (g_{\beta_6 \to \beta_6} x_{\beta_6})_{\beta_6}$$

$$R_2 = R_9 R_3, \bar{d}_2 = (let\, g_{\beta_6 \to \beta_6} = (\lambda x_{\beta_6}.(y_{\beta_6 \to \beta_6}(y_{\beta_6 \to \beta_6} x_{\beta_6})_{\beta_6})_{\beta_6})_{\beta_6 \to \beta_6}\ in\, (g_{\beta_6 \to \beta_6} x_{\beta_6})_{\beta_6})_{\beta_6}$$

$$R_1 = R_2, \bar{d}_1 = (\lambda y_{\beta_6 \to \beta_6}.(\ldots))_{(\beta_6 \to \beta_6) \to \beta_6}$$

Ergebnis:

$$\begin{aligned} \mathcal{W}(\emptyset, f) \;=\; & ([\beta_1 := \beta_6][\beta_5 := \beta_6][\beta_3 := \beta_5][\beta_4 := \beta_5][\beta_2 := \beta_5 \to \beta_5], \\ & (\lambda x_{\beta_6}.(\lambda y_{\beta_6 \to \beta_6}.(let\, g_{\beta_6 \to \beta_6} = \\ & (\lambda x_{\beta_6}.(y_{\beta_6 \to \beta_6}(y_{\beta_6 \to \beta_6} x_{\beta_6})_{\beta_6})_{\beta_6})_{\beta_6 \to \beta_6} \\ & in\, (g_{\beta_6 \to \beta_6} x_{\beta_6})_{\beta_6})_{\beta_6})_{(\beta_6 \to \beta_6) \to \beta_6})_{\beta_6 \to ((\beta_6 \to \beta_6) \to \beta_6)} \end{aligned}$$

Am Anfang werden den vorkommenden Typen hier die Typvariablen β_1 bis β_6 zugewiesen. Die Struktur dieses Programmstücks erzwingt dann die Einschränkung dieser Typen durch die Substitutionen, bis schließlich nur noch die Typvariable β_6 vorkommt.

3.1.5 Fazit

Der Algorithmus $\mathcal{W}$ bietet die Möglichkeit eine Typprüfung durchzuführen, die gleichzeitig eine relativ große Freiheit für die Typen der Ausdrücke zuläßt.

In der obigen Form ist der Algorithmus für eine Implementierung allerdings zu unhandlich. In [Mil78] wird zusätzlich zum Algorithmus $\mathcal{W}$ noch der Algorithmus $\mathcal{J}$ vorgestellt. Dieser leistet dasselbe, ist aber effizienter zu implementieren, da er u. a. bei den Substitutionen mit einer globalen Variablen arbeitet und auf die Ausführung der Substitutionen oft verzichtet. Die in der Programmiersprache ML durchgeführte Typprüfung basiert auf diesem Algorithmus.

3.2 Sammlung von Problemen und Optimierungsverfahren

3.2.1 Einleitung

Im Rahmen der Projektgruppe „Evolutionäre Algorithmen“ sollen einige Vorträge den beteiligten StudentInnen den schnellen Einstieg in das Thema ermöglichen. Diese Ausarbeitung zu einem Hauptseminar soll einen Überblick über typische Optimierungsprobleme geben und einige Optimierungsverfahren erläutern.

Im ersten Teil wird auf zwei grundsätzliche Problemklassen eingegangen: die mathematischen Funktionen und die praktischen Anwendungen.

Der zweite Teil beschäftigt sich mit Optimierungsverfahren, die sich in drei große Gruppen einteilen lassen:

- enumerierende Verfahren
  Bei diesen Verfahren wird der gesamte Lösungsraum durchsucht und die beste Lösung durch Vergleich ermittelt. Dies ist die simpelste Vorgehensweise und muß daher nicht weiter erklärt werden.

- kalkülbasierte Verfahren
  Diese Verfahren lassen sich nur auf mathematische Probleme anwenden, d.h. ggf. muß ein Problem erst in eine mathematische Notation gebracht werden. Es gibt nun zwei mögliche Varianten, die „direkten“ und die „indirekten“ Verfahren. Letztere betrachten nicht die Werte der Problemfunktionen selbst, sondern die von aus ihnen abgeleiteten Funktionen. Beispielsweise werden Nullstellen der ersten Ableitung einer Funktion gesucht, da sich an diesen Stellen (lokale) Extrema befinden können. Dies hat natürlich zur Folge, daß viele Funktionen nicht mit diesen Verfahren untersucht werden können, da erste Ableitungen nicht formuliert werden können oder nicht existieren.

  Dagegen verwenden die direkten Verfahren die Funktionswerte der Problemfunktion selbst, um sich an ein Extremum anzunähern.

  Die bekanntesten Vertreter der kalkülbasierten Verfahren sind die „Hill-Climbing“ – Strategien, die im folgenden vorgestellt werden.

- zufallsgesteuerte Verfahren
  Hier werden Anfangskonfigurationen zufällig ausgewählt und nach verschiedenen Methoden aus diesen neue Konfigurationen bestimmt. Sie

werden bewertet und entweder übernommen oder verworfen. Dies wird wiederholt, bis eine Abbruchbedingung erfüllt ist.

Rein zufallsgesteuerte Verfahren, wie z.B. „Monte–Carlo“ und „Random Walk“, sind nicht Gegenstand dieser Ausarbeitung. Es werden stattdessen die naturanalogen Verfahren nach physikalischen und biologischen Modellen betrachtet.

3.2.2 Problemüberblick

Es gibt zwei große Gruppen von Problemen, auf die Optimierungsverfahren angewendet werden. Zum einen existiert die Gruppe der mathematischen Funktionen (vgl. [Wei95]), zum anderen meist nur umgangssprachlich definierte Probleme aus der Praxis.

Mathematische Funktionen Die folgenden Beispiele für mathematische Funktionen dienen weniger zur Beschreibung konkreter Probleme als zur Beurteilung des Verhaltens von Optimierungsverfahren. Hierfür ist zudem die Kenntnis des Minimums hilfreich.

3.2.2.1 reelle Funktionen

Es gilt: $\vec{x} \in \mathbb{R}^n$

- Schwefelfunktion ([var94] 189 ff)

$$F_1(\vec{x}) = \sum_{i=1}^{n} \left(\sum_{j=1}^{i} x_j \right)^2 ; \qquad \mathrm{Min}(F_1(\vec{x})) = F_1(\vec{0})$$

- Summe verschiedener Potenzen ([var94] 189 ff)

$$F_2(\vec{x}) = \sum_{i=1}^{n} |x_i|^{i+1}; \qquad \mathrm{Min}(F_2(\vec{x})) = F_2(\vec{0})$$

- Achsenparallele Hyperellipsoide ([var94] 189 ff)

$$F_3(\vec{x}) = \sum_{i=1}^{n} (i \cdot x_i)^2; \qquad \mathrm{Min}(F_3(\vec{x})) = F_3(\vec{0})$$

- Hypersphäre ([var94] 199 ff) / Sphärenmodell ([var94] 428 ff)
  $F_4(\vec{x}) = \sum_{i=1}^{n} x_i^2$; $\quad \text{Min}(F_4(\vec{x})) = F_4(\vec{0})$ (Abb. 3.1).

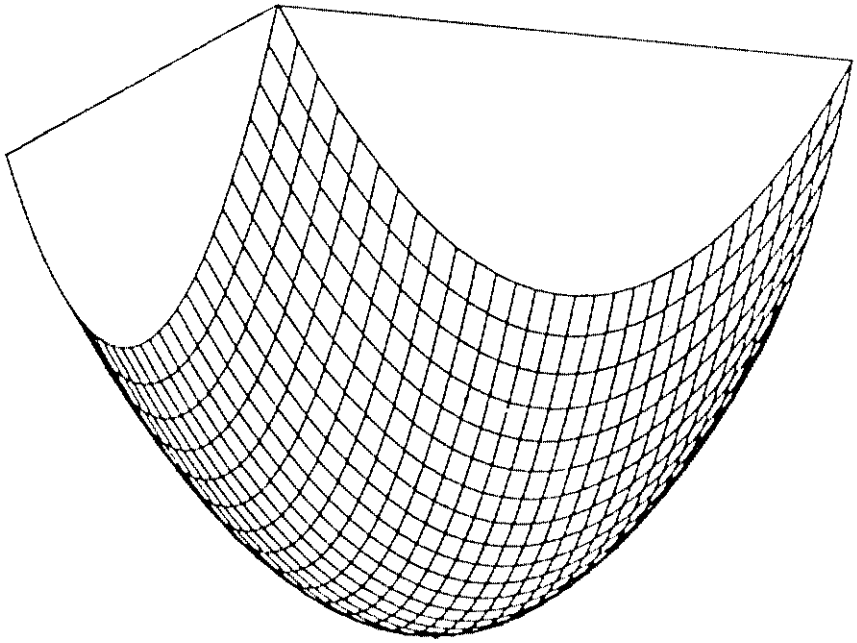


Abbildung 3.1: F_4 – Hypersphäre/Sphärenmodell

- Verallgemeinerte Rosenbrock Funktion ([var94] 189 ff, 199ff, 249 ff)

$$F_5(\vec{x}) = \sum_{i=1}^{n} \left(100 \cdot (x_{i+1} \Leftrightarrow x_i^2)^2 + (1 \Leftrightarrow x_i)^2\right)$$

  $\text{Min}(F_5(\vec{x})) = F_5(\vec{1})$; $\quad \Leftrightarrow 5.12 \leq x_i \leq 5.12$. (Abb. 3.2).

- Griewank's Funktion ([var94] 199 ff, 249 ff)

$$F_6(\vec{x}) = \sum_{i=1}^{n} \frac{x_i^2}{4000} \Leftrightarrow \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

  $\text{Min}(F_6(\vec{x})) = F_6(\vec{0})$, mit $n = 100, \Leftrightarrow 600 \leq x_i \leq 600$.

- Rastingin Funktion ([var94] 249 ff)

$$F_7(\vec{x}) = 3.0 \cdot n + \sum_{i=1}^{n} x_i^2 \Leftrightarrow 3.0 \cdot \cos(2 \cdot \pi \cdot x_i)$$

  $\text{Min}(F_7(\vec{x})) = F_7(\vec{0})$, mit $n = 20, \Leftrightarrow 5.12 \leq x_i \leq 5.12$.

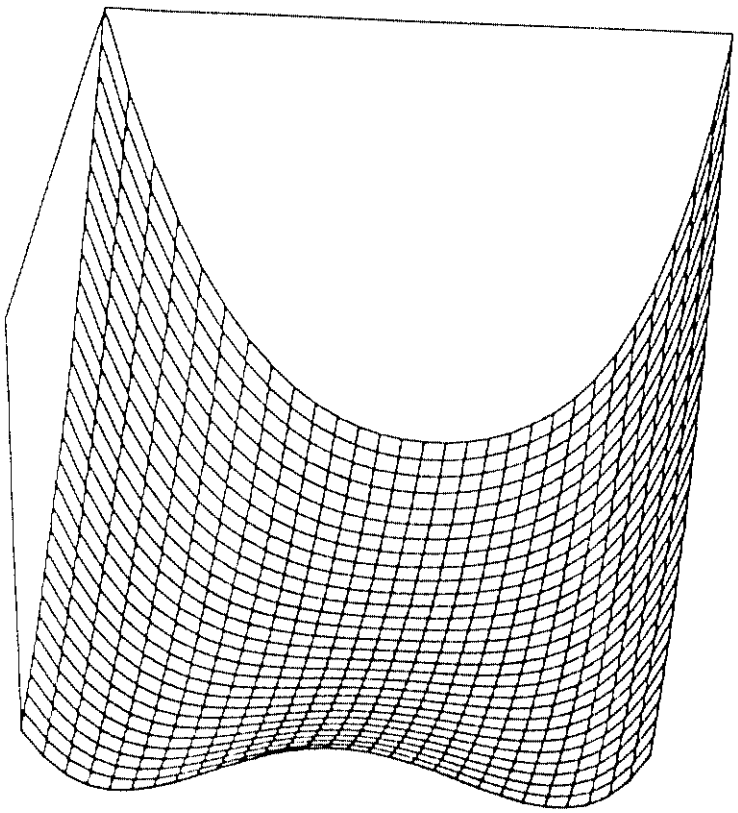


Abbildung 3.2: F_5 – Verallg. Rosenbrock Funktion

- andere Schwefel Funktion ([var94] 249 ff)

$$F_8(\vec{x}) = 418.9829 \cdot n + \sum_{i=1}^{n} x_i \cdot \sin\left(\sqrt{|x_i|}\right)$$

  $\text{Min}(F_8(\vec{x})) = F_8(420.9687, 420.9687, \ldots)$, mit $n = 10, \Leftrightarrow 500 \leq x_i \leq 500$.

- Ackley Funktion ([var94] 249 ff)

$$\begin{aligned} F_9(\vec{x}) &= 20 + e \Leftrightarrow 20 \cdot \exp\left(\Leftrightarrow 0.2 \cdot \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) \\ &\Leftrightarrow \exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2 \cdot \pi \cdot x_i)\right) \end{aligned}$$

  $\text{Min}(F_9(\vec{x})) = F_9(\vec{0})$, mit $n = 30, \Leftrightarrow 30.0 \leq x_i \leq 30.0$.

- Stufenfunktion ([var94] 428 ff)

$$F_{10}(\vec{x}) = 6 \cdot n + \sum_{i=1}^{n} |x_i| \qquad \text{Min}(F_{10}(\vec{x})) = F_{10}(\vec{0}), \text{ mit } n = 5.$$

- Quadratic + Noise ([var94] 428 ff)

$$F_{11}(\vec{x}) = \sum_{i=1}^{n} i \cdot x_i^4 + gauss(0,1)$$

  $\text{Min}(F_{11}(\vec{x})) = F_{11}(\vec{0})$, mit $n = 30, gauss(0,1) =$ Zufallszahl $\in [0,1] \subset \mathbb{R}$. (Abb. 3.3)

- Shekek's Foxholes ([var94] 428 ff)

$$\frac{1}{F_{12}(\vec{x})} = \frac{1}{K} + \sum_{j=1}^{25} \frac{1}{c_j + \sum_{i=1}^{n}(x_i \Leftrightarrow a_{ij})^6}$$

  $\text{Min}(F_{12}(\vec{x})) =?$, mit $n = 2, K, c_j, a_{ij} =??$. Vermutlich berechenbar in Abh. von $K, \vec{c}, (a_{ij})$. (Abb. 3.4)

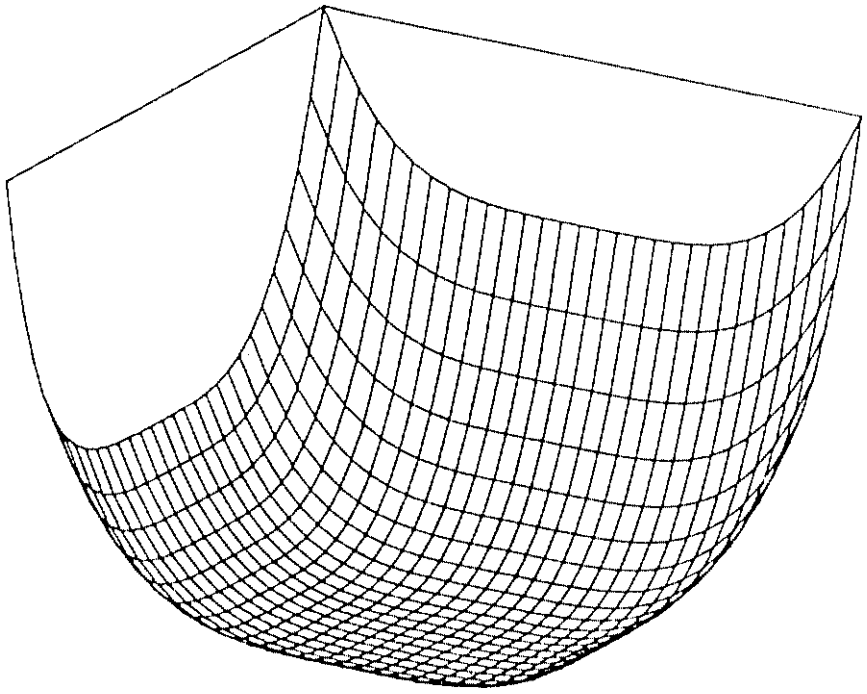


Abbildung 3.3: F_{11} – Quadratic + Noise

- Sphären-Wechsel-Funktion ([var94] 428 ff)

$$F_{13}(\vec{x}(t)) = \begin{cases} \sum_{i=1}^{n} x_i^2(t) & t \bmod a \text{ even} \\ \sum_{i=1}^{n} (x_i(t) \Leftrightarrow b)^2 & t \bmod a \text{ odd} \end{cases}$$

  $\text{Min}(F_{13}(\vec{x})) =?$, mit $n = 30$ und $a, b =?$. Berechenbar in Abhängigkeit von a und b.

- Rechenberg Funktion ([var94] 199 ff)

$$F_{14}(\vec{x}) = \sum_{i=1}^{20} \left((100 \Leftrightarrow i) \cdot \exp\left(\Leftrightarrow \sum_{k=1}^{n} \left(\frac{x_k \Leftrightarrow z_{30 \cdot i + k}}{\sigma} \right)^2 \right) \right)$$

  wobei $z_j = (32 \cdot z_{j-1} + 13(i+1)) \bmod 31, z_0 = 1; \qquad \Leftrightarrow 100 \leq x_i \leq 100.$

- De Jong's 3. Testfunktion
  $F_{15}(\vec{x}) = \sum_{i=1}^{n} integer(x_i)$ (Abb. 3.5)

- C-Funktion (Prof. Claus)

$$C : S_n \rightarrow \mathbb{R}^+$$

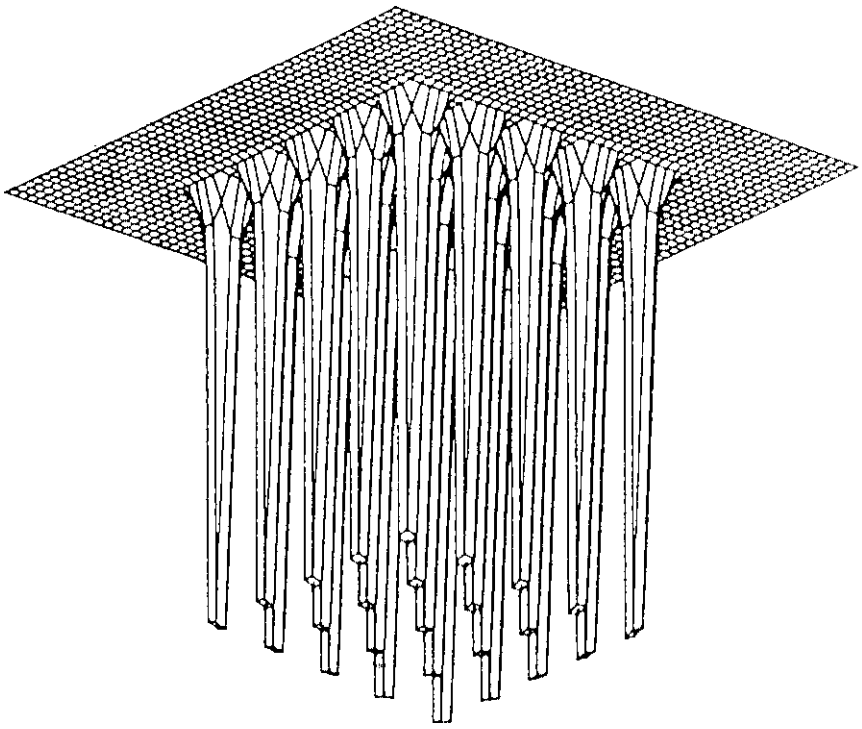


Abbildung 3.4: F_{12} – Shekek's Foxholes

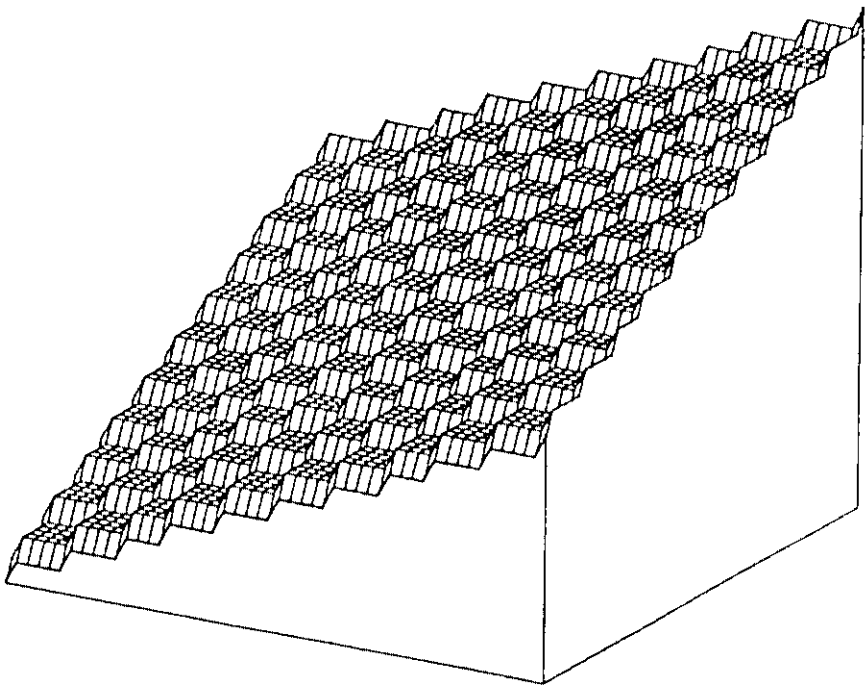


Abbildung 3.5: F_{15} – De Jong's 3. Testfunktion

$$\pi \quad \mapsto \quad \sum_{\substack{i,j=1 \\ i \neq j}} \left| \frac{\pi_j \Leftrightarrow \pi_i}{j \Leftrightarrow i} \right| = 2 \cdot \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{|\pi_j \Leftrightarrow \pi_i|}{j \Leftrightarrow i}$$

3.2.2.2 Integer-Funktionen

- $$F_{16}(\vec{x}) = \Leftrightarrow\!\!\!\Rightarrow ||\vec{x}||_1, \quad \vec{x} \in \mathbb{Z}^{30}, \quad \text{Min}\{F_{16}(\vec{x})\} = F_{16}(\vec{0}).$$

- $$F_{17}(\vec{x}) = \Leftrightarrow\!\!\!\Rightarrow \vec{x}^T \cdot \vec{x}, \quad \vec{x} \in \mathbb{Z}^{30}, \quad \text{Min}\{F_{17}(\vec{x})\} = F_{17}(\vec{0}).$$

- $$F_{18}(\vec{x}) = \prod_{i=0}^{18} \left[1 \Leftrightarrow (1 \Leftrightarrow x_i)^{x_i} \right], \quad \vec{x} \in \mathbb{Z}^{18}, \quad x_i \geq 0.$$

3.2.2.3 Praktische Probleme

1. Stundenplan–Problem

   Quelle: [var94] 557

   Gegeben ist eine Menge von Ereignissen $E = \{e_1, e_2, \ldots, e_v\}$ und eine Menge von Zeiten $T = \{t_1, t_2, \ldots, t_s\}$, sowie oft eine Menge von Orten $P = \{p_1, p_2, \ldots, p_m\}$ und/oder Ausführenden $A = \{a_1, a_2, \ldots, a_n\}$.

   Ein Auftrag ist ein 4–Tupel (e, t, p, a) mit $e \in E, t \in T, p \in P, a \in A$, wobei man dies wie folgt interpretieren kann: „Ereignis e findet zur Zeit t am Ort p statt, durchgeführt von a." Dies kann eine Vorlesung beschreiben, aber auch einen Fertigungsprozeß.

   Das zu lösende Problem ist nun, einen Stundenplan zu finden, der es ermöglicht, alle Ereignisse auszuführen. Prinzipiell wäre somit eine *Erfüllung* der Anforderungen ausreichend, allerdings kann auch eine *Bewertung* eines Plans (nach anwendungsspezifischen Kriterien) vorgenommen werden, wodurch nach einem optimalen Stundenplan gesucht werden kann.

   Es ist offensichtlich, daß z.B. zur selben Zeit am selben Ort nicht zwei Ereignisse stattfinden können; ggf. sind aber zusätzliche Rahmenbedingungen zu berücksichtigen, z.B.:

   - Zwischen zwei Ereignissen soll ein Zeitabstand existieren.
   - Bestimmte Ereignisse dürfen zu gewissen Zeiten nicht stattfinden.
   - Ein Ereignis soll zu einem bestimmten Zeitpunkt stattfinden.
   - Relationen zwischen Ereignissen sollen eingehalten werden. Z.B. soll ein bestimmtes Ereignis vor einem anderen stattfinden.
   - Ereignisse seien an eine Auswahl von Orten gebunden.

2. Eisenbahn–Fahrplan Problem

   Quelle: [var94] 566 ff

   Gegeben ist die Beschreibung eines Eisenbahnnetzes. So gibt es Bahnhöfe und Züge in verschiedenen hierarchischen Klassen, z.B. „Intercity" (IC), „Interregio" (IR) und „Regionalbahn" (RB). Züge halten nur in Bahnhöfen mit derselben oder höheren Klasse, d.h. RB–Züge in allen. Außerdem werden die Strecken erfaßt, d.h. welche Bahnhöfe miteinander verbunden sind.

Gesucht wird nun ein Zeitplan, der einige „weiche“ und „harte“ Einschränkungen erfüllt.

Harte Einschränkungen legen fest, daß ein Zugpaar[1] einen gemeinsamen Abschnitt mit einer gewissen Frequenz bedienen muß. Sie setzen einen Bereich für die Umsteigezeit zwischen zwei Zügen an einem bestimmten Bahnhof fest. Züge müssen zudem einen Mindestabstand von drei Minuten auf derselben Strecke einhalten. Schließlich muß noch erreicht werden, daß die Züge sinnvolle Haltezeiten einhalten und bei Verwendung eines Modells mit Stundentakten (d.h. ohne Beachtung der Stunde) die Abfahrtszeiten zwischen 0 und 59 Minuten liegen.

Alle Zeitpläne müssen die harten Einschränkungen erfüllen, d.h. ein Zeitplan, der diese nicht erfüllt, wird niemals als Lösung akzeptiert werden. Um nun die Fahrpläne, die die harten Einschränkungen erfüllen, vergleichen zu können, bestimmen die weichen Einschränkungen die Qualität eines Zeitplans.

Weiche Einschränkungen sind z.B., die Haltezeiten so gering wie möglich zu halten oder günstige Umsteigemöglichkeiten anzubieten. Somit ist es über die weichen Einschränkungen möglich, die Qualität eines Fahrplans zu bewerten – dies ist die Voraussetzung bei der Suche nach der *optimalen* Lösung.

3. n–Damen Problem

   Quelle: [var94] 48, Abb. 3.6 – Beispiel mit $n = 6$

   Gegeben ist ein Schachbrett mit $n * n$ Feldern. Nun sollen auf diesem Feld n Damen so plaziert werden, daß keine Dame eine andere schlagen kann. Hier geht es also nicht um eine *Optimierung*, sondern nur um die *Erfüllung* einer Bedingung und *eine* Lösung reicht aus.

   Laut den Schachregeln kann eine Dame senkrecht, waagrecht und diagonal beliebig viele Felder vor- oder zurückziehen. Somit kann man die Beschreibung einer Belegung vereinfachen: Zwei Damen können nicht in derselben Reihe des Bretts stehen; man muß also nur die Spalte notieren. Daher reicht ein n–Tupel der Form $(x_1, x_2, \ldots, x_n)$; $x_i \in \{1, 2, \ldots, n\}$ aus. Außerdem kann die Diagonalenbedingung als $|x_i \Leftrightarrow x_j| \neq |i \Leftrightarrow j|, \quad i \neq j$ notiert werden, was den Suchraum weiter einschränkt.

4. Travelling Salesman Problem - TSP

[1] Auf zweigleisigen Strecken fährt jeweils ein Zug in jede Richtung.

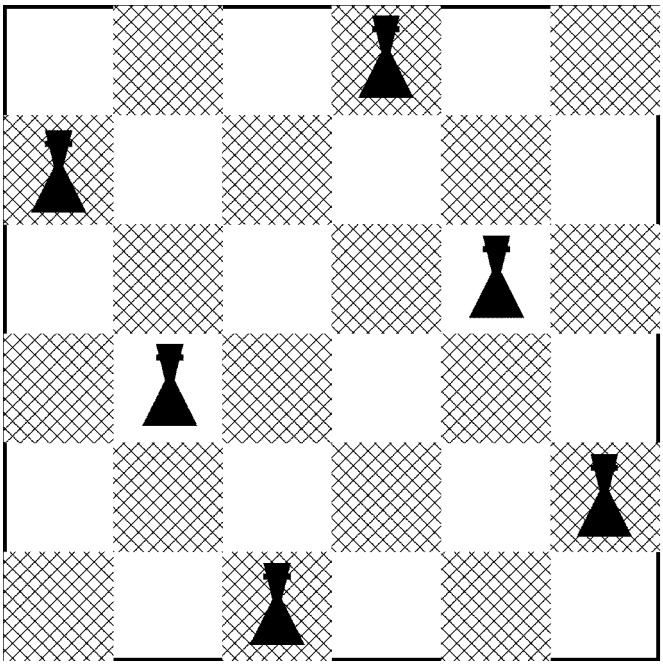


Abbildung 3.6: n–Damen Problem

Quelle: [VC94] 375 f, Abb. 3.7 – Beispiel mit $n = 5$, optimaler Weg

Gegeben sind n Knoten $k_1, k_2, \ldots, k_n$ die durch ungerichtete Kanten vollständig verbunden sind. Jeder Kante (k_i, k_j) ist ein Gewicht $d_{ij} \in \mathbb{R}$ zugeordnet.

Gesucht ist nun die Anordnung $k_{i_1}, k_{i_2}, \ldots, k_{i_n}$ mit $i_1 = 1$ (o.B.d.A. sei der Startknoten festgelegt) und $\forall j \in \{1, \ldots, n\}\ \exists!\ i_m : i_m = j$ (d.h. jeder Knoten wird einmal besucht) für die gilt:

$$\sum_{k=1}^{n--1} d_{i_k i_{k+1}} + d_{i_n i_1} \quad \text{ist minimal.} \tag{3.1}$$

5. n–Personen Spiel mit eingeschränkter Interaktion

   Quelle: [var94] 514 ff

   Gegeben ist eine Menge von n Spielern $N = \{1, 2, \ldots, n\}$, eine Menge $S_k = \{s_{k1}, \ldots, s_{km}\}$ von Strategien für jeden Spieler $k \in N$ und eine Erfolgsfunktion („payoff function“) $u_k : S_k \to \mathbb{R}$ und:

$$(s_k, s_{k1}, s_{k2}, \ldots, s_{kn_k}) \mapsto u_k(s_k, s_{k1}, s_{k2}, \ldots, s_{kn_k}) \tag{3.2}$$

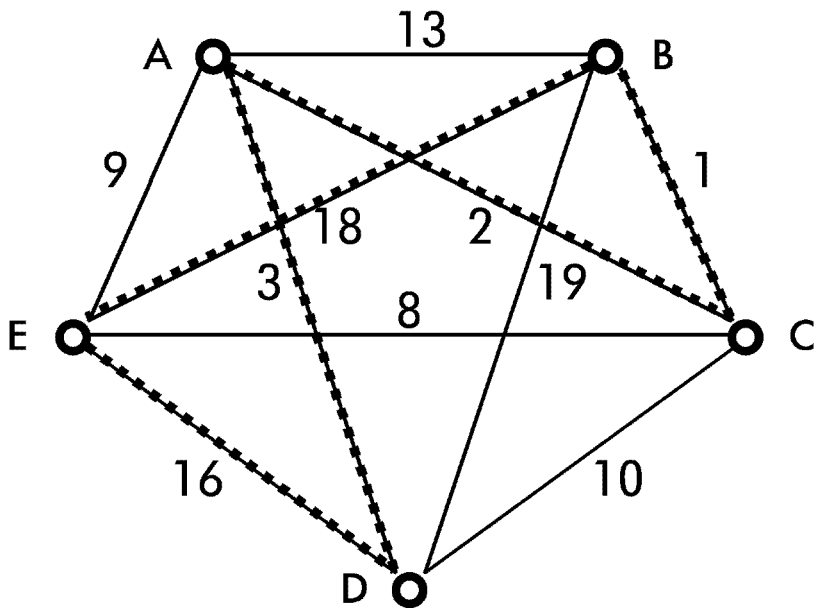


Abbildung 3.7: Travelling Salesman Problem

die nur von den Strategien einer beschränkten Anzahl von Spielern abhängt: der eigenen s_k und der der n_k Nachbarn.

Das Spiel kann nun als gerichteter Graph $G =< V, E >$ dargestellt werden, dem sog. „Interaktionsgraphen“. V, die Menge der Knoten, symbolisiert die Spieler, die Kanten E repräsentieren Muster der Interaktion zwischen den Spielern. So zeigen die von Spieler k ausgehenden Kanten auf die Spieler, deren Erfolg von k beeinflußt wird, wie die auf Spieler k zielenden Kanten die Spieler definieren, deren Strategie den Erfolg des Spielers k beeinflußt.

Gesucht ist nun ein optimaler Ablauf des Spiels, wobei aus der Quelle die Definition der Optimalität nicht eindeutig hervorgeht. Denkbar ist z.B., daß ein Spieler das absolute Optimum erreichen soll oder der mittlere Erfolg aller Spieler möglichst groß ist.

6. Steiner–Netz

   Quelle: [var94] 197, Abb. 3.8 – Beispiel

   Gegeben ist eine Menge von festen Punkten h_i in einem kartesischen Koordinatensystem. Diese Punkte sollen nun durch ein Netz verbunden

werden, wobei n Gabelungspunkte[2] eingeführt werden dürfen, die sog. „Steiner–Punkte“. Diese Punkte können mit drei festen oder Steiner–Punkten verbunden werden, sie sind zudem verschiebbar. Ziel ist es nun, die Länge des Netzes, d.h. die Gesamtlänge der Verbindungskanten, zu minimieren. Daher wurde ein kartesisches Koordinatensystem vorausgesetzt.

Man kann sich das Problem wie folgt vorstellen: Auf einer Insel stehen Häuser h_i, die mit Wegen verbunden werden sollen. Gesucht ist nun das kürzeste Netz, d.h. dasjenige, dessen Weglänge am geringsten ist. Dabei können Wege auch Gabelungen haben, oben „Steiner–Punkte“ genannt.

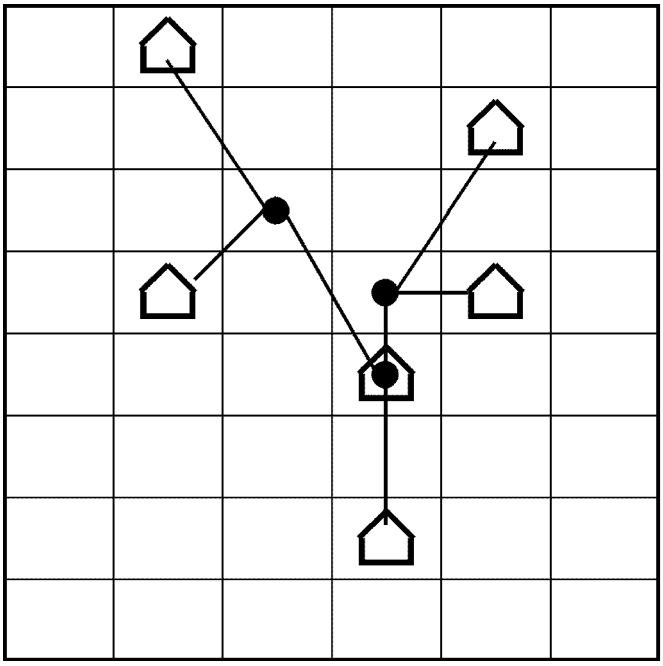


Abbildung 3.8: Steiner–Netz

7. Gewichtetes Graphzweiteilungsproblem

   Quelle: [var94] 618

   Gegeben ist ein Graph (V, E, w) mit Knotenmenge V, die $2n$ Knoten enthält, der Kantenmenge E und einer Gewichtsfunktion $w : E \to \mathbb{R}$, die jeder Kante ein Gewicht zuordnet. Dieser Graph soll in zwei

[2]Die Quelle ist in der Beschreibung nicht eindeutig!

gleichgroße, disjunkte Teilgraphen geteilt werden, wobei gelte: $A = \{a_1, a_2, \ldots, a_n\},\ B = \{b_1, b_2, \ldots, b_n\},\ A, B \subset V,\ A \cap B = \{\}$ und $A \cup B = V$.

Ziel ist nun, die Summe der Gewichte der Verbindungskanten zwischen den Teilgraphen zu minimieren:

$$C(\{A, B\}) := \min_{A,B} \sum_{a_i \in A} \sum_{b_j \in B} w((a_i, b_j)) \tag{3.3}$$

8. Rucksackproblem

   Quelle: [VC94] 418

   Hier wird versucht, einen Rucksack so zu füllen, daß dieser zum einen möglichst voll ist, andererseits der Wert der eingepackten Gegenstände möglichst hoch ist.

   Gegeben sind also n Gegenstände, denen jeweils ein Gewicht g_i und ein Wert $w_i,\ i \in \{1, 2, \ldots, n\}$ zugeordnet ist. Gegeben ist weiterhin eine Ladekapazität G. Gesucht ist nun eine Indexmenge $I \subset \{1, 2, \ldots, n\}$, für die gilt:

$$\left(\sum_{i \in I} w_i \ \text{ ist maximal} \right) \wedge \left(\sum_{i \in I} g_i \leq G \right). \tag{3.4}$$

   Diese Beschreibung läßt sich auf viele praktische Probleme anwenden. So können z.B. statt der Gewichtswerte auch Volumengrößen verwandt werden; bei Betrachtung von Flächen handelt es sich um das „Plazierungsproblem“.

9. Cliquenproblem

   Quelle: [VC94] 97

   Gesucht wird hierbei in einem ungerichteten Graphen eine *Clique* von k Knoten, die jeweils paarweise durch eine Kante verbunden sind.

   Gegeben ist also ein ungerichteter Graph $G = (V, E)$, mit Knotenmenge V und Kantenmenge E. Gesucht wird nun die *größte Teilmenge* $V' \subset V$ mit k Elementen, für die gilt:

$$\forall v, w \in V' : (v, w) \in E. \tag{3.5}$$

10. Erfüllbarkeitsproblem

    Quelle: [VC94] 192

    Hierbei wird versucht, für einen gegebenen Booleschen Ausdruck eine Belegung zu finden, die den Ausdruck erfüllt.

    Beispiel Für $B = (\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_4) \wedge x_2$ soll eine Belegung der Variablen $x_1, \ldots, x_4;\ x_i \in \{true, false\}$ gefunden werden, so daß gilt: $B \equiv true$.

3.2.3 Optimierungsverfahren

Um Lösungen für diese und andere schwere Probleme zu finden, reichen herkömmliche Methoden oft nicht aus, da sie zu schlecht, zu langsam oder zu aufwendig sind. Hervorzuheben sind dabei v.a. die NP–vollständigen/harten Probleme, zu denen z.B. TSP (s.o.) gehört:

Beispiel Ein Handelsvertreter plant seine Kundenbesuche. Er sucht nun die schnellste/kostengünstigste Route, die ihn wieder an den Ausgangspunkt führt. Seien nun 10 Besuche vorgesehen. Somit gibt es $9! = 362.880$ mögliche Rundreisen; der Startort sei festgelegt. Müßte der Vertreter aber nur fünf Kunden mehr besuchen, so stiege die Anzahl der zu untersuchenden Möglichkeiten auf $14! = 87.178.291.200$. Schon dieses kleine Beispiel macht deutlich, daß eine vollständige Durchsuchung des Problemraums nicht praktikabel ist.

Um dennoch brauchbare Lösungen zu finden, wurden verschiedene Algorithmen entwickelt. Zum einen gibt es die sog. Hill–Climbing–Verfahren, die in der Regel allerdings nur das dem Startpunkt nächste *lokale* Minimum finden. Es gibt hier also keine Garantie, daß nicht an anderer Stelle ein besserer Wert erreicht werden kann. Andererseits gibt es die naturanalogen Algorithmen, die das *globale* Minimum suchen.

Hierbei muß beachtet werden, daß der hohe Aufwand dieser Strategien nur bei Zutreffen folgender Bedingungen gerechtfertigt ist:

- Der Problemraum ist deutlich zu groß, um ihn einfach mit den verfügbaren Rechnern zu durchsuchen. Dies trifft besonders bei einer großen Anzahl von Variablen zu.
- Das Problem läßt sich nicht mit üblichen mathematischen Verfahren analysieren und es existieren auch keine ausreichend genauen (einfacheren) Näherungen.

- Es existieren keine herkömmlichen heuristischen Verfahren, die hinreichend gute Lösungen liefern.
- Das Problem ist NP–hart.

In allen Fällen werden eine oder mehrere Start–Konfigurationen, d.h. Belegungen der freien Variablen, zufällig gewählt und diese solange modifiziert, bis eine Endbedingung erfüllt ist.

3.2.3.1 Kalkülbasierte Verfahren – Hill–Climbing

Quelle: [Sch81] 20 - 86

Kann ein Problem durch eine mathematische Funktion dargestellt werden, so bietet sich an, eine Lösung durch ein sog. „Hill–Climbing“ zu finden. Es wird hier ein *lokales* Extremum der Funktion, „Zielfunktion“ genannt, gesucht. Hierbei spielt es keine Rolle, ob ein Maximum oder ein Minimum gesucht ist, da gilt:

$$\mathrm{Max}\{f(x)\} = \Leftrightarrow \mathrm{Min}\{\Leftrightarrow f(x)\}. \tag{3.6}$$

Handelt es sich bei der Zielfunktion um eine leicht analysierbare Funktion, so ist es natürlich nicht erforderlich, eine Hill–Climbing–Strategie zu verwenden. Folgende Eigenschaften verhindern allerdings einen solchen Ansatz:

- Extrema sind nicht mathematisch ermittelbar, da die Funktion z.B. unstetig ist. Dies ist besonders bei Funktionen der Fall, die nicht glatt (smooth) sind.
- Ableitungen sind nicht berechenbar.
- Lösungskandidaten sind nicht immer vom gewünschten Typ, sondern Minimum, Maximum oder Sattelpunkt. Hierdurch wird eine weitere Untersuchung nötig.
- Die zu betrachtenden Gleichungssysteme sind nicht direkt lösbar.

Für die folgende Liste von Strategien wird in der Quelle [Sch81] das Problem $\min_{\vec{x}}\{F(\vec{x})|\vec{x} \in \mathbb{R}^n\}$ betrachtet, d.h. die Suche nach einem Minimum. Dabei kann noch zwischen „starken“ und „schwachen“ Minima unterschieden werden. Letztere besitzen eine Umgebung, deren Werte nur um $\varepsilon > 0$ höher sind; es handelt sich dann nicht um einzelne Punkte. Außerdem kann ein Minimum lokal oder global sein, allerdings bieten diese Verfahren keine Garantie für die Art des Minimums. (Dies gilt natürlich nicht für „unimodale“

Funktionen, da hier lokale Minima auch globale Minima sind und somit keine Unterscheidung existiert.)

1. Eindimensionale Suche
   Gibt es nur eine zu betrachtende Variable, so spricht man von eindimensionaler Suche. Die hier gefundenen Strategien können z.T. auf die Lösung von Problemen mit n Variablen erweitert werden.

   Um nun eine gute Lösung zu finden, gibt u.a. drei Typen von Verfahren:

   - Simultane Methoden [Sch81] 23: Es werden (bevorzugt parallel) Werte aus dem Definitionsbereich ausprobiert. Bei „Gitter–Methode“ wird ein Intervall in gleichgroße Teile zerlegt und jeweils ein Funktionswert aus dem Intervall berechnet.

   - Näherungen: Polynome inkl. ihrer Ableitungen nähern die zu untersuchende Funktion an und erlauben das Ermitteln einer Lösung. Das Problem ist hier, eine ausreichend genaue Näherungsfunktion zu finden.

   - sequentielle Methoden: Diese Verfahren eignen sich besonders für sequentiell arbeitende Rechner. In jeder Iteration wird ein neuer Punkt ermittelt, der das Minimum weiter annähern soll. Dies wird solange wiederholt, bis eine bestimmte Anzahl von Iterationen durchlaufen und so das Minimum hinreichend genau angenähert wurde.

     ```
     Wähle Startwert x_0, Startintervall [a_0, b_0]
     repeat
       Berechne x_{i+1}, a_{i+1}, b_{i+1}
     until  Abbruchbedingung
     ```

   Beispiele für sequentielle Methoden zur Suche eines Minimums $x \in \mathbb{R}$ mit $\forall y \in \mathbb{R} : f(y) \geq f(x)$ sind:

   (a) Verfahren, die *einen Punkt* x_i betrachten und aus diesem einen neuen Punkt x_{i+1} berechnen.

   - „Boxing the minimum“ [Sch81] 25: Man bestimmt einen Startwert $x_0 \in \mathbb{R}$, jeder weitere Wert wird über $x_{k+1} = x_k + s, s \in \mathbb{R}$ ermittelt. Dies wird solange wiederholt, bis gilt $f(x_{k+1}) > f(x_k)$. Danach wird die Schrittweite verkleinert und in der anderen Richtung gesucht. Dieser Algorithmus kann verbessert werden, z.B. indem die Schrittweite im Falle von $f(x_{k+1}) <$

$f(x_k)$ verdoppelt wird, andernfalls jedoch halbiert (und negiert). Dieses Verfahren ist brauchbar, wenn das Intervall nicht bekannt ist, in dem das Minimum liegt (oder erst bestimmt werden soll). Andernfalls existieren bessere Algorithmen.

- Newton–Raphson - Iteration [Sch81] 32: Umsetzung des Newton–Verfahrens zur Bestimmung einer Nullstelle von $f(x)$. (Somit muß die Zielfunktion umformuliert werden, d.h. es wird z.B. die erste Ableitung betrachtet.)

$$x_{k+1} = x_k \Leftrightarrow \frac{f(x_k)}{f'(x_k)} \tag{3.7}$$

(b) Verfahren, die ein *Intervall* $[a_i, b_i]$ betrachten. In jeder Iteration werden neue Werte a_{i+1} und b_{i+1} für die Grenzen ermittelt (Abb. 3.9).

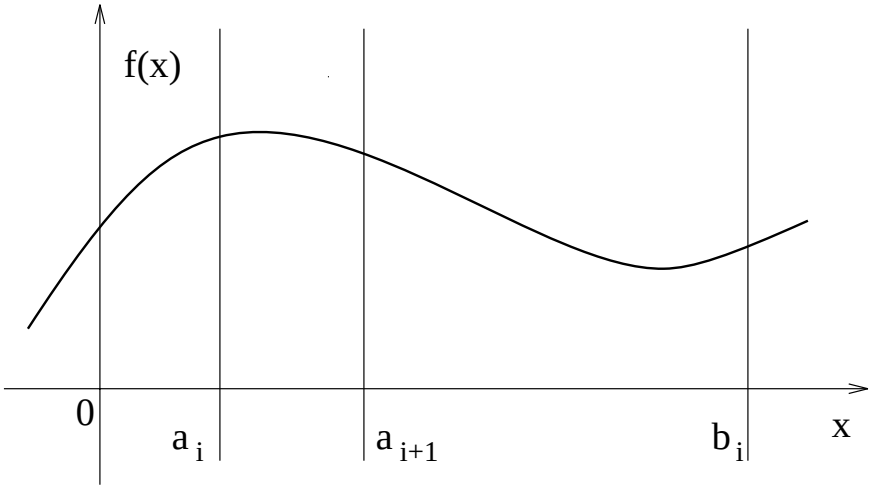


Abbildung 3.9: Intervall–Verfahren

- Elimination [Sch81] 26: die Intervallgröße wird mit $\alpha < 1$ multipliziert und diejenige Grenze verschoben, die weiter vom Minimum entfernt ist.

  Ausgehend vom Intervall $[a, b]$ werden die Grenzen aufeinander zu verschoben, wobei die Schrittweite der Quotient zweier Fibonacci–Zahlen ist. Schließlich wird das Intervall ausgesucht, in dem sich das Minimum befindet. Welches dies ist, wird festgestellt, indem der Funktionswert an den möglichen

neuen Intervallgrenzen innerhalb des alten Intervalls berechnet wird.

Bei der Variante „Fibonacci–Division“ ist α der Quotient zweier Fibonacci–Zahlen; eine andere Variante verwendet einen konstanten Wert, wodurch die Berechnung der Fibonacci–Zahlen entfällt.

- „Regula–falsi“–Iteration [Sch81] 31: Nach Start mit dem Intervall $[a_k, b_k]$ wird eine neue Grenze c_k

$$c_k = a_k \Leftrightarrow f(a_k) \frac{b_k \Leftrightarrow a_k}{f(b_k) \Leftrightarrow f(a_k)} \tag{3.8}$$

  bestimmt, die zu a_{k+1} (oder b_{k+1}) wird.

(c) Interpolationsprozeduren [Sch81] 31
Es wird eine einfache Funktion durch einige Punkte der Zielfunktion gelegt und das Minimum dieser Hilfsfunktion berechnet. Bei einfachen Hilfsfunktionen kann diese Stelle direkt aus den Funktionswerten der Zielfunktion ermittelt werden; sie ersetzt dann einen der bisher verwendeten Punkte, dann wird das Verfahren wiederholt.

- Lagrange–Interpolation [Sch81] 33: Für dieses Verfahren wird nur die Zielfunktion selbst, aber keine ihrer Ableitungen benötigt. Ein Polynom p–ter Ordnung wird durch $p + 1$ Punkte der Zielfunktion gelegt. Dann wird die Minimalstelle des Polynoms ermittelt; für $p = 2$ kann dies direkt aus den Funktionswerten geschehen. Der Funktionswert der Zielfunktion an dieser Stelle wird berechnet und abhängig vom Ergebnis ersetzt sie einen der alten Punkte.

- Hermitische Interpolation [Sch81] 36: Ein Polynom 3. Grades wird als Test–Funktion verwendet, allerdings wird zudem die erste Ableitung $f'(x)$ benötigt. Daher reichen zwei Punkte der Zielfunktion aus.

  Für alle Interpolationsverfahren gilt: Je größer die Übereinstimmung der Test–Funktion mit der Zielfunktion ist, desto schneller und verläßlicher konvergieren die Verfahren.

2. Multidimensionale Strategien
Die meisten Probleme lassen sich nur durch Funktionen mit $n > 1$

Variablen ausdrücken. Somit ist die Anzahl der Funktionswerte etwa $O(N^n)$ groß, wobei N die Anzahl der Werte darstellt, die eine Variable annehmen kann. Dies macht es offensichtlich unmöglich, ein Minimum durch Ausprobieren zu finden.

Um trotzdem ein Minimum zu finden, wurden diverse Algorithmen entwickelt. Diese unterscheiden sich zum einen in der Art der benötigten Information. So gibt es „direkte“ Verfahren, die mit der Zielfunktion allein auskommen. Die „Gradienten“–Verfahren benötigen dagegen die erste(n) Ableitung(en) $(\nabla f(x), \nabla^2 f(x), \ldots)$.

Allgemein wird eine Rekursion folgenden Zuschnitts durchlaufen:

$$x_{k+1} = x_k + s_k \cdot v_k; \qquad x_k, v_k \in \mathbb{R}^n, s_k \in \mathbb{R}. \tag{3.9}$$

Hierbei ist s_k die Schrittlänge und v_k die Suchrichtung.

Der Vorteil der direkten Methoden ist ihre Einfachheit und die positive Erfahrung in der Anwendung. Man „rät“ einen vielversprechenden Punkt und überprüft, ob er das Minimum besser annähert; im negativen Fall wird ein anderer Punkt ermittelt. Allerdings sind die Resultate der direkten Methoden oft schlechter als die der Gradienten–Methoden.

- Koordinaten–Strategie [Sch81] 40: Von einem Startpunkt aus wird parallel zu den Koordinatenachsen der nächste Punkt ermittelt. Am einfachsten geht man reihum in alle Richtungen, als Verbesserung wird jedoch eine vielversprechende Richtung beibehalten. Dies kann z.B. durch die Verwendung einer eindimensionalen Strategie für eine Variable geschehen.

  Natürlich beeinflußt hier die Schrittweite den Erfolg des Verfahrens beträchtlich. Ist sie zu groß gewählt, so kann das Minimum nicht genau ermittelt werden. Ist sie andererseits zu klein, so erhöht sich die Anzahl der nötigen Versuche. Außerdem kann diese Methode zu prinzipiellen Problemen führen, da nur parallel zu den Achsen gesucht wird.

- Hooke, Jeeves – Mustersuche [Sch81] 43: Bei dieser Variante der Koordinaten–Suche werden zwei Arten von Schritten vorgenommen. Zum einen gibt es Erforschungsschritte, d.h. es wird der Wert der Zielfunktion an neuen Stellen ermittelt. Zum anderen wird abhängig vom Erfolg der einzelnen Tests eine Extrapolation in die vielversprechendste Richtung vorgenommen.

- Rosenbrock – „rotating coordinates“ [Sch81] 48: Das Koordinatensystem wird in eine bessere Lage rotiert, analog zur Mustersuche. Dieses Verfahren ist sehr robust, braucht keine Ableitungen und kann auf Liniensuche verzichten. Allerdings ist die Rotation des Koordinatensystems mit umfangreichen Matrizenoperationen verbunden, die dieses Verfahren für große n unanwendbar machen.

- Davies, Swann, Campey (DSC) [Sch81] 53: Dieses Verfahren verbindet mehrere einfachere: Es werden Liniensuchen parallel zu allen Koordinatenachsen vorgenommen, dann in Richtung des Gesamterfolgs gewandert, und schließlich das Koordinatensystem entsprechend rotiert. Dieses Verfahren ist effektiv, jedoch nur bei glatten Funktionen und für kleine n anwendbar.

- Nelder, Mead – Simplex–Strategien [Sch81] 57: Hier werden für eine Funktion mit n Variablen $n+1$ Punkte äquidistant ausgewählt. Für $n = 2$ wird somit ein Dreieck verwandt; bei $n = 3$ handelt es sich um einen Tetraeder. Daher der Name „Simplex“. Die Punkte werden als Knoten bezeichnet.

  Nun wird die Zielfunktion an allen Knoten berechnet und der Knoten mit dem höchsten Wert (hier also der schlechteste Punkt) durch seine Spiegelung an der (Hyper–) Fläche der restlichen Punkte ersetzt. Ist dieser Punkt wiederum schlechter, so wird statt des Punktes mit dem höchsten Funktionswert der nächst bessere ersetzt. Somit wird der Körper also einmal um den Knoten mit dem besten Wert rotiert, danach jedoch wird die Kantenlänge des Simplex verkleinert und das Verfahren wiederholt.

- „Complex Strategy of Box“/„constrained simplex“ [Sch81] 59: Diese Variante der Simplex - Strategie verwendet einen Körper mit mehr Knoten und expandiert diesen bei jeder Spiegelung. Außerdem kann dieses Verfahren Randbedingungen einbeziehen; der Startpunkt darf sogar „verboten“ sein, d.h. außerhalb des eigentlichen Lösungsraums liegen.

Neben den direkten Verfahren existieren noch solche, die die partiellen Ableitungen verwenden. Dabei ist jedoch zu beachten, daß das allgemeine Verfolgen einer mehrdimensionalen „Flugbahn“ ein schwereres Problem darstellen kann, als das eigentlich zu lösende ([Sch81] 65). Daher kann nur iterativ gearbeitet werden, nach folgender Formel:

$$x_{k+1} = x_k \Leftrightarrow s_k \frac{\nabla f(x_k)}{|\nabla f(x_k)|}. \tag{3.10}$$

Aus der Verwendung des Gradienten folgt, daß die Zielfunktion stetig partiell ableitbar und die Ableitungen eindeutig sein müssen. Wieder bekommt die Schrittweite s_k eine große Bedeutung, da sie die Anzahl der Tests bestimmt und auch die Konvergenz beeinflußt. Auch hier kann die Liniensuche für Verbesserungen eingesetzt werden.

- Powell – konjugierte Richtungen [Sch81] 69: Es werden linear unabhängige Vektoren $\vec{v}_i$ aus den Gradienten bestimmt, entlang derer eine Liniensuche vorgenommen wird.
- Newton–Strategien [Sch81] 75: Ist die Zielfunktion $f(x)$ beliebig oft differenzierbar, so kann aus den Werten der Funktion und ihrer Ableitungen an der Stelle x_k der Funktionswert an einer anderen Stelle angenähert werden (vgl. Taylor–Reihe). Als Zielfunktion wird nun die einfachere Funktion betrachtet (z.B. eine quadratische) und der Punkt bestimmt, an der die einfache ein Minimum hätte. Diese Stelle wird als x_{k+1} für neue Iterationen benutzt.

  Dieses Verfahren ist jedoch sehr teuer, da zum einen eine große Matrix invertiert werden muß, zum anderen jedoch Gleichungssysteme zu lösen sind ([Sch81] 76). Daher ist dieses Verfahren weniger geeignet für nicht–quadratische Zielfunktionen.

  Allerdings ist dieses Verfahren Basis für diverse Verbesserungen, sog. „quasi–Newton – Strategien“. Stewart entwickelte z.B. eine ableitungsfreie Variante, die zusätzliche Werte der Zielfunktion zum Abschätzen der Steilheit verwendet und so auf die Berechnung der Ableitungen verzichten kann. ([Sch81] 79)

3.2.3.2 Naturanaloge Verfahren

Weitere Strategien zur Lösung schwieriger Probleme sind die naturanalogen Verfahren, die sich physikalische bzw. biologische Vorgänge in der Natur zum Vorbild nehmen. Diese Vorgänge werden simplifiziert, um sie auf allgemeine Probleme anwenden zu können.

3. Physikalische Modelle
   Die folgenden Verfahren wurden in Anlehnung an Vorgänge in der Physik entwickelt. Gemeinsam ist ihnen, daß eine neue Lösung stets übernommen wird, so sie besser als die zuletzt gemerkte ist. Die Algorithmen unterscheiden sich nur in den Umständen, die zur Übernahme einer schlechteren Konfiguration führen. Im allgemeinen Algorithmus

(Abb. 3.10) wird hierzu die Funktion g verwendet, die $\vec{x}_{\text{alt}}$ anhand der alten Belegung, der neuen Belegung, der Fitneßfunktion f und des Parameters T bestimmt. Der Parameter T kann danach mit der Funktion h verändert werden.

```
Wähle Anfangsbelegung x⃗_alt
Wähle Parameter T
repeat
      x⃗_neu := kleine_Änderung(x⃗_alt)
      x⃗_alt := g(x⃗_alt, x⃗_neu, f, T)
      T := h(x⃗_alt, x⃗_neu, f, T)
until Abbruchbedingung
```

Abbildung 3.10: Allgemeiner Algorithmus physikalischer Modelle

- Simulated Annealing (SA) [var90] 445 - 454
  Hier werden die Vorgänge beim Auskühlen einer Schmelze in ein Verfahren zum Finden einer Lösung übertragen: Atome suchen sich beim Abkühlen den energetisch „günstigsten“ Platz. Dabei hängt ihre Bewegungsfähigkeit direkt von der Temperatur ab. Ist die Temperatur hoch, so kann sich ein Atom weit bewegen, bevor es zur Ruhe kommt; es kann insbesondere ein energetisch ungünstigeres Energieniveau durchlaufen, um einen noch besseren Platz zu finden. Je kälter es wird, desto eher bleibt es auf seinem Platz bzw. verändert seinen Platz nur noch zugunsten eines energetisch besseren.

```
Wähle Anfangskonfiguration x⃗_alt
Setze Temperatur T auf Startwert
repeat
   x⃗_neu := kleine_Änderung(x⃗_alt)
   ΔE := Qualität(x⃗_neu) ⇔Qualität(x⃗_alt)
   P([x⃗_alt := x⃗_neu]) := { 1            ΔE≥0
                          { exp(ΔE/T)    sonst
   if    lange keine Verbesserung
         oder zu viele Iterationen
   then  verringere T
until Abbruchbedingung
```

Abbildung 3.11: Simulated Annealing–Algorithmus

Das Modell (Abb. 3.11) startet nun mit einer zufällig gewählten Konfiguration und setzt den Parameter T („Temperatur“) auf einen Startwert. Im Laufe jeder Iteration wird die Konfiguration leicht zufällig verändert. Ist diese neue Konfiguration besser als die alte, so ersetzt sie diese in jedem Fall. Ist sie dagegen schlechter als die Ausgangskonfiguration, dann wird diese abhängig von der Temperatur (einem Zahlenwert) und dem Ausmaß der Verschlechterung zufällig ersetzt. Dabei nimmt diese Wahrscheinlichkeit ab, je „kälter“ es wird und je größer die Verschlechterung ist (es gilt dann: $\Delta E < 0$).

Folgende Parameter haben Einfluß auf den Ablauf des Algorithmus:

- Temperaturabnahme: Es zeigt sich, daß die Wahl der Temperaturabnahme großen Einfluß auf den Ablauf hat.
- Änderungsfunktion: Das Ausmaß der Änderung an einer Konfiguration bestimmt, wie der Suchraum durchschritten wird. Sind die Änderungen zu gering, so ist das betrachtete Gebiet nicht ausreichend groß, d.h. ein entfernter liegendes Minimum bliebe unbeachtet. Sind die Änderungen dagegen zu groß gewählt, so kann das Minimum übersprungen werden.
- Abbruchbedingung: Die Suche nach einem globalen Minimum kann abgebrochen werden, wenn eine bestimmte Anzahl von Versuchen ausgeführt wurde oder sich über einen bestimmten Zeitraum die Konfiguration nicht ändert.

- Threshold Algorithmus (TA) [GD90]
  Dieses Verfahren ist eine Abwandlung des Simulated Annealing, bei der die Wahrscheinlichkeitsberechnungen entfallen. Das Kriterium zur Übernahme einer schlechteren neuen Konfiguration ist hierbei, daß der Betrag der Verschlechterung kleiner als eine Schwelle T ist, die im Laufe der Berechnungen immer kleiner wird.

  Beeinflussende Parameter sind hier:

  - Abnahme des Schwellwertes: Analog zum Simulated Annealing hat der Werteverlauf der Schwelle großen Einfluß auf die Suche, sowohl die Qualität des Optimums, als auch die Suchgeschwindigkeit betreffend. Experimente haben jedoch gezeigt, daß dieses Verfahren robuster gegenüber der Wahl

dieses Parameters ist als das SA.

- Änderungsfunktion: s. Simulated Annealing.
- Abbruchbedingung: s. Simulated Annealing.

- Great Deluge Algorithmus (GDA) [Due93]
  Dieser Algorithmus nutzt die Idee einer „Sintflut“, die eine Landschaft überflutet. Dabei stellt ein Parameter den „Wasserstand“ dar, ein weiterer Parameter bestimmt die „Regengeschwindigkeit“, also wie schnell der Wasserstand steigt.[3]

  Im Problemraum liegen die Werte der Funktion in Form einer mehrdimensionalen Landschaft. Nun wird diese Landschaft überflutet, d.h. der Wasserstandspegel erhöht sich immer mehr. Man hofft nun, daß sich die Konfiguration auf den „höchsten Berg“, also das globale Maximum rettet.

  Wieder wird eine Konfiguration zufällig verändert und ihr Funktionswert berechnet. Diesmal wird jedoch diese neue Konfiguration *unabhängig* von der Qualität der alten dann akzeptiert, wenn ihre Güte größer als der Pegelstand ist; sie sich also weiter „auf dem Trockenen“ befindet. Danach wird der Pegelstand um einen Bruchteil der Differenz zwischen neuer Güte und Pegel erhöht.

  Dieses Verfahren produziert überraschend gute Resultate, birgt jedoch die Gefahr, auf einer „Insel“ gefangen zu werden.

  Parameter sind:

  - Wasserpegel: Der Startwert für den Wasserpegel kann sehr klein gewählt werden, da dann schon frühe Werte besser sein werden und den Pegel so sehr schnell auf ein sinnvolles Niveau anheben.
  - Regengeschwindigkeit: Der Wasserpegel wird ggf. auf einen Wert zwischen dem alten Stand und der Qualität der neuen Konfiguration erhöht. Dieser Parameter legt fest, wo der neue Wert liegt.
  - Änderungsfunktion: s. Simulated Annealing.
  - Abbruchbedingung: s. Simulated Annealing.

[3] Zur besseren Anschaulichkeit wird hier also nach einem *Maximum* gesucht.

- Record–To–Record Travel (RRT) [Due93]
  Dieses Verfahren ist eine Variante des GDA. Hierbei wird aber nicht ein Wasserstand betrachtet, sondern der beste bisher gefundene Wert. Eine neue Konfiguration wird übernommen, wenn sie zumindest nur wenig schlechter als das bisherige Optimum ist; ist sie sogar besser, so bestimmt sie den neuen Bestwert.

  Diese Änderung beschleunigt die Suche, birgt aber die selben Gefahren wie GDA.

  Parameter sind:

  - erlaubte Abweichung: Um diesen Wert darf eine neue Konfiguration schlechter als die alte sein, um trotzdem angenommen zu werden.
  - Änderungsfunktion: s. Simulated Annealing.
  - Abbruchbedingung: s. Simulated Annealing.

Das Besondere dieser Verfahren ist ihre Einfachheit. Hierdurch können in einer kurzen Zeit viele Konfigurationen betrachtet werden und so hofft man, ein Optimum mit einer hohen Güte zu finden.

4. Biologische Modelle
   Unter den naturanalogen Verfahren gibt es neben jenen, die physikalische Vorgänge zum Vorbild haben, die *Evolutionären Algorithmen*. Diese Verfahren verwenden Ideen, die aus der Evolutionstheorie, z.B. von Charles Darwin, abgeleitet sind.

   Es gibt im wesentlichen zwei biologische Verfahren, die zur die Lösung eines Optimierungsproblems herangezogen werden: Evolutionsstrategien und Genetische Algorithmen. Der größte Unterschied zu den physikalisch Methoden ist, daß hier eine *Population* von Lösungskandidaten betrachtet wird. Diese Konfigurationen werden bewertet und dienen als Eltern–Generation zur Bildung von Nachkommen durch Rekombination. Diese Nachkommen werden zusätzlich mutiert, bevor eine neue Eltern–Generation aus den Individuen nach verschiedenen Kriterien ausgewählt wird.

   Formal wird ein allgemeiner evolutionärer Algorithmus als 9–Tupel

$$EA = (P(0), \lambda, \mu, l, s, r_{p_c}, r_{p_m}, \Phi, t) \tag{3.11}$$

   definiert. Die Bedeutung der Parameter kann Tabelle 3.1 entnommen werden. Die im folgenden vorgestellten Verfahren sind *Instanzen* des

allgemeinen Algorithmus nach Abb. 3.12.

$P(0) \in I^\lambda$	Anfangspopulation
I	Suchraum (Menge der Individuen)
$\lambda \in \mathbb{N}$	Populationsgröße
$\mu \in \mathbb{N}$	Nachkommenzahl
$l \in \mathbb{N}$	String–Länge
$s : I^{\lambda+\mu} \to I^\lambda$	Selektionsoperator
$r_{p_c} : I^\lambda \to I^\mu$	Crossover–Operator
$p_c \in [0,1]$	Crossover–Wahrscheinlichkeit
$m_{p_m} : I \to I$	Mutationsoperator
$p_m \in [0,1]$	Mutationswahrscheinlichkeit
$\Phi : I \to \mathbb{R}$	Fitneß–Funktion
$t : I^\lambda \to \{0,1\}$	Abbruchbedingung

Tabelle 3.1: Bedeutung der EA–Parameter

```
Wähle eine Anfangspopulation
Wähle Belegung für Parameter
Bewerte Individuen
while    keine ausreichend gute Lösung
         und nicht genug Versuche
do   Erzeuge neue Population aus alter
     Mutiere die Population
     Bewerte die Population
     Wähle neue Elterngeneration aus
end
```

Abbildung 3.12: Allgemeiner Evolutionärer Algorithmus

- Evolutionsstrategien [TB93]
  Analog zu den Vorgängen in der Natur wird hier eine Menge (*Population*) von Stichproben (*Individuen*) betrachtet. Ein Individuum wird durch einen Vektor reeller Werte dargestellt und ist ein Lösungskandidat.

  Aus einer Anfangspopulation werden durch Rekombination, d.h. Kreuzung, der Individuen Nachkommen gebildet, denen die „Eltern“ ihre Eigenschaften vererben, hier also das „Wissen“ über

den Lösungsraum. Die Nachkommen werden zuerst mutiert und dann bewertet. Die besten bilden dann die neue Population. Dabei kann die Konkurrenz der Individuen die Elterngeneration mit einbeziehen oder auch nur aus Nachkommen bestehen.

Durch die Mutation durchwandern die Konfiguration den Lösungsraum. Auf die Parameter wird eine normalverteilte Zufallsvariable mit Erwartungswert 0 addiert, wobei die Standardabweichung entweder fest vorgegeben, meist jedoch von einem Strategieparameter bestimmt wird. Dies ähnelt dem Hill–Climbing, da ein Punkt in der Umgebung der Konfiguration betrachtet wird.

Die Kreuzung belegt jeden Parameter des Nachkommen a' mit einem Wert, der zwischen den Werten a_1 und a_2 der Elternindividuen an dieser Stelle liegt (Abb. 3.13). Dies führt zu größeren Sprüngen im Suchraum, als es die Mutation vermag – die Suche wird in neue Gebiete gelenkt.

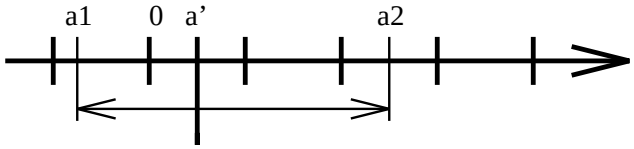


Abbildung 3.13: Ermittlung eines neuen Parameterwerts

Jedes Individuum besteht aus Strategie- und Problemparametern. Die Problemparameter definieren den Punkt im Suchraum. Es ist also das Ziel, diese Parameter zu optimieren. Die Strategieparameter andererseits beeinflussen, *wie* dieses Ziel erreicht wird. Sie gehen daher nicht in die Qualitätsbeurteilung ein.

Das Besondere dieser Strategien ist, daß die Strategieparameter ebenfalls der Evolution unterliegen. Somit entwickeln sich nicht nur die Parameter der Konfigurationen auf ein Optimum hin, sondern auch die Art und Weise, *wie* der Suchraum durchwandert wird, paßt sich dem Suchraum an.

Folgende Parameter beeinflussen den Ablauf der Suche:

- Populationsgröße λ: Anzahl der Individuen einer Population.
- Nachkommenzahl μ: Anzahl der Nachkommen, die aus einer Population erzeugt werden.

- Strategieparameter: Zum Start der Suche können ihnen zufällige Werte zugewiesen werden, da sie wie die Lösungsparameter optimiert werden. Sie beeinflussen die Mutation, die Hauptoperation der evolutionären Algorithmen. Sie geben die Schrittweite an, mit der der Suchraum durchsucht wird und die Wahrscheinlichkeit, mit der eine Suchrichtung ausgewählt wird.
- Vererbungsstrategie: Dies ist die Art, auf die die Werte der Eltern–Individuen in die Nachkommen übergehen. Möglich ist z.B., daß die Werte direkt von einem Elternteil genommen werden oder daß ein neuer Wert zufällig zwischen den beiden Werten der Eltern liegt.
- Auswahlstrategie: Sie bestimmt, welche Individuen einer Generation überleben und die neue Elterngeneration bilden. Normalerweise werden die μ besten Exemplare übernommen.

- Genetische Algorithmen [HPS92]
  Diese Verfahren sind den Evolutionsstrategien ähnlich, da sie ebenfalls auf Populationen von Individuen arbeiten; jedoch betrachten sie nur Bitvektoren. Der Lösungsraum muß also zunächst binär kodiert werden.

  Die Rekombination („Crossover“) der Individuen erfolgt durch Zusammensetzen eines neuen Exemplars aus den Bits der Eltern. Dabei wird für jede Position entweder das Bit des einen oder des anderen Elternteils übernommen.

  Die Strategieparameter werden vor dem Start des Programms festgelegt und ändern sich *nicht* während der Ausführung. Sie legen z.B. die Häufigkeit der Mutationen fest, oder wie sehr die Fitneß eines Individuums in die Wahrscheinlichkeit eingeht, für die Rekombination ausgewählt zu werden.

  Bei der Kreuzung wird jedes Bit entweder vom einen oder anderen Elternindividuum genommen. Üblich ist hierbei der „Einpunkt–Crossover“, d.h. beide Individuen werden an der selben Stelle zerschnitten und der vordere Teil vom einen, der hintere Teil vom anderen übernommen. Dies geschieht jedoch *ohne* Beachtung der Bedeutung der einzelnen Bits. Daher kann der Nachkomme sehr „weit“ von den Eltern entfernt liegen – Somit wird vor allem durch die Kreuzung der Suchraum durchschritten, was sie zum Hauptoperator der Genetischen Algorithmen macht.

Mutiert werden die Strings, indem an einer zufällig gewählten Stelle ein Bit negiert oder zufällig besetzt wird. Dies geschieht aber nicht wie bei den Evolutionsstrategien sehr häufig (z.B. mit Mutationswahrscheinlichkeit $p_m \approx 0.5$), sondern nur sehr selten ($p_m = 10^{-3}$) und dient nur dazu, auch andere als die zum Start existenten Bit–Belegungen im Laufe der Zeit zu erfassen: Ist der Wert eines Bits in allen Individuen gleich, so kann auch in den Nachkommen durch das Crossover kein anderer Wert an dieser Stelle erzeugt werden. Ein Teil des Suchraums wäre also ohne Mutation nicht mehr erreichbar.

Im übrigen verläuft der Algorithmus analog zu den Evolutionsstrategien. Besonders beachtet werden muß jedoch, daß die Art der Kodierung einen großen Einfluß auf den Ablauf und die Qualität der Lösung hat. Da nur Binärkodierungen möglich sind, muß eine Abbildungsfunktion verwendet werden. Zudem schränkt die Anzahl der Bits die Genauigkeit Δx ein:

$$\Delta x = \frac{o \Leftrightarrow u}{2^l \Leftrightarrow 1} \tag{3.12}$$

wobei $o,\ u$ Ober- und Untergrenze, l die Anzahl der Bits darstellt. Dies ist ein weiterer Unterschied zu den Evolutionsstrategien, bei denen oft der Vektor selbst den Punkt im Suchraum darstellt.

Folgende Parameter beeinflussen den Ablauf der Suche:

- Populationsgröße λ und Nachkommenzahl μ.
- Auswahlstrategie: Die Wahrscheinlichkeit, mit der ein Individuum zur Rekombination ausgewählt wird, hängt von seiner Fitneß ab, im einfachsten Fall proportional. Zudem kann hiermit die Crossover–Häufigkeit festgelegt werden.
- Mutationswahrscheinlichkeit: Hiermit kann die Häufigkeit der Mutation eingestellt werden.
- Crossover–Verfahren: Es gibt viele Möglichkeiten, wie Nachkommen aus einem Elternpaar erzeugt werden können. Neben einfachen Lösungen wie Vertauschen der Bit–Ketten an einer zufällig gewählten Stelle, gibt es die Möglichkeit, jedes Bit zufällig von einem Elternteil zu übernehmen.
- Kodierung: Sie hat einen großen Einfluß auf die Konvergenz des Verfahrens. Ist sie ungünstig gewählt, so verhindert dies das schnelle Finden des Optimums.

– Skalierungsfunktion: Da der Qualitätsunterschied der einzelnen Individuen mitunter zu gering ist, um die Auswahlwahrscheinlichkeit effektiv zu beeinflussen, müssen diese Unterschiede „verstärkt“ werden. So kann eine Stagnation der Entwicklung vermieden werden.

3.2.4 Fazit

Diese Sammlung von Problemen und Optimierungsverfahren stellt natürlich keine vollständige Aufzählung dar. Es sind vielmehr typische Beispiele betrachtet worden. Außerdem werden ständig neue Verfahren entwickelt und bekannte abgewandelt. Eine neuere Entwicklung sind z.B. die „hybriden Verfahren“, die mehrere einfache Optimierungsverfahren zu einem neuen kombinieren.

3.3 Genetisches Programmieren

Programme zur Bearbeitung von Problemen mit naturanalogen Verfahren sind bereits in größerer Anzahl verfügbar. Die meisten dieser Programme sind nicht flexibel einsetzbar, d.h. sie können nur wenige (ein) Verfahren auf einfache „Standardprobleme“ anwenden. Ziel der Projektgruppe ist die Erstellung eines Systems, das eine große Flexibilität bietet. Der Benutzer soll die Möglichkeit haben, auch kompliziertere Probleme mit dem System zu bearbeiten. Ein solches Problem ist Genetisches Programmieren. Während genetische Algorithmen die Punkte des Suchraums meist als Bit–Tupel fester Länge darstellen, verwendet man bei Genetischem Programmieren Syntaxbäume einer Programmiersprache. Entsprechend müssen die Operatoren Teilbäume bestimmen, entfernen und hinzufügen können. Auch die Fitneßfunktion ist komplizierter, sie muß u.a. einen Interpreter für die verwendete Programmiersprache enthalten. Um eine Vorstellung der Möglichkeiten, die das System bieten soll, zu vermitteln, wurde Genetisches Programmieren im Rahmen eines Seminarvortrags der Projektgruppe vorgestellt.

3.3.1 Einführung

Viele in der Praxis auftretende Probleme sind NP–vollständig, d.h. es ist kein deterministischer Algorithmus bekannt, der das Problem mit vertretbarem (polynomiellem) Aufwand löst. Zur Bearbeitung solcher Probleme kann man Genetische Algorithmen einsetzen, die zwar in der Regel keine optimale, oft aber eine „gute“ Lösung für das Problem liefern. Hier soll eine Möglichkeit vorgestellt werden, einen Genetischen Algorithmus zur Erzeugung von Computerprogrammen einzusetzen. Da eine Kodierung, wie bei Genetischen Algorithmen normalerweise verwendet, hier nicht sinnvoll erscheint, unterscheidet sich Genetisches Programmieren deutlich von anderen Genetischen Algorithmen.

3.3.2 Informelle Beschreibung

3.3.2.1 Genetischer Algorithmus

Konventionelle Genetische Algorithmen arbeiten wie folgt: Gegeben ist eine reellwertige Funktion, gesucht ein Element aus der Definitionsmenge der Funktion, für das der Funktionswert möglichst groß (möglichst klein) ist.

Die Funktion bezeichnet man als Fitneßfunktion, ihre Definitionsmenge als Suchraum oder Lösungsraum. Üblicherweise kodieren Genetische Algorithmen die Punkte des Suchraums in Bit–Tupel. Der Algorithmus verwaltet einen Vektor (Population) solcher Bit–Tupel (Individuen), die er schrittweise dem Optimum anzunähern versucht. In jedem Schritt (Generation) erzeugt er durch Verändern der Bit–Tupel einen neuen Vektor. Zur Erzeugung der neuen Individuen werden ein Rekombinations– und ein Mutationsoperator verwendet. Der Rekombinationsoperator bildet zwei neue kodierte Individuen durch Vermischen der Bit–Tupel zweier alter Individuen. Man versucht dadurch, „gute“ Eigenschaften der Eltern in den Nachkommen zu vereinen. Der Mutationsoperator ändert dann zufällig in den neuen Bit–Tupeln einige Bits, um ggf. „verlorengegangene“ Eigenschaften wieder herzustellen. Die Wahrscheinlichkeit ein Individuum zu mutieren, wählt man sehr klein, da sich der Genetische Algorithmus sonst wie eine Zufallssuche verhält.

3.3.2.2 Genetisches Programmieren

Bei Genetischem Programmieren besteht der Suchraum aus der Menge aller Zeichenfolgen, die syntaktisch korrekte Programme in einer beliebig gewählten Programmiersprache sind. Da eine Kodierung der Programme in Bit–Tupel fester Länge kaum möglich ist, arbeitet der Algorithmus für Genetisches Programmieren direkt auf den unkodierten Individuen [Koz92]. Prinzipiell kann jede Programmiersprache verwendet werden. Da Rekombinations– und Mutationsoperator auf den Syntaxbäumen der Programme arbeiten, sind Lisp–ähnliche Sprachen besonders geeignet. Der Syntaxbaum kann bei diesen Sprachen direkt aus dem Progamm abgelesen werden, Umformungen entfallen. Die hier verwendete Sprache wird in Kapitel 3.3.4.1 definiert, ein Programm in der Sprache ist z.B.:

```
(ifeqz (neg (X)) (pi) (* (X) (Y)))
```

Der Rekombinationsoperator erzeugt aus zwei Programmen zwei neue Programme, indem er in beiden Syntaxbäumen zufällig je einen Knoten auswählt und die an diesen Knoten beginnenden Unterbäume austauscht. Abbildung 3.14 zeigt die beiden Eltern, in denen die ausgewählten Knoten dick umrandet sind, sowie die beiden entstehenden Nachkommen. Der Mutationsoperator wählt aus einem Baum zufällig einen Knoten aus, und tauscht den Unterbaum gegen einen zufällig erzeugten aus. Der ausgewählte Knoten ist in Abbildung 3.15 dick umrandet.

An der Rekombination und der Mutation nehmen i.A. nicht alle Individuen der Population teil. Die Eltern für die Rekombination werden durch einen

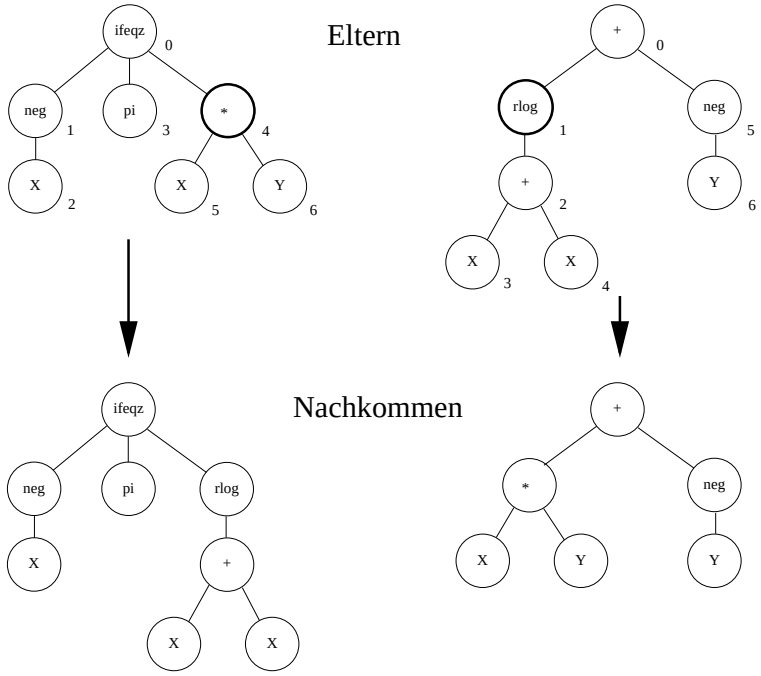


Abbildung 3.14: Rekombination

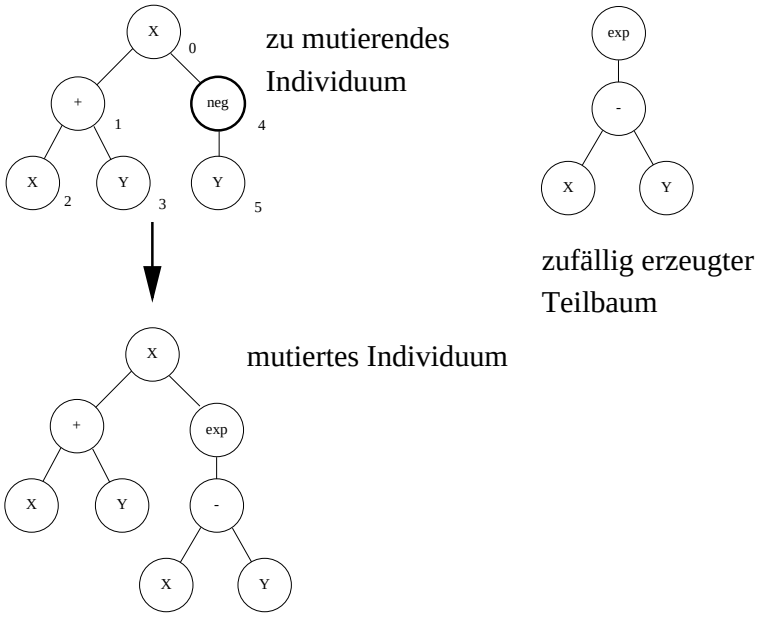


Abbildung 3.15: Mutation

Auswahloperator bestimmt, nach der Mutation werden durch einen Selektionsoperator einige Individuen aus der alten Population unverändert in die neue Population kopiert (siehe auch Abbildung 3.16). Auswahl– und Selektionsoperator wählen Individuen mit hoher Fitneß mit größerer Wahrscheinlichkeit aus.

Die Fitneßfunktion gibt hier an, wie gut ein Programm ein vorgegebenes Problem löst. Der Funktionswert wird ermittelt, indem man das Programm mit mehreren repräsentativen Eingaben testet und die Ergebnisse bewertet.

3.3.3 Schwierigkeiten bei der Implementierung

Bei der Bewertung der Programme ergeben sich folgende Probleme, die einige Einschränkungen der Beispielsprache nötig machen:

- *Ergebnistypen der Funktionen*: Rekombinations– und Mutationsoperator können Programme erzeugen, in denen Funktionen mit Argumenten des falschen Typs aufgerufen werden. Um dieses Problem zu umgehen, gibt es in der Beispielsprache nur einen Datentyp.
- *fehlerhafte Programme*: Rekombinations– und Mutationsoperator werden so definiert, daß sie nur syntaktisch korrekte Programme erzeugen können. Bei Funktionen mit Definitionslücken kann es zu Laufzeitfehlern kommen, in der Beispielsprache werden daher nur totale Funktionen zugelassen.
- *Endlosschleifen*: Falls ein Programm eine Schleife mit unerfüllbarer Abbruchbedingung enthält, terminiert die Funktion zur Berechnung der Fitneß nicht. Die Beispielsprache läßt deswegen keine Schleifen oder Rekursionen zu. Dieses Problem wird in Kapitel 3.3.7 noch ausführlicher diskutiert.

3.3.4 Beispielsprache

3.3.4.1 Syntax

Die Menge C der Funktionssymbole der Sprache setzt sich aus drei disjunkten Teilmengen zusammen. Die Elemente der Mengen sind hier nur als Beispiele angegeben, da die Mengen jeweils dem konkreten Problem angepaßt werden.

- $T_{const} = \{\texttt{null}, \texttt{pi}, \ldots\}$ Konstantensymbole

- $T_{var} = \{\texttt{X}, \texttt{Y}, \ldots\}$ Variablensymbole
- $F = \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{ifeqz}, \ldots\}$ Menge von mehrstelligen Funktionssymbolen
- $T = T_{const} \cup T_{var}$ Menge von nullstelligen Funktiossymbolen
- $C = T \cup F$

Auf der Menge der Funktionssymbole ist eine totale Funktion $z : C \to \mathbb{N}$ definiert, die die Stelligkeit der Funktionssymbole angibt. Die Sprache wird von der kontextfreien Grammatik G erzeugt.

$$G = (\{S\}, \Sigma, P, S) \text{ mit}$$

$$\Sigma = C \cup \{\texttt{(}, \texttt{)}\},$$
$$P = \{S \to u \mid u = \texttt{(}fS^i\texttt{)}, f \in C, i = z(f)\}$$

3.3.4.2 Semantik

Um die Fitneß eines Programms berechnen zu können, muß für die Beispielsprache eine Semantik definiert werden. Hier wird eine denotationelle Semantik angegeben, d.h. die Semantik eines Programms in der Beispielsprache ist eine Funktion, die eine Speicherbelegung auf einen Ergebniswert abbildet. Eine Speicherbelegung

$$mem \in MEM = (T_{var} \to \mathbb{R})$$

ist eine totale Funktion, die jeder Variable einen Wert zuordnet. Die Semantikfunktion

$$Sem : L(G) \to (MEM \to \mathbb{R})$$

bildet ein Programm auf seine Semantik ab.

Analog zur Menge der Funktionssymbole wird auch deren Semantik für jedes Problem angepaßt. Beispielhaft hier die Semantik einiger Funktionssymbole:

- $Sem(\texttt{(null)}) := \lambda mem.0$
- $Sem(\texttt{(X)}) := \lambda mem.mem(\texttt{X})$ (analog für alle Variablen aus T_{var})
- $Sem(\texttt{(+ } u\ v\texttt{)}) := \lambda mem.Sem(u)(mem) + Sem(v)(mem)$
- $Sem(\texttt{(ifeqz } u\ v\ w\texttt{)}) :=$
  $\begin{cases} \lambda mem.Sem(v)(mem), & \text{falls } Sem(u)(mem) = 0 \\ \lambda mem.Sem(w)(mem), & \text{sonst} \end{cases}$

3.3.5 Formalisierung

Der Algorithmus für Genetisches Programmieren wird als Instanz des Basisalgorithmus [Koh95, Kapitel 4] formalisiert. In Kapitel 3.3.5.1 werden die Parameter des Basisalgorithmus vorbelegt sowie einige zusätzliche Parameter eingeführt, in Kapitel 3.3.5.3 werden die Operatoren definiert. Der Algorithmus selbst wird in Kapitel 3.3.5.2 kurz vorgestellt. Sofern in [Koh95] und [Koz92] unterschiedliche Bezeichnungen für Parameter verwendet werden, orientieren sie sich hier an [Koh95].

3.3.5.1 Parameter

Spezielle Parameter für Genetisches Programmieren:

- t_{max}: Anzahl Generationen, die bis zum Abbruch erzeugt werden
- $D_{initial}$: maximale Tiefe der Individuen in der Anfangspopulation und der Teilausdrücke, die vom Mutationsoperator in existierende Individuen eingesetzt werden
- p_m: Wahrscheinlichkeit, daß ein Individuum mutiert wird
- $(mem_1, \ldots, mem_n) \in MEM^n$: Speicherbelegungen zur Berechnung der Fitneß

Parameter des Basisalgorithmus (siehe [Koh95]):

- $Ind \in IND := L(G)$: Individuum
- $\mu := 20n, n \in \mathbb{N}$: Anzahl der Individuen in einer Population
  $Pop \in POP := IND^{\mu}$
- $Pop_0 \in POP$: Startpopulation
- $\lambda := \frac{9}{10}\mu$: Anzahl der Individuen für Rekombination
- $n_m = \sum_{k=0}^{D_{initial}} (max\{z(f) \mid f \in C\})^k + 1$: Anzahl Zufallszahlen für lokale Mutation
- $n_M = (n_m + 1)\lambda$: Anzahl Zufallszahlen für Mutation

3.3.5.2 Algorithmus

Abbildung 3.16 veranschaulicht die Arbeitsweise des Algorithmus bei der Erzeugung einer neuen Generation. Die Operatoren r und m sind lokale Un-

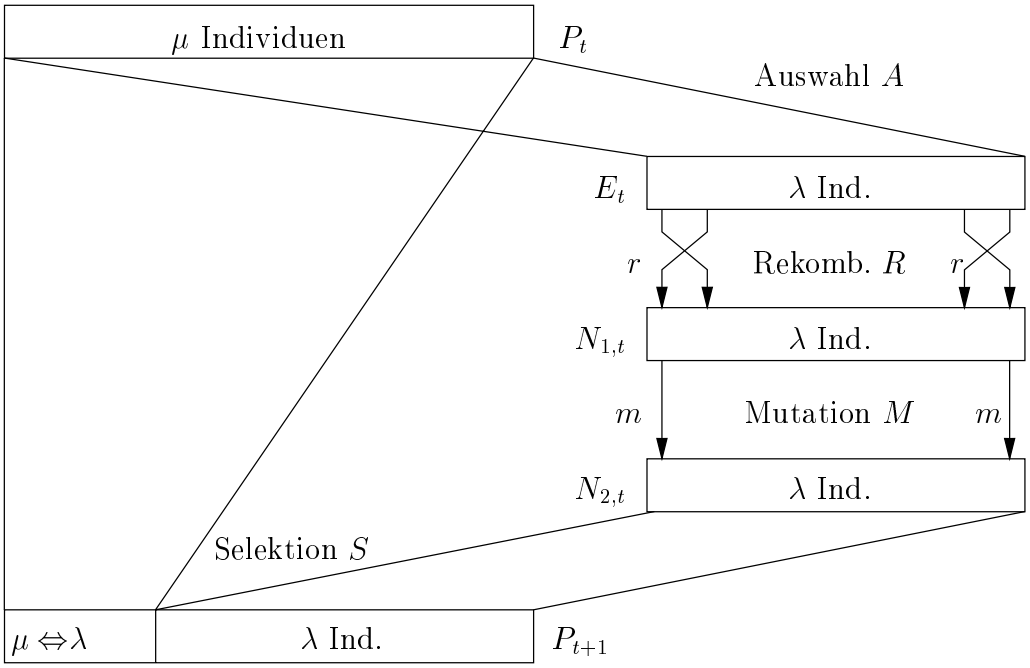


Abbildung 3.16: Erzeugung einer neuen Generation

teroperatoren von R und M, die jeweils auf zwei bzw. einem Individuum arbeiten.

$t := 0$
`REPEAT`
$E_t := A(P_t, \mathit{Uniform}([0,1), \lambda))$
$N_{1,t} := R(E_t, \mathit{Uniform}([0,1), \lambda))$
$N_{2,t} := M(N_{1,t}, \mathit{Uniform}([0,1), n_M))$
$P_{t+1} := S(P_t, N_{2,t}, \mathit{Uniform}([0,1), \mu \Leftrightarrow \lambda))$
$t := t + 1$
`UNTIL` $H(t)$

Dabei sind die Variablen:

- P_t Population zum Zeitpunkt t
- E_t Elterngeneration
- $N_{1,t}$ vom Rekombinationsoperator erzeugte Nachkommengeneration
- $N_{2,t}$ mutierte Nachkommengeneration

$Uniform([0,1),n)$ liefert n zufällig aus dem Intervall $[0,1)$ gewählte reelle Zahlen.

3.3.5.3 Operatoren

Fitneß Die Fitneß eines Individuums wird berechnet, indem man das Individuum auf jeder der Speicherbelegungen $(mem_1,\ldots,mem_n) \in MEM^n$ ablaufen läßt, und mit den Ergebnissen zunächst eine Raw–Fitneß $f_{raw} : IND \to \mathbb{R}$ bestimmt. Diese Funktion kann z.B. so aussehen:

- falls für die Speicherbelegungen Sollergebnisse $(x_1,\ldots,x_n)$ bekannt sind: $f_{raw}(u) = \sum_{i=1}^{n} \mid x_i \Leftrightarrow Sem(u)(mem_i) \mid$
- für Maximierungs– oder Minimierungsprobleme, bei denen das Ergebnis nicht bekannt ist: $f_{raw}(u) = \sum_{i=1}^{n} Sem(u)(mem_i)$.

Aus der Raw–Fitneß läßt sich die Fitneß $f : IND \to [0,1]$ des Individuums berechnen. Der Funktionswert muß zwischen null und eins liegen und für bessere Individuen größer sein. Da bessere Individuen abhängig vom Problem eine größere oder kleinere Raw–Fitneß haben, ist die Berechnung problemabhängig, z.B. für Minimierungsprobleme, oder falls Sollwerte vorgegeben sind, mit

$$f(u) = \frac{1}{1 + f_{raw}(u)},$$

bei Maximierungsproblemen mit

$$f(u) = \frac{f_{raw}(u)}{1 + f_{raw}(u)}.$$

Auswahl– und Selektionsoperator verwenden proportionale Selektion und benötigen die relative Fitneß $f_{rel} : \{1,\ldots,\mu\} \times POP \to [0,1]$, mit

$$f_{rel}(i,(u_1,\ldots,u_\mu)) = \frac{f(u_i)}{\sum_{k=1}^{\mu} f(u_k)},$$

die die Fitneß des i–ten Individuums in der Population $\vec{u} = (u_1,\ldots,u_\mu)$ ergibt, wobei die Summe über die Fitneßwerte aller Individuen aus $\vec{u}$ eins ist.

Auswahl Der Auswahloperator

$$A : POP \times [0,1)^\lambda \to IND^\lambda$$

$$(\vec{u}, (a_1, \ldots, a_\lambda)) \mapsto \vec{u}'$$

bestimmt die Individuen aus der Population $\vec{u} = (u_1, \ldots, u_\mu)$, die an der Rekombination teilnehmen. Es wird proportionale Selektion angewendet, d.h. die Wahrscheinlichkeit, daß ein bestimmtes Individuum ausgewählt wird, verhält sich propotional zu seiner relativen Fitneß. Dadurch werden bessere Individuen mit einer größeren Wahrscheinlichkeit übernommen, aber auch schlechte Individuen können in $\vec{u}'$ aufgenommen werden. Ein in $\vec{u}'$ eingefügtes Individuum wird nicht aus $\vec{u}$ gelöscht, so daß dasselbe Individuum aus $\vec{u}$ mehrmals in $\vec{u}'$ vorkommen kann.

$$\begin{array}{rcl} A((u_1, \ldots, u_\mu), (a_1, \ldots, a_\lambda)) & = & (u'_1, \ldots, u'_\lambda) \\ :\Leftrightarrow \forall i \in \{1, \ldots, \lambda\} : u'_i & = & u_j, 1 \leq j \leq \mu \\ \text{mit } \sum_{k=1}^{j-1} f_{rel}(k, \vec{u}) \leq & a_i & < \sum_{k=1}^{j} f_{rel}(k, \vec{u}) \end{array}$$

Rekombination Der lokale Rekombinationsoperator tauscht zwischen zwei Individuen zwei Teilausdrücke aus. Die Funktion

$$SubExp : L(G) \times \mathbb{N} \to L(G)$$

$$(u, n) \mapsto v$$

wird dabei verwendet, um den n–ten Teilausdruck von u zu bestimmen (Der lokale Mutationsoperator verwendet diese Funktion ebenfalls).

$$SubExp(u, n) := \begin{cases} v, \text{ falls } 0 \leq n < FkSym(u) \\ \varepsilon, \text{ sonst} \end{cases}$$

v ist das kürzeste Wort, so daß
$\exists \alpha, \beta, \gamma \in \Sigma^* : u = \alpha v \beta, v = (\gamma), \sharp_(\alpha = n, \sharp_(v = \sharp_) v$

Dabei ergibt die Funktion $FkSym : L(G) \to \mathbb{N}$ mit $FkSym(u) := \sharp_(u$ die Anzahl der Funktionssymbole in u.

Beispiel 1
Sei $u =$ `(ifeqz (neg (X)) (pi) (* (X) (Y)))`. Dann ist
$SubExp(u, 4) =$ `(* (X) (Y))` (siehe auch Abbildung 3.14).

Mit dem lokalen Rekombinationsoperator

$$r : IND \times IND \times [0, 1) \times [0, 1) \to IND \times IND$$

$$(u, v, a, b) \mapsto (u', v')$$

werden in zwei Individuen u und v mit den Zufallszahlen a und b zwei Teilausdrücke bestimmt und durch Austausch der Teilausdrücke die neuen Individuen u' und v' erzeugt.

$$r(u,v,a,b) = (u',v') :\Leftrightarrow \exists x,y,\alpha,\beta,\gamma,\delta \in \Sigma^* :$$
$$x = SubExp(u, \lfloor FkSym(u) \cdot a \rfloor), y = SubExp(v, \lfloor FkSym(v) \cdot b \rfloor),$$
$$u = \alpha x \beta, v = \gamma y \delta, u' = \alpha y \beta, v' = \gamma x \delta$$

Der Rekombinationsoperator

$$R : IND^\lambda \times [0,1)^\lambda \to IND^\lambda$$

wendet die lokale Rekombination auf je zwei nebeneinanderstehende Individuen in der Population $\vec{u}$ an.

$$R((u_1,\ldots,u_\lambda),(n_1,\ldots,n_\lambda)) = (v_1,\ldots,v_\lambda)$$
$$:\Leftrightarrow \forall k \in \{1,\ldots,\tfrac{\lambda}{2}\} : (v_{2k-1},v_k) = r(u_{2k-1},u_k,n_{2k-1},n_k)$$

($\frac{\lambda}{2} \in \mathbb{N}$, da $\lambda = 18n$ mit $n \in \mathbb{N}$)

Mutation Der Mutationsoperator tauscht in einem Individuum einen Teilausdruck gegen einen zufällig erzeugten aus. Der zu ersetzende Teilausdruck wird mit der Funktion *SubExp* bestimmt, den neuen Teilausdruck erzeugt die Funktion

$$Init : \mathbb{N} \times L'(G) \times [0,1)^+ \to L'(G)$$
$$(n,u,\vec{a}) \mapsto v$$

mit $L'(G) = \{u \in (\Sigma \cup \{S\})^* \mid S \Rightarrow^* u\}$. $\vec{a}$ ist ein Vektor aus Zufallszahlen, die zur Auswahl der Funktionssymbole verwendet werden, n die maximale Schachtelungstiefe des erzeugten Ausdrucks.

Die Funktion

$$Level : (\Sigma \cup \{S\})^* \to \mathbb{N}$$

gibt die Tiefe an, auf der sich das erste Nichtterminal aus G im teilweise abgeleiteten Wort u befindet.

$$Level(u) = n :\Leftrightarrow \exists v \in \Sigma^*, w \in (\Sigma \cup \{S\})^* : u = vSw, n = \sharp_(v \Leftrightarrow \sharp_) v$$

Sei $P_T = \{(S \to (f)) \mid f \in T\} \subseteq P$ die Menge aller Regeln in G, deren rechte Seite kein Nichtterminal enthält. Seien

$$d_T : \{1,\ldots,\mid P_T \mid\} \to P_T,\ d_C : \{1,\ldots,\mid P \mid\} \to P$$

beliebige Bijektionen.

Die Funktion *Init* simuliert Linksableitungsschritte der Grammatik G. Bei jedem Funktionsaufruf wird ein Nichtterminal S durch ein Funktionssymbol ersetzt, wobei nur Symbole für nullstellige Funktionen verwendet werden, falls sich das zu ersetzende Nichtterminal auf der maximal erlaubten Tiefe n befindet.

$$Init(n, u, (a_1, \ldots, a_k)) := \begin{cases} u, \text{ falls } \sharp_S u = 0 \\ \\ Init(n, v, (a_2, \ldots, a_k)), \text{ sonst} \end{cases}$$

$$\text{mit } \begin{cases} u \Rightarrow_L v \text{ bei Anwendung der Regel } d_C(\lfloor a_1 \cdot \mid P \mid \rfloor + 1), \text{ falls} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Level(u) < n \\ u \Rightarrow_L v \text{ bei Anwendung der Regel } d_T(\lfloor a_1 \cdot \mid P_T \mid \rfloor + 1), \text{ sonst} \end{cases}$$

Beispiel 2
Sei $T_{const} = \{\}, T_{var} = \{\texttt{X}, \texttt{Y}\}, F = \{\texttt{exp}, \texttt{-}\}$.
Sei $d_T(1) = r_C(1) = \texttt{X}, d_T(2) = r_C(2) = \texttt{Y}, d_C(3) = \texttt{exp}, d_C(4) = \texttt{-}$.
Zur Erzeugung des Ausdrucks wurden die Zufallszahlen 0.6, 0.9, 0.2 und 0.7 gezogen. Dann gilt:
$Init(2, S, (0.6, 0.9, 0.2, 0.7)) = \texttt{(exp (- (X) (Y)))}$,
wobei die Ableitung
$S \Rightarrow_L \texttt{(exp } S\texttt{)} \Rightarrow_L \texttt{(exp (- } S\ S\texttt{))} \Rightarrow_L \texttt{(exp (- (X) } S\texttt{))}$
$\Rightarrow_L \texttt{(exp (- (X) (Y)))}$
durchgeführt wird.

Der lokale Mutationsoperator

$$m : IND \times [0,1)^{n_m} \to IND$$

$$(u, \vec{a}) \mapsto v$$

bestimmt mit der Zufallszahl n_1 einen Teilausdruck im Individuum u und tauscht ihn gegen einen mit den Zufallszahlen $n_2, \ldots, n_{n_m}$ erzeugten Ausdruck aus.

$$m(u, (a_1, \ldots, a_{n_m})) = v :\Leftrightarrow \exists x, y, \alpha, \beta \in \Sigma^* :$$
$$x = SubExp(u, \lfloor FkSym(u) \cdot a_1 \rfloor), y = Init(D_{initial}, S, (a_2, \ldots, a_{n_m})),$$
$$u = \alpha x \beta, v = \alpha y \beta$$

Der Mutationsoperator

$$M : IND^\lambda \times [0,1)^{n_M} \to IND^\lambda$$

$$(\vec{u}, (a_1, \ldots, a_{n_M})) \mapsto \vec{u}'$$

entscheidet zunächst mit der Zufallszahl $a_{n_m\lambda+i}$, ob das i–te Individuum der Population $\vec{u} = (u_1 \ldots, u_\lambda)$ mutiert wird, wendet ggf. die lokale Mutation auf das Individuum an oder kopiert es unverändert in $\vec{u}'$.

$$M((u_1, \ldots, u_\lambda), (a_1, \ldots, a_{n_M})) = (u'_1, \ldots, u'_\lambda)$$

$$:\Leftrightarrow \forall i \in \{1, \ldots, \lambda\} : u'_i = \begin{cases} m(u_i, (a_{(i-1)n_m+1}, \ldots, a_{in_m})), \text{ falls } a_{n_m\lambda+i} \leq p_m \\ u_i, \text{ sonst} \end{cases}$$

Selektion Der Selektionsoperator

$$S : POP \times IND^\lambda \times [0,1)^{\mu-\lambda} \to POP$$

$$(\vec{u}, \vec{v}, (a_1, \ldots, a_{\mu-\lambda})) \mapsto \vec{w}$$

kopiert die vom Mutationsoperator ausgegebenen λ Individuen aus $\vec{v} = (v_1, \ldots, v_\lambda)$ unverändert in die nächste Generation $\vec{w}$ und wählt, um die neue Generation auf μ Individuen aufzufüllen, aus der letzten Generation $\vec{u} = (u_1, \ldots, u_\mu)$ weitere $\mu \Leftrightarrow \lambda$ Individuen mit proportionaler Selektion aus, die ebenfalls unverändert in die neue Generation übernommen werden.

$S((u_1, \ldots, u_\mu), (v_1, \ldots, v_\lambda), (a_1, \ldots, a_{\mu-\lambda})) = (u'_1, \ldots, u'_{\mu-\lambda}, v_1, \ldots, v_\lambda)$
$:\Leftrightarrow \forall i \in \{1, \ldots, \mu \Leftrightarrow \lambda\} : u'_i = u_j, 1 \leq j \leq \mu,$
mit $\sum_{k=1}^{j-1} f_{rel}(k, \vec{u}) \leq a_i < \sum_{k=1}^{j} f_{rel}(k, \vec{u})$

Abbruchbedingung Es werden t_{max} Generationen erzeugt:

$$H : \mathbb{N} \to \mathbb{B}$$

$$H(t) :\Leftrightarrow t = t_{max}$$

Die Abbruchbedingung kann ggf. noch erweitert werden. So kann z.B. der Algorithmus abbrechen, sobald die Fitneß des besten Individuums einer Generation einen bestimmten Wert überschreitet.

3.3.6 Beispiel: Symbolic Regression

Beim folgenden Beispiel handelt es sich um eine sehr einfache Anwendung für Genetisches Programmieren. Es wird kein „Programm“ im engeren Sinn,

sondern nur ein arithmetischer Ausdruck erzeugt. Einige der in Kapitel 3.3.7 genannten Probleme werden dadurch umgangen.

Gegeben ist eine Menge von Stützstellen in einem cartesischen Koordinatensystem. Gesucht ist eine Funktion, deren Funktionswerte an den Stützstellen möglichst wenig von den vorgegebenen Werten abweicht.

Zunächst wird die zu verwendende Sprache festgelegt, die hier nur eine Variable sowie einige mathematische Funktionen enthalten muß.

- $T_{const} = \{\}, T_{var} = \{\texttt{X}\}$
- $F = \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{\%}, \texttt{sin}, \texttt{cos}, \texttt{exp}, \texttt{rlog}\}$
- $z(\texttt{X}) = 0,$
  $z(\texttt{sin}) = z(\texttt{cos}) = z(\texttt{exp}) = z(\texttt{rlog}) = 1,$
  $z(\texttt{+}) = z(\texttt{-}) = z(\texttt{*}) = z(\texttt{\%}) = 2$

Die Semantik von `X` wurde bereits in Kapitel 3.3.4.2 definiert. Die Semantik der Funktionssymbole `+`, `-`, `*`, `sin`, `cos`, `exp` sind die Funktionen gleichen Namens. Da die Division und der Logarithmus auf $\mathbb{R}$ keine totalen Funktionen sind, werden sie hier durch die totalen Funktionen `%` und `rlog` ersetzt.

- $Sem(\texttt{(\% } u\ v\texttt{)}) = \begin{cases} \lambda mem.0, \text{ falls } Sem(v)(mem) = 0 \\ \lambda mem.Sem(u)(mem)/Sem(v)(mem), \text{ sonst} \end{cases}$
- $Sem(\texttt{(rlog } u\texttt{)}) = \begin{cases} \lambda mem.0, \text{ falls } Sem(u)(mem) = 0 \\ \lambda mem.ln \mid Sem(u)(mem) \mid, \text{ sonst} \end{cases}$

Anschließend wird die Fitneßfunktion definiert: Stützstellen sind die Funktionswerte des Polynoms $x^4 + x^3 + x^2 + x$ an 20 Stellen aus dem Intervall $[\Leftrightarrow 1, 1]$.

- $mem_1(\texttt{X}) = \Leftrightarrow\frac{19}{20}, mem_2(\texttt{X}) = \Leftrightarrow\frac{17}{20}, \ldots, mem_{19}(\texttt{X}) = \frac{17}{20}, mem_{20}(\texttt{X}) = \frac{19}{20}$
- $\forall i \in \{1, \ldots, 20\} : x_i = mem_i(\texttt{X})^4 + mem_i(\texttt{X})^3 + mem_i(\texttt{X})^2 + mem_i(\texttt{X})$

Die Raw–Fitness ist hier: $f_{raw}(u) = \sum_{i=1}^{20} \mid x_i \Leftrightarrow Sem(u)(mem_i) \mid$. Im Gegensatz zum formalisierten Algorithmus wird hier auch die Tiefe der vom Rekombinations– und Mutationsoperator erzeugten Individuen beschränkt ($D_{created} = 17$), sowie in jeder Generation das beste Individuum ermittelt. Weitere Parameter sind:

- $D_{initial} = 6$
- $\mu = 500$
- $t_{max} = 51$

Ergebnisse (die Klammern um die Funktionssymbole sind weggelassen):

- Bestes Individuum in Generation 0:
  ```
  (* X (+ (+ (- (% X X) (% X X)) (sin (- X X)))
   (rlog (exp (exp X)))))
  ```
  entspricht xe^x
- Bestes Individuum in Generation 2:
  ```
  (+ (* (* (+ X (* X (* X (% (% X X) (+ X X)))))
   (+ X (* X X))) X) X)
  ```
  entspricht $\frac{3}{2}x^4 + \frac{3}{2}x^3 + x$
- Bestes Individuum in Generation 34:
  ```
  (+ X (* (+ X (* (* (+ X (- (cos (- X X)) (- X X)))X)X))X))
  ```
  entspricht $x^4 + x^3 + x^2 + x$

In Generation 34 trat ein Individuum mit bestmöglicher Fitneß auf. Ihm ist die Äquivalenz zu dem zur Berechnung der Testwerte verwendten Ausdrucks nicht mehr anzusehen (siehe auch Kapitel 3.3.7).

3.3.7 Fazit

Die Festlegung der Funktionen und Variablen der Sprache und der Parameter des Algorithmus ($D_{created}$ und $D_{initial}$) setzen eine grobe Vorstellung von der Art und Größe eines Programms, das das gestellte Problem lösen kann, voraus. Falsche Wahl der Parameter führt entweder zu einer langen Laufzeit des Algorithmus oder zu einem Programm, das das Problem nur schlecht oder gar nicht löst. Dasselbe gilt für die Funktionen der Programmiersprache. Das ist kein spezieller Nachteil des Genetischen Programmierens, auch bei anderen naturanalogen Verfahren (z.B. Simulated Annealing) ist die Qualität des Ergebnisses stark von der Wahl der Parameter abhängig.

Ein größeres Problem ist die Bewertung der erzeugten Programme: Falls die Menge der möglichen Eingaben für das Programm sehr groß oder unendlich ist, kann die Fitneßfunktion den Test nur auf einer Teilmenge der möglichen Eingaben durchführen. Man kann i.A. nicht schließen, daß ein Programm, das für diese Testwerte optimale Ergebnisse erzeugt, das auch für alle anderen Eingaben leistet. Die erzeugten Programme sind auch bei kleinen Problemen sehr schwer nachvollziehbar (siehe Kapitel 3.3.6), so daß eine nachträgliche Bewertung durch den Benutzer ebenfalls kaum möglich ist.

Da die Berechnung der Fitneß sicher terminieren soll, muß der Fitneßoperator die Möglichkeit haben, zu entscheiden, ob ein Programmlauf mit einer bestimmten Eingabe terminiert. Weil das Halteproblem für turingmächtige Sprachen nicht entscheidbar ist, darf die für Genetisches Programmieren verwendete Sprache nicht turingmächtig sein.

Mit diesen Einschränkungen kann man mit Genetischem Programmieren kaum ein einzelnes Programm zur Bearbeitung eines schwierigen Problems erzeugen. Besser geeignet ist der Algorithmus für Probleme, die sich mit mehreren einfachen Programmen lösen lassen, z.B. auf den Gebieten

- Verteilte künstliche Intelligenz (emergent behaviour): Beobachtungen aus der Natur zeigen, daß viele Systeme, deren Einzelkomponenten ein sehr einfaches Verhalten aufweisen, ein wesentlich komplexeres Verhalten realisieren können (z.B. Insektenstaaten). Steuerprogramme für die Einzelkomponenten können mit Genetischem Programmieren erzeugt werden.
- Robotik (subsumption architecture): Aufgabe ist die Steuerung eines mobilen Roboters, der ein übergeordnetes Ziel verfolgt (z.B. einen bestimmten Punkt anfahren), dabei aber Randbedingungen beachten muß (z.B. Hindernisse umfahren). Der Roboter wird von mehreren Programmen gesteuert, die jeweils ein Ziel zu erreichen versuchen. Jedes dieser Programme erhält Eingaben von den Sensoren und erzeugt Steuerbefehle für die Motoren. Die Steuerbefehle werden gewichtet (z.B. hat das Ausweichen vor Hindernissen eine höhere Priorität als das Erreichen eines Punkts), so daß die Motoren immer nur einen Steuerbefehl gleichzeitig erhalten.

Teil II

Erste Konzepte und Prototyp

Kapitel 4

Zeitplan/Status

4.1 Zeitplan

4.1.1 Seminarphase

1. Es wurde ein Hauptseminar abgehalten, um das zu bearbeitende Gebiet vorzustellen, und um einen Überblick dafür zu erhalten. Die Seminarvorträge:

   Polymorphe Datentypen

   Problemen und Optimierungsverfahren

   Verschiedene Modelle für Parallele Genetische Algorithmen

   Genetisches Programmieren

2. Parallel dazu : Einarbeitung in Tools und Spezialisierung einzelner Personen. Es wurden verschiedene Verantwortungs- und Spezialisierungsbereiche geschaffen, und diese auf die einzelnen Gruppenmitglieder aufgeteilt. Dies waren:

   (a) Ein LaTeX-Experte

   (b) Ein UNIX-Experte

   (c) Ein Dokumentator

   (d) Ein Kümmerer - dieser wurde auf die ganze Gruppe übertragen

4.1.2 Planungsphase

Ziele: Festlegung der inhaltlich zu erfüllenden Aufgaben.

1. Das entgültige Programm soll eher ein Werkzeug sein, um verschiedene Lösungsverfahren aus dem Bereich der Evolutionären Algorithmen behandeln zu konnen.
2. Es soll eine möglichst große Auswahl der in den Seminarvorträgen vorgestellten Verfahren und Problemen implementiert und als Bibliothek bereitgestellt werden.
3. Eigenschaften
   Es werden sich kleinere Gruppen von ca. 2 Personen in verschiedenen Gebieten spezialisieren. Diese wären:
   (a) Populationsverwaltung
   (b) Kodierung(sverwaltung)
   (c) Parameterverwaltung
   (d) Bibliotheken (Operatoren)
   (e) Graphische Oberfläche
   (f) Labels
   (g) Log-Funktionen
   (h) Anbindung externer Probleme
   (i) Parallelität
4. Entscheidung für Hard- und Software die verwendet wird.
   Wir arbeiten mit einem UNIX-System und benutzen als Sprache ML. Die weitere Software ist auf UNIX-Systemen verfügbar, bzw. kommt aus dem Public-Domain Bereich.

4.1.3 Entwurfsphase

Begriffsbestimmungen, Anwendungen und Modelle sind weitgehend geklärt.

Tätigkeiten:

1. Lösungsverfahren festlegen.

2. Teilprobleme herauskristallisieren und diese durch Schnittstellen verbinden.
3. Grundlegende Datenstrukturen und Kommunikationswege festlegen.
4. Prototyp entwerfen und implementieren.

Ergebnisse:

1. Formale Spezifikation des Problems und seiner Teilprobleme
2. Entwurf und Implementierung des Prototyps

4.1.4 Zwischenphase

Die Erfahrung des Prototyps wurden zusammenfaßt und ausgewertet, sowie der Zwischenbericht erstellt.

4.1.5 Implementierungsphase

Noch nicht eingtreten.

4.1.6 Integrations-, Experimentier- und Schlußphase

Noch nicht eingetreten.

4.1.7 Projektbegleitende Dokumentation

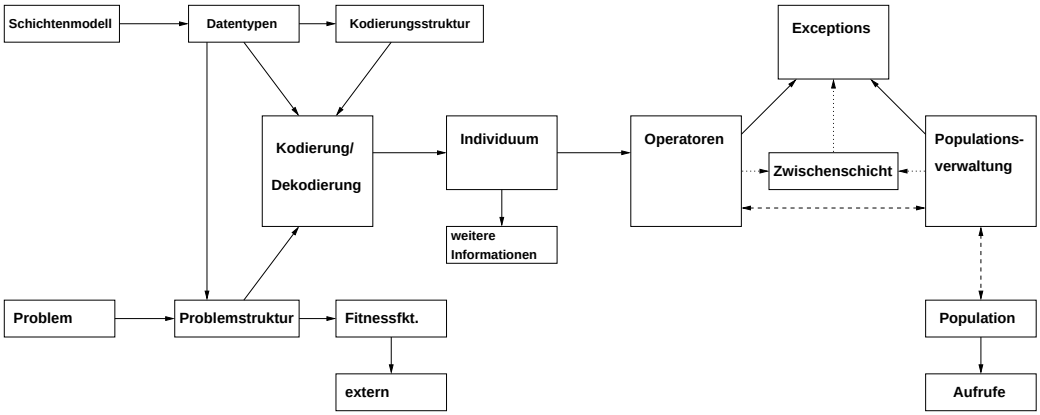
Diese ensteht permanent parallel zur Projektarbeit, und soll folgende Elemente enthalten:

1. Problemstellung
2. Literaturüberblick
3. zugrundeliegende (formale) Modelle
4. Einsatz- und Anwendungsmöglichkeiten
5. Anforderungen

6. Spezifikation
7. Design
8. Entwurfskriterien und -entscheidungen (vor allem „unlogische“)
9. Daten- und Objektstrukturen
10. Schnittstellen
11. Benutzungsoberfläche
12. Erfahrungen
13. Auswertungen
14. „Warnungen“
15. Organisation des Projekts
16. Anhang:
    - Programmlistings
    - Bedienungshinweise
    - Beschreibung von Hard-, Software und Werkzeugen
    - Anschluß an andere Systeme
    - Testumgebung
    - Testläufe
    - Experimente und deren maschinelle Auswertung

4.2 Entscheidungsgraph

Um einen besser Überblick über die Reihenfolge, und Dringlichkeit, verschiedener Entscheidungen zu erhalten, wurde folgendes Schema erstellt. Hieraus ersieht man, welche Abhängigkeiten bestehen und gesetzt wurden.


Schichtenmodell
Datentypen
Kodierungsstruktur
Exceptions
Kodierung/
Dekodierung
Individuum
Operatoren
Zwischenschicht
Populations-
verwaltung
weitere
Informationen
Problem
Problemstruktur
Fitnessfkt.
Population
extern
Aufrufe

Kapitel 5

Untergruppenberichte

5.1 Ein Schichtenmodell

Das Ziel, ein möglichst flexibles System zu schaffen macht es notwendig, den Nutzer nicht mit Aufgaben und Entscheidungen zu überfrachten. Ein Benutzer soll auf eine einfache Art mit bereits vorhandenen Verfahren experimentieren können, ohne sich um die Definition der Operatoren oder gar um die Datenstrukturen kümmern zu müssen. Andererseits soll einem Benutzer dieser Zugang nicht grundsätzlich verbaut werden. Dies soll durch das folgende Schichtenmodell beschrieben werden:

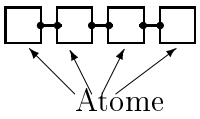
Schicht 4	Einstellung der Parameter
Schicht 3	Definition von Verfahren
Schicht 2	Definition von Operatoren
Schicht 1	Definition von Strukturen/Atomen

Auf der obersten Ebene des Schichtenmodells (Schicht 4) wird ein Verfahren aus einer Bibliothek ausgewählt und die Parameter gesetzt. Enthält die Bibliothek nicht das gewünschte Verfahren, so kann in Schicht 3 das Verfahren mit Hilfe von bereits existierenden Operatoren definiert werden. Existiert ein Operator nicht in der gewünschten Form, so kann er in Schicht 2 definiert werden. Schicht 1 ermöglicht schließlich, auch Datenstrukturen für die Problem- und Kodierungsstruktur zu definieren.

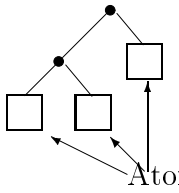
5.2 Ein einheitliches Konzept für Kodierungsstrukturen

5.2.1 Ziel

Hier zwei Beispiele für Strukturen, die als Kodierungsstrukturen auftreten können. Im folgenden wird versucht, ein einheitliches Konzept zu entwickeln, daß diese verschiedenen Strukturen beschreibt.



Kodierung mit einer Liste



Kodierung mit einer Baumstruktur

5.2.2 Konzepte

5.2.2.1 Atome

Atome sind die elementaren, unteilbaren Bausteine, aus denen Kodierungsstrukturen aufgebaut sind. Atome haben alle den selben Typ, nämlich die Vereinigung über alle Basistypen.

5.2.2.2 Basistypen

Basistypen sind reelle Zahlen, ganze Zahlen (beide evtl. mit Genauigkeiten), boolesche Werte und Permutationen. Die darauf definierten Funktionen stellt das System zur Verfügung.

5.2.2.3 Verbindung zwischen Basiselementen und Atomen

Da die Atome alle vom gleichen Typ sind, werden für jeden Basistyp je eine Funktion benötigt von Basiselement nach Atom und umgekehrt.

5.2.2.4 Übergeordnete Struktur

Die Atome sind in einer übergeordneten Struktur angeordnet, die beispielsweise eine Liste oder ein Baum ist. Die Art der Struktur ist frei bis auf die Einschränkung, daß es möglich sein muß, die Position von Atomen in der Struktur eindeutig zu definieren.

5.2.2.5 Kodierungssignatur

Die Kodierungssignatur hat folgende Elemente:

- Ein polymorpher Datentyp $T(\alpha)$.
- Einen Typ P, mit dem die Position innerhalb der Kodierungsstruktur angezeigt werden kann.

5.2.2.6 Operatoren

Operatoren sind Funktionsterme, die mit den Funktionen aus den Abschnitten 5.2.2.5 und 5.2.2.3 definierbar sind. Um den Zugriff sicherer zu gestalten, kann in der Ausbaustufe eine Klasse von Termen definiert werden, die diese Operatoren bilden, und dazu dann eine Auswertungsfunktion, die diese in Funktionsterme der Abschnitte 5.2.2.5 und 5.2.2.3 übersetzt. Gleichzeitig kann dann diese Auswertungsfunktion die Ablaufsteuerung mit Interrupts etc. übernehmen. So währen dann die Operatoren von der Ablaufsteuerung getrennt.

5.2.3 konkretes Beispiel: EAGLE

```
signature KODIERUNG =
    sig type 'a T;
        type P;
    end;

signature ATOM =
    sig datatype Basistyp = reell | integer | bit | permutation;
        val element_typ = Basistyp;
        val R : real;
        val I :  integer;
```

```
        val B : bool;
        val P : perm;
    end;

structure Listenkodierung : KODIERUNG =
    struct type 'a T = 'a list;
           type P = int;
    end;

fun list_get_atom (x::l , 0) = x
    list_get_atom (x::l , i)  = list_get_atom (l, i-1)
    list_get_atom (_, _)      = raise „Falscher Funktionsname“
fun list_set_atom (x::l , 0, y) = y::l
    list_set_atom (x::l , i, y)  = list_set_atom (l, i-1, y)
    ...
fun list_crossover (l1, l2, 0) = l2
    list_crossover (x::l, l2, i)= x::(list_crossover (l, l2, i-1))
    ...
```

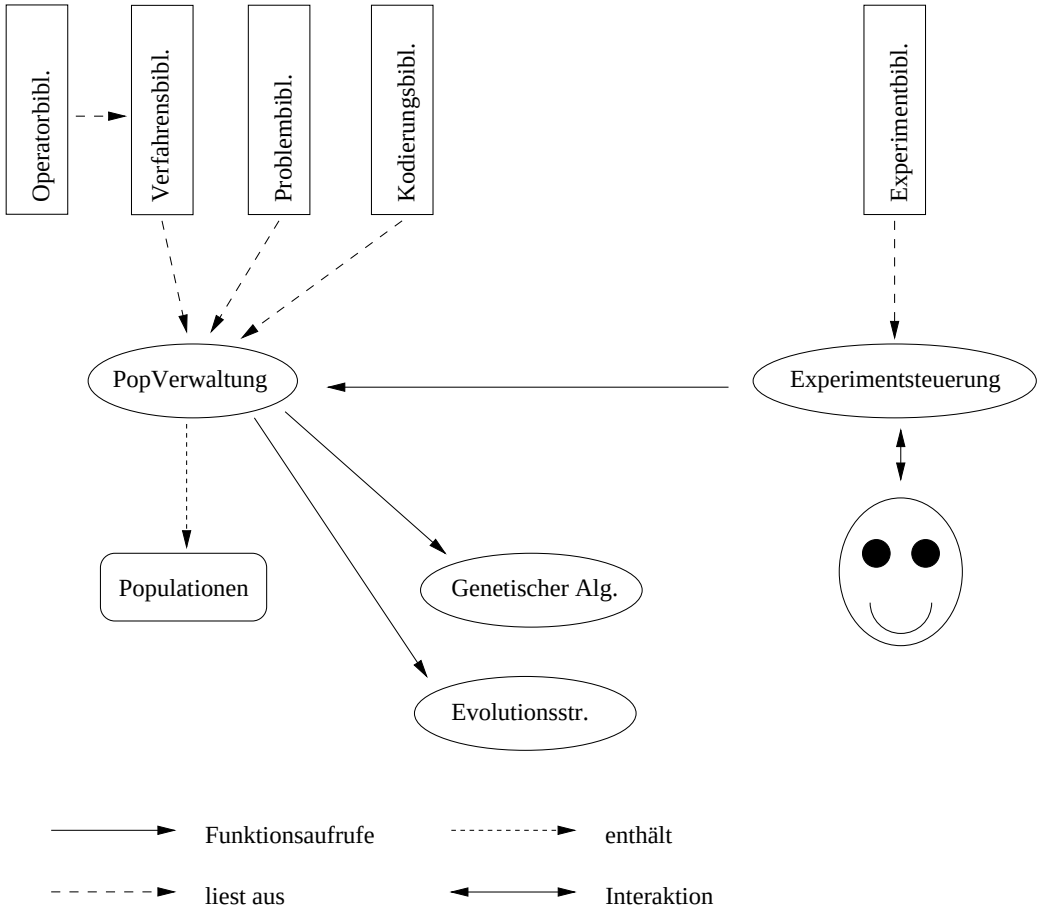
5.3 Operatorkonzept

5.3.1 Inhalt

Aufgabe der Untergruppe ist die Erarbeitung eines Konzepts für die Verfahrensoperatoren, das die gegenüber EAGLE veränderte Struktur (mehrere Populationen, freiere Kodierungsstruktur) berücksichtigt und eine kleinere Schnittstelle zwischen den Operatoren und dem restlichen System aufweist. Obwohl die Operatoren auf Funktionen, zugreifen, die von der Populationsverwaltung zur Verfügung gestellt werden, werden diese Funktionen hier nicht festgelegt. Eine exakte Definition der Schnittstelle muß in Zusammenarbeit mit der Untergruppe Populationsverwaltung definiert werden. Dasselbe gilt für Problem– und Kodierungsstruktur.

5.3.2 Aufbau

Da im System immer genau eine Populationsmenge existieren soll, ist sie in der Populationsverwaltung gekapselt. Zum Zugriff auf die Populationen stellt



die Populationsverwaltung z.B. folgende Funktionen zur Verfügung (unvollständig):

- *MainOp* : *PopBez* × *OpBez* × *ParamList* führt das unter *OpBez* in der Verfahrensbibliothek gespeicherte Verfahren auf der durch *PopBez* bestimmten Population mit den Parametern aus *ParamList* aus. Beispiel[1]:
  `MainOp(3,"Standard GA",[("lambda",30),("pm",0.001)]);`

  In einer Erweiterung könnte der Funktion eine Liste von Tripeln übergeben und die Verfahren parallel auf den Populationen ausgeführt werden. Jede Population benötigt dann einen eigenen Zufallszahlengenerator, damit die Experimente wiederholbar sind.

- *SetProblem* : *ProblemBez* legt das Problem (Problemstruktur und Fitneßfunktion) für alle Populationen fest.

[1] Der Aufbau der Parameterliste wird hier nicht festgelegt.

- $RndPop : KodBez \times int \rightarrow PopBez$ Erzeugt zufällig eine Population der angegebenen Größe mit der angegebenen Kodierung.
- $ExchangeInd : PopBez \times IndBez \times PopBez \times IndBez$ tauscht zwei Individuen zwischen zwei Populationen aus.

5.3.3 Funktionalität der Experimentsteuerung

Die Experimentsteuerung bestimmt den gesamten Ablauf eines Experiments. Sie

- wählt das Problem aus,
- legt für jede Population Größe, Kodierung und Anfangsbelegung fest,
- bestimmt, welche Verfahren mit welchen Parametern auf den Populationen arbeiten,
- tauscht Individuen zwischen Populationen aus.

Eine einfache Experimentsteuerung, die Standardverfahren auf eine Population anwendet, ist im System enthalten. Kompliziertere Experimentsteuerungen können vom Benutzer implementiert werden. Für jede Experimentsteuerung sollte eine eigene Experimentbibliothek zur Verfügung stehen, weil die dort gespeicherten Parameterbelegungen stark von Problem, Verfahren und der Experimentsteuerung selbst abhängen.

5.3.4 Bibliotheken

Operatorbibliothek: Enthält Suboperatoren, die von Verfahrensoperatoren importiert werden können.

Verfahrensbibliothek: Enthält die Verfahrensoperatoren. Ein Verfahrensoperator bildet eine Population und eine Liste von Parametern auf eine neue Population ab. Jeder Verfahrensoperator hat einen eindeutigen Namen, unter dem er von der Experimentsteuerung referenziert werden kann.

Experimentbibliothek: Enthält zu einem Experiment die Belegung der Verfahrensparameter und die Random Seeds.

Problem– und Kodierungsbibliothek: Ähnlich zur Verfahrens bibliothek lassen sich Probleme und Kodierungen über Namen referenzieren.

5.4 Populationsverwaltung

Um eine Möglichkeit zur Realisierung einer Populationsverwaltung mit mehreren Populationen aufzuzeigen, wird hier eine Erweiterung des Systems, aufbauend auf den bisherigen Vorstellungen, beschrieben. Die hier beschriebene Populationsverwaltung läßt zu, daß die einzelnen Populationen verschieden kodiert werden. Dazu wird auch eine Kodierungsverwaltung eingeführt. Die anderen Teile des Systems werden hier nur kurz oder gar nicht beschrieben.

Das Programmsystem besteht aus drei Hauptkomponenten:

- Populationsverwaltung
- Kodierungsverwaltung
- Operatoren

1. Populationsverwaltung:
   Die Populationsverwaltung speichert einzelne Populationen. Sie stellt Funktionen zum Erzeugen und Löschen von Populationen zur Verfügung, zudem innovative Konzepte zum Mischen mehrerer Populationen als auch zum Migrieren einzelner Individuen von einer Population in eine andere.

   Die einzelnen Populationen verwalten ihre Individuen und stellen Funktionen zum Einfügen und Löschen eines Individuums, zum Finden des besten bzw. schlechtesten Individuums, zum Bewerten der ganzen Population, usw. zur Verfügung. Außerdem nimmt die Population die De-/Kodierung der einzelnen Individuen vor, da sie die Kodierungsart speichert.

   Die einzelnen Populationen speichern einen Verweis auf die Kodierungsart. Hierdurch können zwei Populationen dieselbe Kodierungsart verwenden. Der eigentliche De-/Kodierungsvorgang wird aber von der Kodierungsverwaltung vorgenommen.

   Ein Individuum selbst besteht aus einem Chromosom, d.h. den kodierten Atomen, und dem zuletzt berechneten Fitneßwert. Es weiß, zu

welcher Population es gehört und kann daher seine Dekodierung veranlassen, um Operatoren Zugriff auf seine dekodierten Atome zu ermöglichen. Hierdurch kann es auch die Berechnung seiner Fitneß veranlassen.

2. Kodierungsverwaltung:
   Die Kodierungsverwaltung ist eine Bibliothek von Kodierungen und De- bzw. Kodierungsfunktionen. Sie enthält die Problemstruktur und stellt Funktionen zur Verfügung, die eine Kodierungsstruktur in die Problemstruktur überführt und umgekehrt. Sie ermöglicht den Zugriff auf zusätzliche Atome (die nicht in der Problemstruktur auftauchen). Sie ermöglicht das Erzeugen und Löschen neuer Kodierungen.

   Eine Kodierung beschreibt ggf. die zusätzlichen Atome und die Art der Kodierung der einzelnen Atome. Außerdem wird eine Sortierung der kodierten Atome festgelegt. Sie stellt Funktionen zum Zugriff auf die Atome zur Verfügung.

3. Operatoren:
   Der Hauptoperator des Systems bestimmt den Ablauf der Berechnungen auf den einzelnen Populationen. Von diesem Operator werden neue Populationen erzeugt und an die einzelnen konkreten evolutionären Verfahren übergeben. Es existiert eine Bibliothek von Standardverfahren, genannt `op1, op2, ...`

   Somit wird einem Operator eine Population als Parameter zugewiesen, dieser führt dann eine gewisse Anzahl von Schritten seines Verfahren auf dieser Population aus und liefert nach Beendigung ein Ergebnis, z.B. die Population, zurück. Der Hauptoperator kann dann mit dem Inhalt der Population weiterarbeiten, z.B. Individuen an eine andere Population senden. Da jede Population eine andere Kodierung verwenden kann, ist dieser Transfer nur in Form der Problemstruktur möglich. Die Population selbst muß dann die Überführung in die eigene Kodierung veranlassen.

   Jeder Operator kann primitivere Unteroperatoren (`SubOps`) verwenden, die ebenfalls in der Bibliothek verwaltet werden.

   Ein besonderer Operator ist der Fitneßoperator. Seine einzige Funktion ist die Berechnung der Fitneß eines Individuums, die alleine durch die Problemparameter bestimmt wird. Daher wäre die Zuordnung des Fitneßoperators zur Kodierungsverwaltung möglich, zumal sonst kein Operator auf die Problemstruktur zugreifen soll.

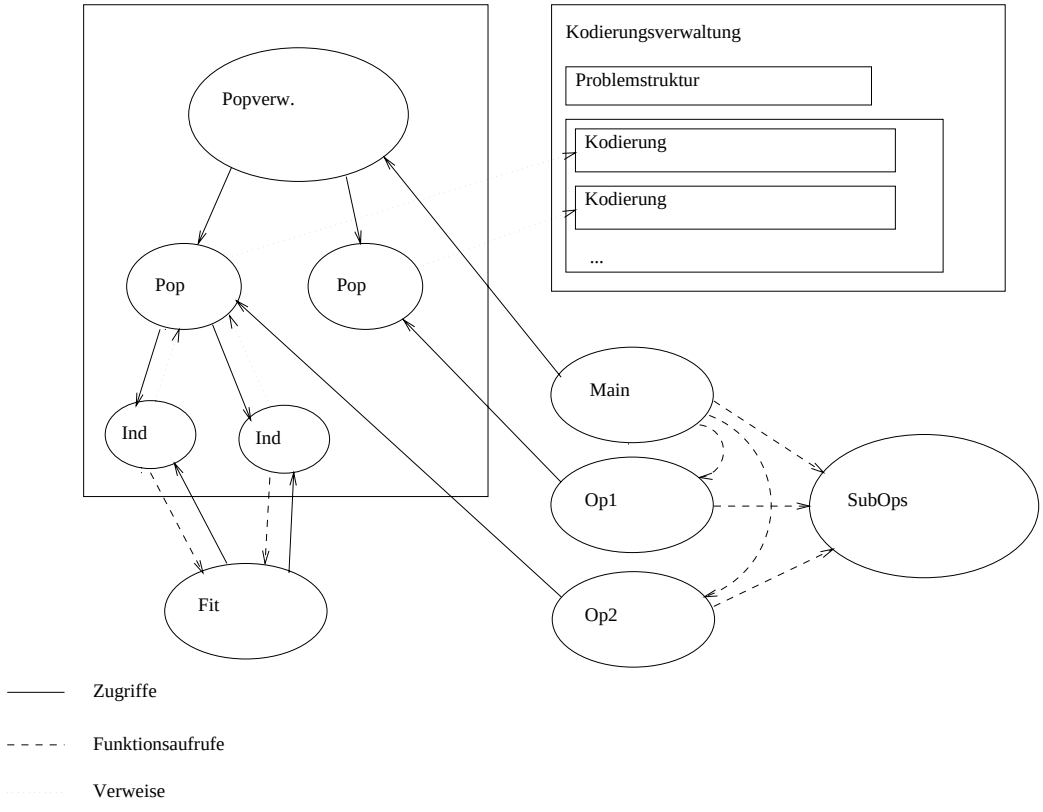


Abbildung 5.1: Aufbau der Populationsverwaltung

Kapitel 6

Prototyp

Um zu sehen wie sich die ersten Vorgaben für das endgültige System in der Praxis realisieren lassen, wurde beschlossen einen Prototypen zu erstellen. Dabei sollte ein Eindruck davon gewonnen werden, welche Ideen gut umgesetzt werden können und welche nicht. Zusätzlich bestand die Hoffnung, dabei Ideen für das endgültige System zu gewinnen. Schließlich sollten die Mitglieder der Projektgruppe dadurch konkrete Erfahrungen in ML sammeln.

Der Prototyp soll nur dazu dienen, die Ideen daran auszuprobieren. Es wird kein Teil davon direkt in das endgültige System übernommen.

In diesem Abschnitt soll ein Überblick über die wichtigsten Eigenschaften des Prototyps und seinen Aufbau gegeben werden. Zuerst werden die Vorgaben geschildert, die für den Prototyp festgelegt wurden. Danach folgt die Beschreibung der konkreten Realisierung.

6.1 Vorgaben

Da der Prototyp nur einen kleineren Aufwand darstellen sollte, wurden hier gegenüber dem endgültigen System einige Einschränkungen getroffen. Auch waren vor Entwurf des Prototyps zu einigen wichtigen Punkten, wie der Kodierungsstruktur, noch keine Entscheidungen gefallen. In diesen Punkten lehnt sich die Realisierung des Prototyps an den Entwurf von EAGLE an. Auch sollte der Prototyp keine speziellen Funktionen zur Eingabe besitzen, sondern sich aus einzelnen ML-Dateien zusammensetzen, die die Eingaben enthalten. Um einen anderen Ablauf zu erhalten, müssen daher die entsprechenden ML–Dateien geändert werden. Insgesamt wurden für den Prototyp

folgende Entscheidungen getroffen:

- **Atome** sind ein Vereinigungstyp von ganzen Zahlen, Bits und reellen Zahlen.
- Die **Kodierungsstruktur** ist eine Liste fester Länge von Atomen.
- Die **Problemstruktur** ist ebenfalls eine Liste fester Länge von Atomen.
- **Kodierung**: es existieren elementare Kodierungsfunktionen (Atom nach Liste von Atomen). Mit Hilfe dieser Funktionen wird die Kodierungsfunktion definiert. Mindestanforderung sind binäre Kodierungen für natürliche und reelle Zahlen mit einer angegebenen Genauigkeit, sowie die Identität. Die elementaren Kodierungen bzw. Dekodierungen sollen eindeutig und vollständig sein. Bei der Kodierung können zusätzliche Atome angegeben werden.
- Ein **Individuum** ist eine Instanz der Kodierungsstruktur (eine kodierte Instanz der Problemstruktur und zusätzlichen Atome) (bzw. ein Genotyp).
- Die **Population** ist eine Menge von Individuen. Es existiert nur eine Population. Es gibt daher keine Populationsverwaltung und nur eine Problem- und Kodierungsstruktur.
- Die **Operatoren** werden als ML–Funktionen geschrieben und in eigenen Dateien abgelegt, die mit `use` in das System eingebunden werden. Auf jeden Fall sollen ein Standard GA, eine Standard ES sowie ein Threshold-Algorithmus realisiert werden.
- Die **Eingabe** wird mit ML-Dateien realisiert, die in das System eingebunden werden.
  - Operatorendatei:
    Diese enthält eine ML-Funktion `main (pop → pop)`, die den Hauptoperator darstellt.
  - Problemdatei:
    Diese Datei enthält die Problemstruktur `PS (atom string)` und den Fitneßoperator `phenofit (ind → real)`.
  - Kodierungsdatei:
    Hier stehen die Kodierung und die zusätzlichen Atome.

– Initdatei:
Diese enthält eine Funktion (`initpop (pop → pop)`), die eine leere Population initialisiert und evtl. den Anfangswert des Zufallszahlengenerator setzt.

- Die **Ausgabe** soll in eine Log-Datei geschrieben werden. Dorthin können auch während der Entwicklung Debug–Ausgaben geschrieben werden. Es sollen dorthin ganze Populationen und evtl. auch Individuen, einzelne Atome, Strings, Zahlen etc. geschrieben werden können. In diese Log-Datei wird am Ende die Population geschrieben, die der Hauptoperator als Ergebnis liefert. Auf diese Population kann später vielleicht ein anderes Verfahren dann aufsetzen.

6.2 Struktur

Für die Realisierung des Prototyps wurde dieser in einzelne Module aufgeteilt. Die Beziehung der Module untereinander zeigt die folgende Abbildung. Diese Aufteilung mit festen Schnittstellen zwischen den Modulen war auch nötig, um die Programmierung auf die verschiedenen Mitglieder der Projektgruppe aufteilen zu können.

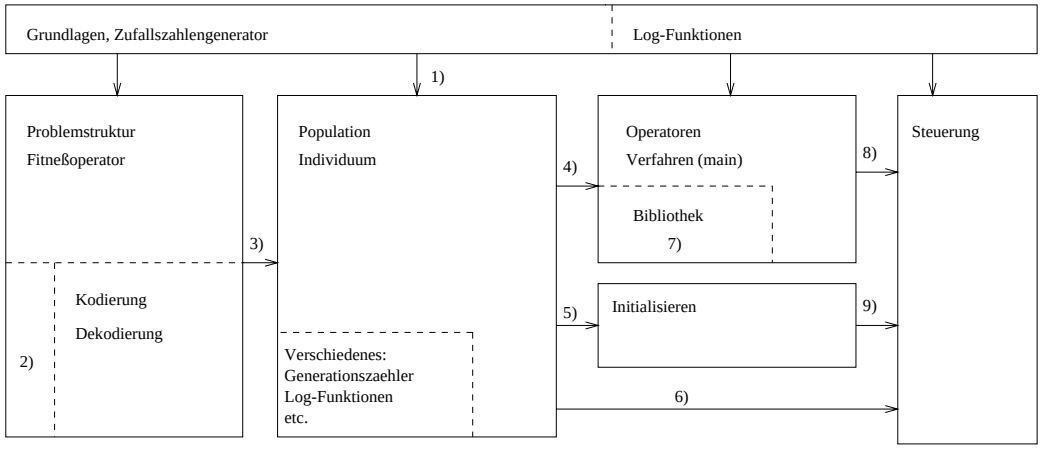


Abbildung 6.1: Struktur des Prototyps

Die einzelnen Teile wurden als jeweils eine ML–Struktur erstellt, die alle Funktionen und Datentypen des entsprechenden Moduls enthält. Die Schnittstellen zwischen den Strukturen wurden durch ML–Signaturen realisiert, denen dann die Strukturen entsprechen müssen. Wenn ein Modul von mehr

als einem anderen Modul verwendet wird, kann es vorkommen, daß zu einer Struktur auch mehr als eine Signatur existiert. Diese Struktur muß dann beiden Signaturen entsprechen. Im folgenden werden die Schnittstellen bzw. die dazugehörenden Module kurz beschrieben.

1. Das Modul „Grundlagen“ stellt Funktionen und Datentypen bereit, die von allen anderen Modulen verwendet werden können. Die Schnittstellen zu den anderen Modulen sind immer gleich.

   Der Datentyp `atom` für die Atome der Kodierungs- und Problemstruktur ist als Vereinigungstyp von ganzen Zahlen, Bits und reellen Zahlen definiert. Für diese Typen gibt es hier jeweils Umwandlungsfunktionen, die diesen aus einem Atom auslesen oder aus diesem ein Atom erstellen (z.B. `atom2int` und `int2atom`).

   Der Zufallszahlengenerator befindet sich ebenfalls in diesem Modul. Es gibt dazu eine Funktion `randomseed`, mit der der Zufallszahlengenerator initialisiert wird und die Funktion `random` mit der die nächste Zahl der Zufallszahlenfolge ausgelesen wird.

   Auch die Konstante für den Stream, der mit der Log–Datei verbunden ist, (`log`) und die Funktionen mit denen einfachere Typen in die Log–Datei geschrieben werden können (z.B. `writeatom` und `writestring`) sind hier für alle anderen Module sichtbar.

2. Zur Definition der Kodierungs- und Dekodierungsfunktion werden zwei Funktionen bereitgestellt, die einzelne Atome kodieren und dekodieren. Es sollten hier die Kodierungsart `nocoding` für keine Kodierung und die Kodierungsarten `codintbit` bzw. `codrealbit` für eine Kodierung von ganzen bzw. reellen Zahlen in Bitstrings unterstützt werden. Dies geschieht mit den Funktionen `codeatom` und `decodeatom`, die ein Atom mit einer angegebenen Kodierung kodieren bzw. dekodieren. Auf diese Funktionen bauen die Funktionen des Kodierungsmoduls auf.

3. Das Modul „Kodierung/Fitneß“ enthält die Kodierungsfunktionen und die Fitneßfunktion `fitness`. Es definiert einen Datentyp `genotype`, der der Kodierungsstruktur entspricht und aus einer Liste fester Länge von Atomen besteht. Die Atome der Problemstruktur und die Strategieatome sind in dieser Struktur kodiert gespeichert. Mit den Funktionen `getrepatom`, `setrepatom`, `getstratatom` und `setstratatom` wird auf diese Atome zugegriffen. Dabei werden sie entsprechend kodiert bzw. dekodiert.

4. Die Funktionen, auf denen die Operatoren aufbauen, werden vom Mo-

dul „Population/Individuen“ bereitgestellt. So werden hier die Datentypen für Populationen (`pop`) und Individuen (`ind`) definiert. Eine Population besteht dabei aus einer Liste von Individuen. Ein Individuum ist eine Instanz der Kodierungsstruktur, also ebenfalls eine Liste von Atomen mit fester Länge. Dazu gibt es hier Funktionen zum Zugriff auf einzelne Individuen einer Population (z.B. `getind` und `setind`) und Funktionen, die Informationen über die Population bzw. Individuen liefern (z.B. `getbestind`, `getavgfitness`, etc.). Alle Funktionen, die von den Operatoren verwendet werden um die Population und die Individuen zu verändern, sind hier definiert, auch wenn manche nur direkte Aufrufe von Funktionen des Kodierungsmoduls sind.

5. Zusätzlich zu den Funktionen aus 4. können vom Initialisierungsmodul aus noch Funktionen verwendet werden, mit denen Atome der Problemstruktur gelesen und geschrieben werden (`getrepatom`, `setrepatom`). Zusätzlich können von hier aus auch die Strategieatome gesetzt werden.

6. Das Hauptprogramm kann auf alle Funktionen des Moduls „Population/Individuen“ zugreifen, die auch von den beiden anderen Modulen „Operatoren“ und „Initialisierung“ verwendet werden. (siehe 4. und 5.).

7. In der Operatorenbibliothek befinden sich die Hauptoperatoren für einen Threshold Algorithmus, einen genetischen Algorithmus und eine Evolutionsstrategie. Außerdem gibt es hier die Suboperatoren, die bei der Realisierung von eigenen Hauptoperatoren verwendet werden können.

8. Vom Hauptprogramm aus ist vom Modul „Operatoren“ nur die Funktion `main` sichtbar. Diese wendet den Hauptoperator auf eine Population an und gibt die Population, die dabei entsteht, als Ergebnis zurück.

9. Das Initialisierungsmodul stellt die Funktion `initpop` für das Hauptprogramm zur Verfügung. Diese Funktion initialisiert die Population, bevor der Hauptoperator darauf angewendet wird. Im einfachsten Fall wird einfach eine bestimmte Anzahl zufällig erzeugter Individuen in die Population eingefügt.

6.3 Realisierung

6.3.1 Grundlagen

Das Modul „Grundlagen“ stellt Funktionen und Datentypen bereit, die von allen anderen Modulen verwendet werden. Es besteht hauptsächlich aus der Definition eines Datentyps für die Atome und einem Zufallszahlengenerators.

6.3.1.1 Atome

Der Datentyp `atomtype` definiert die Konstanten `intatom`, `bitatom` und `realatom`, die den Typ eines Atoms angeben. Jedes Atom enthält eine Komponente diesen Typs, die angibt, um was für ein Atom es sich handelt. Der Datentyp ist folgendermaßen realisiert:

```
datatype atomtype = intatom | bitatom | realatom
```

Atome stellen einen Vereinigungstyp der ML-Typen `int`, `bool` und `real` dar. Dieser ist im Prototyp als kartesisches Produkt dieser Typen definiert. Zusätzlich enthält er eine Komponente vom Typ `atomtype`, in der der Typ des Atoms gespeichert wird. Sie gibt an, welches Element des kartesischen Produkts für den Wert des Atoms relevant ist.

```
type atom = (atomtype * int * bool * real)
```

Weiterhin gibt es für jeden der Typen `int`, `bit` und `real` eine Funktion, mit der ein Atom in einen Wert dieses Typs umgerechnet werden kann. Dies ist natürlich nur möglich, wenn es sich um ein Atom mit dem entsprechenden Atomtyp handelt (s. o.). Andernfalls wird eine Exception `WrongType` ausgelöst. Ebenso gibt es eine Funktion, mit der ein Atom aus einem Wert dieses Typs erstellt werden kann. Bei diesen Umwandlungen wird bei den Atomen jeweils nur das Element des kartesischen Produkts gesetzt bzw. ausgelesen, das dem Atomtyp entspricht. Die folgenden ML–Zeilen zeigen diese Funktionen für den Typ `int`:

```
fun atom2int ((intatom, x, _, _)) = x
  | atom2int (_) = raise WrongType
fun int2atom (x) = (intatom, x, false, 0.0)
```

6.3.1.2 Zufallszahlengenerator

Der Zufallszahlengenerator berechnet eine Folge mit der Berechnungsvorschrift $R_{n+1} = (a * R_n + c)\ mod\ m$. Dabei bestimmen die Konstanten a, c und m die Eigenschaften der Zufallszahlenfolge. Im Prototyp wurden $a = 125$, $c = 0$ und $m = 2796203$ verwendet. Die zuletzt berechnete Zahl der Zufallsfolge wird im Prototyp in einer globalen Variablen gespeichert. Diese wird mit 100001 initialisiert. Falls nicht ein anderer Anfangswert gesetzt wird, beginnt also die Zufallszahlenfolge mit diesem Wert. In ML ist dies folgendermaßen realisiert:

```
val x = ref 100001
```

Um den Zufallszahlengenerator zu initialisieren, kann der Anfangswert der Folge mit der Funktion `randseed` auf einen bestimmten Wert gesetzt werden. Dabei wird der Wert der globalen Variable auf diesen Wert gesetzt, falls es sich um eine reelle Zahl zwischen 0 und 1 handelt.

```
fun randseed (seed) =
    if seed < 0.0 orelse seed > 1.0
       then raise OutOfScope
       else (x := (floor)(seed * real(m)))
```

Das folgende Beispiel zeigt die Funktion `rand`, die erst den nächsten Wert der Zufallszahlenfolge nach der oben beschriebenen Formel berechnet und danach diesen Wert, in den Bereich zwischen 0 und 1 skaliert, als reelle Zahl zurückgibt.

```
fun rand () =
    ( (x := (a * !x + c) mod m);
      (real(!x) / real(m)) )
```

Mit dieser Funktion wird auf die Zahlen der Folge zugegriffen. Zusätzlich gibt es noch eine Funktion `randombound`, die diese Zahl in einen beliebigen Bereich skaliert.

6.3.2 Log-Datei

Das Modul „Log–Datei“ enthält die zum Zugriff auf die Log–Datei nötigen Funktionen und Konstanten. In diese wird die Ausgabe des Prototyps geschrieben. Dazu stellt das Modul eine Konstante `log` vom Typ `outstream` bereit, die beim Starten des Programms mit der Log–Datei verbunden wird.

```
val log = open_out(Files.LogFileName)
```

Da es sich dabei um eine Variable vom Typ `outstream` handelt, können alle ML-Ausgabefunktionen zur Ausgabe in die Log–Datei verwendet werden. Zusätzlich stehen in diesem Modul Funktionen (wie `writeatom`), mit denen die grundlegenden Datentypen (z. B. Atome) in einen Stream geschrieben werden können. Sie sind daher nicht nur auf die Log–Datei beschränkt, sondern können auch für andere Dateien verwendet werden. Hier wird als Beispiel die Funktion zum Schreiben eines Atoms gezeigt.

```
fun writeatom (out, atom) =
  ( if (Basics.getatomtype(atom) = Basics.intatom) then
      writestring(out, makestring(Basics.atom2int(atom)))
    else
    ( if (Basics.getatomtype(atom) = Basics.bitatom) then
      ( if (Basics.atom2bit(atom) = true) then
          writestring(out, "1")
        else
          writestring(out, "0"))
      else
      ( if (Basics.getatomtype(atom) = Basics.realatom) then
          writestring(out, makestring(Basics.atom2real(atom)))
        else
          raise Basics.WrongType))
  )
```

Die Funktionen in diesem Modul dienen als Grundlage für die Funktionen `writeindcoded`, `writeinddecoded`, `writepopcoded` und `writepopdecoded`, die sich im Modul Population/Individuen befinden und dazu dienen, ganze Individuen und Populationen in die Log–Datei zu schreiben.

6.3.3 Kodierungsfunktionen und Fitneß

Die Kodierung/Dekodierung von Individuen wurde in einem Programmteil mit der Fitneßfunktion auf Phenotypen zusammengefaßt. Dabei wurde auf die logische Trennung unabhängiger Teile geachtet:

- Festlegung des Problems durch Definition des Phenotyps und der Fitneßfunktion auf Phenotypen.
- Festlegung des Genotyps durch Angabe der Kodierungsart für jeden Problemparameter, der Art der zusätzlichen (Strategie-) Parameter und deren Kodierung, sowie die Angabe einer Sortierung für das ganze Genom.

- Funktionen, die diese Informationen benutzen um konkrete Individuen zu de-/kodieren und die Fitneß eines kodierten Individuums berechnen.

Durch diese Unterteilung muß z.B. bei Verwendung einer anderen Kodierungsstruktur nur an einer Stelle eine Änderung vorgenommen werden, alle anderen Teile bleiben unverändert. Insbesondere müssen die Kodierungsfunktionen nicht geändert werden. (Außerdem gibt es keine Beschränkung der Komplexität der einzelnen Teile.)

Im einzelnen besteht dieser Programmteil somit aus folgenden Elementen:

- Problembeschreibung (`Problem.sml`)
- Kodierungsbeschreibung (`Codstruct.sml`)
- Kodierungsfunktionen (`Coding.sml`)

6.3.3.1 Problembeschreibung

In der Problembeschreibung wird das Problem definiert, d.h. neben der Problemstruktur ist auch die Fitneßfunktion hier zu finden.

Konkret enthält die Datei eine Liste `reptypelist` von Atomtypen (Beispiel: `[realatom, intatom, bitatom]`) und eine Funktion `phenofit`, die bei Eingabe einer Liste dieses Typs einen Real-Wert liefert.

6.3.3.2 Kodierungsbeschreibung

Die Kodierungsbeschreibung enthält alle Informationen zur Kodierung der Problemstruktur in die Kodierungsstruktur (und deren Dekodierung). Konkret sind dies:

- `codtab` Eine Liste, die jedem Element der Problemstruktur eine Kodierungsart und eine Kodierungsgenauigkeit zuordnet.

  Beispiel: `[(codintbit,4), (codrealbit,3), (nocoding,1), (codintbit,3)]`

- `strattypelist` Diese Liste von Atomtypen legt die Art, Anzahl und Reihenfolge der zusätzlichen Parameter fest, die ein Verfahren ggf. benötigt. (Diese Parameter werden oft „Strategie-Parameter“ genannt.)

Beispiel: `[realatom, intatom, bitatom]`

- `strattab` Mit dieser Liste wird jedem Strategie-Atom eine Kodierungsart und eine Kodierungsgenauigkeit zugewiesen. (Selbes Format wie `codtab`.)
- `sorttab` In dieser Liste wird verzeichnet, an welche Position ein Parameter des kodierten Individuums rücken soll. Dabei muß beachtet werden, daß die Strategie-Atome vor der Sortierung an die Problematomliste angehängt werden. Somit muß diese Tabelle eine *Permutation* über die ersten `length` Zahlen sein, wobei `length` die Gesamtlänge eines kodierten Individuums ist.

  Beispiel: `[2,4,7,1,9,3,5,8,6]`

6.3.3.3 Kodierungsfunktionen

Kodierungsfunktionen kodieren die Individuen vom Phenotyp in den Genotyp bzw. dekodieren sie vom Genotyp in den Phenotyp.

Erwähnenswert ist, daß beim Auswerten der Struktur `Coding` schon einige im weiteren Verlauf benötigte Konstanten und Listen aus der „Eingabe“ erzeugt werden, die die beiden anderen Dateien darstellen. So wird z.B. die Tabelle zum Umkehren der Sortierung des Genotyps (`desorttab`) aus der Tabelle `sorttab` erzeugt, die in der Kodierungsbeschreibung festgelegt wurde. Es muß beachtet werden, daß dies nur zur Vereinfachung so gelöst ist – man hätte auch das Bereitstellen verlangen können.

Funktionen des Kodierungsteils Es sollen nun beispielhaft die Funktionen des Kodierungsteils betrachtet werden, die von anderen Programmteilen benutzt werden.

- Bevor die Verfahren mit Individuen arbeiten können, müssen sie diese erst erhalten. Hierzu existiert ein „leeres“ Individuum namens `getcodind`, das als Muster benutzt werden kann. Es wird beim Erzeugen der Struktur `Coding` berechnet und ist konstant im Ablauf des Programms.

```
val getcodind =
     sort(makecodelist(reptypelist@strattypelist,
                       codtab@strattab))
```

- Um ein Atom der Problemstruktur zu setzen, muß der Genotyp erst desortiert werden, d.h. die ursprüngliche Reihenfolge der Atome wird wiederhergestellt. Diese Liste wird an die Funktion `changeatomgt` übergeben, zusammen mit der Liste der Kodierungen, dem neu einzusetzenden Atom und der gewünschten Stelle. Die veränderte Liste wird wieder sortiert und zurückgeliefert.

```
fun setrepatom (gt,at,n) =
  sort(changeatomgt (desort(gt),codtab,at,n))
```

- Die Änderung von Strategieatomen erfolgt im wesentlichen analog zur Bearbeitung von Problematomen, jedoch muß zuerst die Liste der Problematome, die im unsortierten Indiviuum am Anfang stehen, extrahiert werden (mit der Funktion `headn`). Daran wird dann die veränderte Liste der Strategieatome angehängt, wobei der Anfang dieser Liste durch die Funktion `stratstart` ermittelt wird.

```
fun setstratatom (gt,at,n) =
    let val gtdes = desort gt
    in  sort (headn(gtdes,lengthphenocode)
         @changeatomgt(stratstart gtdes,strattab,at,n))
    end
```

- Um die Fitneß eines Genotyps zu ermitteln, muß dieser erst dekodiert und dann an die Fitneß-Funktion für Phenotypen übergeben werden. Die Funktion `phenofit` wird in der Problembeschreibung definiert.

```
fun fitness (gt) = phenofit (decode(gt))
```

- Um den Wert von Problem- oder Strategie-Parametern zu ermitteln, muß der Genotyp zuerst desortiert werden. Danach wird anhand der Kodierungstabelle die Position des gesuchten Parameters ermittelt und die dort beginnende Atomliste (bzw. das dort befindliche Atom) dekodiert.

```
fun getrepatom (gt, n) = getra (desort(gt), n, codtab)
```

- Analog zur Funktion `getrepatom` ermittelt diese Funktion den Wert eines Strategie-Parameters. Allerdings wird hier nur die Liste der Strategie-Atome betrachtet.

```
fun getstratatom (gt, n) =
        getra(stratstart(desort(gt)), n, strattab)
```

Hilfsfunktionen Die folgenden Funktionen sind Beispiele für Hilfsfunktionen, die außerhalb des Programmteils (genauer: der Struktur `Coding`) nicht sichtbar sind.

- Umwandlung einer ganzen Zahl in eine Liste von Bit-Atomen. Das „Least Significant Bit“ steht an erster Stelle. Man beachte, daß bei Verwendung eines Bits jeder Wert ungleich Null in `true` kodiert wird. Dies ist besonders bei Überschreitung des kodierbaren Bereichs bedeutsam.

```
fun int2atomlist (0, 1) = [Basics.bit2atom (false)]
  | int2atomlist (_, 1) = [Basics.bit2atom (true)]
  | int2atomlist (i, p) =
    if p <= 0  then raise coderror
       else Basics.bit2atom (1 = i mod 2)
         :: int2atomlist (i div 2, p - 1)
```

- Mit `codeatom` wird jeweils ein Atom kodiert, wobei die Einträge der Kodierungstabelle benutzt werden. Relle Werte werden zuerst in ganze Zahlen umgerechnet und dann in Bits gewandelt.

```
fun codeatom ((nocoding, 1), a) = [a]
  | codeatom ((nocoding, _), _) = raise coderror
  | codeatom ((codintbit, p), a) =
    if p <= 0  then raise coderror
       else int2atomlist (Basics.atom2int (a), p)
  | codeatom ((codrealbit, p), a) =
    if p <= 0  then raise coderror
       else int2atomlist (floor(Basics.atom2real(a) *
                          real(powerof2(p))), p)
```

- `insertat` setzt ein Atom an die angegebene Position einer Liste, wobei die Liste entsprechend erweitert wird, wenn sie nicht lang genug ist. Diese Atome werden (hoffentlich) später durch die eigentlich gewünschten ersetzt.

```
fun insertat (atom, 1, nil) = [atom]
  | insertat (atom, n, nil) =
       if n < 1 then raise coderror
                else atom::insertat(atom,n-1,nil)
  | insertat (atom, 1, hd::tl) = atom::tl
  | insertat (atom, n, hd::tl) =
                       hd::insertat(atom,n-1,tl)
```

6.3.4 Populationsverwaltung

6.3.4.1 Hauptideen

Die Populationsverwaltung ist im Prototyp eher als Individuenverwaltung anzusehen, da sie nur eine Liste von Individuen verwaltet, und nicht, wie für den Haupttyp vorgesehen, mehrere Populationen. Die Individuen sind in einer Liste gespeichert:

```
Population = Individuum list
```

Ein Individuum ist eine Liste von Atomen :

```
Individuum = atom list
```

6.3.4.2 Ändern von einzelnen Atomen

Die Änderung eines Atoms wird mit Standardfunktionen von SML verwirklicht:

- `nthtail` liefert den Rest einer Liste ab einer bestimmten Position,
- `rev` dreht die Reihenfolge einer Liste um,
- `@` verknüpft 2 Listen zu einer.

```
setatom(L,p,a) = rev(nthtail(rev(L),p))@a@nthtail(L,p-1)
```

6.3.4.3 Bestes/schlechtestes Individuum

Die Funktionen, die ein bestes bzw. schlechtestes Individuum liefern, sind sehr ähnlich. Deshalb betrachten wir hier nur die Funktion `getbestind`, die ein bestes Individuum einer Population zurückgibt.
Die Funktion nimmt die Population als Liste von Individuen, wie definiert. Dann wird das letzte Individuum genommen, und als zeitweilig Bestes angesehen. Die Liste wird nach vorne durchlaufen, und die jeweilige Fitneß des gerade betrachteten Individuums wird mit dem aktuell besten verglichen. Ist das betrachtete besser, merkt man sich dieses, und läuft weiter. Ist die Liste zu Ende, wird das gespeicherte Individuum zurückgegeben.

Wird eine leere Liste als Parameter übergeben, so gibt die Funktion auch eine leere Liste zurück.

```
fun getbestind(l) =
    let
    fun calc(nil) = []
    | calc(i::nil)=i
    | calc(i::r) =
      let
      val ib = calc(r)
      in
      if Coding.fitness(i) >= Coding.fitness(ib)
      then i else ib end
    in
    calc(l)
    end
```

Diese Funktion erhält eine Population und gibt ein bestes Individuum zurück.

6.3.4.4 Erzeugung eines neuen Individuums

Ein neues Individuum zu erzeugen heißt hier nicht, nur eine Liste von „leeren“ Atomen bereitzustellen, sondern diese auch mit zufälligen Werten zu füllen. Es muß dabei einzeln beachtet werden, um was für einen Atomtyp es sich handelt. Die Liste wird als leere Atomtyp-Liste von der Kodierung geholt und die einzelnen Atome entsprechend Ihrer Art mit zufälligen Werten belegt.

```
fun newind(p) =
let val i = Coding.getcodind
 fun help(i,0) = i
 | help(l,n) =
   if Basics.getatomtype(getatom(l,n)) = realatom
      then setatom(help(l,n-1),n,Basics.real2atom(random()))
   else if Basics.getatomtype(getatom(l,n)) = intatom
      then setatom(help(l,n-1),n,Basics.int2atom(floor(
         randombound(0.0,100.0))))
```

```
      else setatom(help(l,n-1),n,if random() > 0.5
         then Basics.bit2atom(true)
         else Basics.bit2atom(false))

in
 (p@help(i,Coding.length)::nil,length(p)+1)
end
```

6.3.4.5 Löschen eines Individuums

Ein Individuum wird gelöscht, indem es aus der Liste gelöscht wird. Das zu löschende Individuum wird dabei als Index der Liste übergeben:

```
  deleteind(L,i) = rev(nthtail(rev(l),length(l)-(n+1)))
                                      @nthtail(l,n+1)
```

6.3.5 Operatoren

Der hier beschriebene Teil des Prototyps umfaßt die Verfahren und Operatoren, d.h. einen Systemteil, der später auch vom Benutzer geändert oder erweitert werden soll. Der Schnittstelle dieses Teils zum übrigen System kommt daher besondere Bedeutung zu. Für den Prototyp wurde ein Genetischer Algorithmus, eine Evolutionsstrategie sowie ein Threshold–Algorithmus implementiert.

6.3.5.1 Aufbau

Die Verfahren und die Operatoren sind in einzelnen Dateien abgelegt. Das Verfahren lädt die Operatoren aus den Dateien.

```
(* load sub-operators *)

use "Operators/OnePointCrossover.sml";
use "Operators/Mutation.sml";
use "Operators/PropSelect.sml";
use "Operators/SimpleTermCriterion.sml";
```

Alle gleichartigen Operatoren (Crossover, Mutation, Abbruchkriterium, ...) haben die gleichen Namen, nur die Namen der Dateien, in denen sie abgelegt

sind, unterscheiden sich. Andere Operatoren lassen sich so durch einfaches Ändern der Namen der einzubindenden Dateien laden.

Die Dateinamen stehen hier (wie auch alle anderen Parameter) als Konstanten direkt im Programmtext.

Der Prototyp führt das Verfahren durch Aufruf von `main()` mit einer zufällig gewählten Startpopulation aus. Analog zu den Operatoren lassen sich andere Verfahren durch Änderen des Dateinamens im Prototyp laden. Als Beispiel für ein Verfahren ist hier der Genetische Algorithmus angegeben, der die Operatoren `termCriterion()` (bricht hier das Verfahren nach 50 Generationen ab), `crossover()` (führt einen Crossover auf der gesamten Population aus), `mutate()` und `select()` (gibt eine durch proportionale Selektion entstandene neue Population zurück) lädt (s.o.).

```
        fun main(mpop)=
            let
             val pc=0.6
             val pm=0.001
             val mu=getsize(mpop)
             val lambda=mu
             fun main_r(mpop)=
              if termCriterion(mpop)
                  then mpop
                  else
                   (
                   writepopdecoded(Logfunct.log,mpop);
                   Logfunct.newline(Logfunct.log);
                   incgencount();
                   main_r(select(mutate(crossover(mpop,pc,
                                              lambda),pm),mu)
                   )
            in
                main_r(mpop)
            end;
```

Für die drei Verfahren wurden mehrere Mutations–, Rekombinations– und Selektionsoperatoren implementiert. Beispielhaft wird hier der Mutationsoperator für den Genetischen Algorithmus vorgestellt (importiert aus `Operators/Mutation.sml`).

```
fun mutate(mpop,pm)=
    let
        fun mutate_r(mpop,pm,0)=mpop
```

```
        |mutate_r(mpop,pm,num)=
          mutate_r(setind(mpop,num,
                          mutate_ind(getind(mpop,num),pm)),
                          pm,num-1)
    in
         mutate_r(mpop,pm,getsize(mpop))
    end;
```

Die Funktion `mutate` : `pop` $\rightarrow$ `pop` mutiert alle Individuen der übergebenen Population. `mutate_r()` ist gegenüber `mutate()` um einen Index erweitert, der jeweils das zu mutierende Individuum bezeichnet. Analog ruft die Funktion `mutate_ind()`, die ein einzelnes Individuum mutiert, `mutate_ind_r()` auf, die den Index des jeweils zu mutierenden Atoms mitführt.

```
fun mutate_ind(mind,pm)=
    let
        fun mutate_ind_r(mind,pm,0)=mind
          |mutate_ind_r(mind,pm,pos)=
            mutate_ind_r(if random()<pm
                         then setatom(mind,pos,bit2atom(not(
                              atom2bit(getatom(mind,pos)))))
                       else mind,
                            pm,pos-1)
    in
        mutate_ind_r(mind,pm,indlength)
    end;
```

6.3.6 Initialisierung

Dem Initialisierungsoperator `initpop` wird vom Hauptoperator eine leere Population übergeben, die er mit einer bestimmten Anzahl von Individuen füllt. Dadurch kann die Größe der Population bestimmt werden, auf die der Hauptoperator angewendet wird. Diese werden im Prototyp noch zufällig erzeugt, könnten aber auch z. B. aus einer Datei eingelesen werden, um auf einen früheren Ablauf aufzusetzen. Die Anzahl der Individuen, die hier eingefügt werden, stehen in der Konstanten `initialpopsize`.

```
val initialpopsize = 10
```

Die Funktion `initpop` ruft dann die lokale Hilfsfunktion `addind` mit dieser Konstanten auf. Diese Funktion hängt die angegebene Zahl von Individuen

an eine übergebene Population an. Der folgende Programmausschnitt zeigt Realisierungen dieser beiden Funktionen.

```
fun addind (pop, 0) = pop
  | addind (pop, n) =
    ( let
        val (respop, _) = InitFunct.newind(addind(pop, n-1))
      in
        respop
      end
    )

fun initpop (pop) = addind(pop, initialpopsize)
```

6.3.7 Hauptprogramm

Das Hauptprogramm besteht zuerst aus einem Teil, in dem die einzelnen Module zusammengebunden werden. Diese werden in der Reihenfolge eingebunden, in der die Module aufeinander aufbauen. Zuerst wird das Grundlagenmodul eingebunden, dann die Funktionen zur Ausgabe, das Kodierungsmodul, die Definition von Individuen und Population, den Initialisierungsoperator und schließlich der Hauptoperator. Der folgende (gekürzte) Ausschnitt aus dem Programmtext zeigt diesen Vorgang.

```
use "Basics.sml";
use "Logfunct.sml";
use "Coding.sml";
use "Population_set.sml";
use Files.InitOpFileName;
use Files.MainOpFileName;
```

Der Rest des Hauptprogramms besteht dann im wesentlichen nur noch daraus, eine leere Population zu erstellen und auf diese zuerst den Initialisierungsoperator und dann den Hauptoperator anzuwenden. Die dadurch gewonnene Population und deren bestes Individuum werden dann noch als Ergebnis in die Log–Datei des Programms geschrieben. Die Realisierung in ML zeigt der folgende Programmteil. Die Definition der lokalen Hilfsfunktion `writeresult` wurde dabei ausgelassen. Diese dient nur dazu die Population und ihr bestes Individuum in die Log–Datei zu schreiben.

```
let
  val mainpop = PopIndFunct.makeemptypop()
  val initialpop = InitPop.initpop(mainpop)
```

```
in
  writeresult(Logfunct.log, MainOp.main(initialpop))
end;
```

6.4 Fazit und Ausblicke

In diesem Abschnitt sind die Eindrücke und Verbesserungsvorschläge gesammelt, die sich bei der Entwicklung des Prototyps ergeben haben.

Für die Koordination des endgültigen Systems sollte folgendes beachten werden. Die Trennung der einzelnen Module mit Hilfe der Signaturen hat zwar relativ gut funktioniert, es wäre jedoch sinnvoll, die Module schon früh kompilieren zu können. Dazu sollten die Module in der Reihenfolge erstellt werden, in der sie verwendet werden. Zumindest könnten in den einzelnen Modulen die wichtigsten der von außen benötigten Funktionen zuerst erstellt werden (anfangs mit eingeschränkter Funktionalität). Ebenso sollte bei der Unterteilung der Aufgaben die zeitliche Reihenfolge berücksichtigt werden, in der die Module benötigt werden und bei einer umfangreicheren Version ein (Zeit-)plan für den Ablauf erstellt werden.

Bei der Populationsverwaltung wurde gewünscht, daß mehr Informationen von den Funktionen zurückgegeben werden sollen (Bsp.: bestes Indidviduum mit Index zurückgeben). Ferner müssen Funktionen bereitgestellt werden, um ganze Populationen bzw. Teile davon zu kopieren, löschen usw. Die Fehlerbehandlung fehlt und muß ebenfalls bereitgestellt werden.

Da die Funktionen, die von den Operatoren verwendet werden, auch als Grundlage von selbstdefinierten Operatoren dienen sollen, ist dieser Teil entsprechend sorgfältig zu entwerfen. Bei der Realisierung des Operatormoduls für das endgültige System sollte folgendes beachtet werden:

Wichtig ist die *Spezifikation der Schnittstellen.* So müssen die Funktionen, die das System den Verfahren zur Verfügung stellt, sehr sorgfältig spezifiziert werden. Unklarheiten über die genaue Funktionalität traten häufig erst beim Programmieren auf, so z.B. ob die Numerierung der Individuen in einer Population bei Funktionen wie `getind()` oder `setind()` bei 0 oder 1 beginnt.

Ein weiteres Problem sind die *globalen Objekte.* Zum Teil wurden Ansätze direkt aus dem Entwurf für EAGLE übernommen, hier z.B. der globale Generationszähler. In einer funktionalen Sprache läßt sich dieser schlecht im-

plementieren, man sollte in diesem und ähnlichen Fällen überdenken, ob der Ansatz noch sinnvoll ist.

Im Prototyp sind die *Typen der Funktionsresultate* teilweise noch unterschiedlich. Es muß eine Möglichkeit geben, eindeutig Individuen in einer Population zu bezeichnen. Im Prototyp wurde das durch Indizes realisiert. Funktionen, die eine Population durch Einfügen oder Entfernen eines Individuums verändern, erwarten diesen Index als Argument. *Alle* Funktionen, die ein Individuum aus einer Population aussuchen, sollten nicht ein Individuum direkt, sondern dessen Index zurückliefern, um z.B. das Individuum anschließend aus der Population entfernen zu können. Das Individuum selbst erhält man mit der Funktion `getind`.

Wünschenswert ist auch die Vermeidung von *Indizes in Funktionsaufrufen.* Man sollte Funktionen wie `appendind` : `pop` $\times$ `ind` $\rightarrow$ `pop`, `getfirstind()` und `remfirstind()` vorsehen, um das Mitführen eines Index für die Individuen in vielen Funktionen zu vermeiden.

Parameter der Operatoren und Namen der Dateien mit den zu verwendenden Operatoren sind hier als Konstanten im Programmtext abgelegt. Im endgültigen System sollten diese Werte von außen an das Verfahren übergeben werden, so daß sie verändert werden können, ohne das Programm ändern zu müssen.

Für die Operatoren sollten *Strukturen und Signaturen* definiert werden, um einerseits Hilfsfunktionen vor dem System zu verbergen und andererseits eine definierte Schnittstelle zu den Operatoren zu haben.

Das Hauptprogramm schließlich besteht im Prototyp eigentlich nur aus einem Zusammenbinden der einzelnen Module. Die Einstellung der verwendeten Verfahren und Probleme geschieht hier in den einzelnen Modulen. Daher war schon beim Testen das Einstellen eines neuen Verfahrens recht mühsam, da meist in 3 Modulen Änderungen vorgenommen werden mußten. In späteren Versionen wird eine Steuerung nötig, mit der ein bestimmtes Problem, Kodierung und Operatoren ausgewählt und zu einem Experiment zusammengefaßt werden können.

Literaturverzeichnis

[AJJ+94] Frank Amos, Karsten Jung, Kurt Jaeger, Bernd Kawetzki, Wilfried Kuhn, Oliver Pertler, Ralf Reißing, and Markus Schaal. Zwischenbericht der Projektgruppe Genetische Algorithmen. Technical report, Universität Stuttgart, Fakultät Informatik, Institut für Informatik, Abteilung Formale Konzepte, 1994.

[AJK+95] Frank Amos, Karsten Jung, Bernd Kawetzki, Wilfried Kuhn, Oliver Pertler, Ralf Reißing, and Markus Schaal. Endbericht der Projektgruppe Genetische Algorithmen. Technical Report FK95/1, Universität Stuttgart, Fakultät Informatik, Institut für Informatik, Abteilung Formale Konzepte, 1995.

[Bar90] Hendrink Barendregt. *Functional Programming and Lambda Calculus*, pages 321–363. Elsvier Science Publishers B. V., 1990.

[Due93] G. Dueck. New optimization heuristics, the great deluge algorithm and the record-to-record travel. *Journal of Computational Physics*, 104:86–92, 1993.

[GD90] T. Scheuer G. Dueck. Threshold accepting – a general purpose optimization algorithm appearing superior to simulated annealing. *Fournal of Computanional Physics*, 90:161–175, 1990.

[HPS92] Th. Bäck H.-P. Schwefel. Künstliche evolution – eine intelligente problemlösungsstrategie ? , pages 1–20, 1992.

[JW74] Kathleen Jensen and Niklaus Wirth. *PASCAL : user manual and report*. Springer, 1974.

[JW95] Karsten Jung and Nicole Weicker. Funktionale Spezifikation des Software–Tools EAGLE. Technical Report FK 2/95, Universität Stuttgart, Fakultät Informatik, Institut für Informatik, Abteilung Formale Konzepte, 1995.

[Koh95] Michael Kohler. Analyse naturanaloger optimierungsverfahren. Master's thesis, Universität Stuttgart, Fakultät Informatik, 1995.

[Koz92] J. Koza. *Genetic Programming: on the programming of computers by means of natural selection.* MIT Press, 1992.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer Science*, pages 348–375, 1978.

[Sch81] Hans-Paul Schwefel. *Numerical optimization of computer models.* John Wiley & Sons, 1981.

[TB93] H.-P. Schwefel Th. Bäck. An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation, The Massachusetts Institute of Technologie*, pages 1–23, 1993.

[var90] various, editor. *Parallel Problem Solving from Nature, 1st Workshop – PPSN I.* Springer Verlag, 1990.

[var94] various, editor. *Parallel Problem Solving from Nature - PPSN*, Lectured Notes in Computer Science 866. Springer Verlag, 1994.

[VC94] A. Schwill (Hrsg.) V. Claus. *Schülerdiden „Die Informatik“.* Bibliographisches Institut, Mannheim, 1994.

[Wei95] N. Weicker. Naturanaloge optimierungsverfahren. Technical report, IfI Stuttgart, Formale Konzepte, Projektgruppe Genetische Algorithmen, 1995.

[Wir82] Niklaus Wirth. *Programming in MODULA-2.* Springer, 1982.

Anhang A

Glossar

A.1 Evolutionäres

Atom Ein Atom ist Teil einer *Kodierungsstruktur* oder einer *Problemstruktur* und legt einen Basistyp fest.

Crossover ist ein spezieller *Rekombinations*operator bei GA's.

Dekodierungsfunktion Ist eine Funktion, die eine *Belegung* der *Kodierungsstruktur* in eine Belegung der *Problemstruktur* abbildet.

Evolutionsstrategie Die Evolutionsstategien sind von I. Rechenberg entwickelte naturanaloge Verfahren. Sie dienen zur Optimierung konkreter technischer Probleme. Typischerweise verwenden sie eine *Population* möglicher Lösungen, die nach dem Vorbild der biologischen Evolution mutiert und rekombiniert werden. Sie treten dann in Konkurrenz zueinander, wobei die „fittesten" Lösungen ausgewählt werden.
Die Bewertung der *Fitneß* einer Lösung findet durch eine *Fitneß-Funktion* statt, die die Problemparameter zur Berechnung heranzieht. Daneben kann ein Element der *Population* aber auch noch *Strategie-Parameter* enthalten, die den weiteren Verlauf der Suche beeinflussen, daneben aber ebenfalls optimiert werden.

Extremum Lösung, die einen ausgezeichneten Wert hat, entweder in einer Umgebung (lokal) oder im gesamten Lösungsraum (global).

Fitness s. Qualität.

Genetischer Algorithmus GAs sind *Optimierungsverfahren,* die Vorgänge

in der Biologie nachbilden. Sie arbeiten auf einer Population von Individuen, die als Binärstring kodiert dargestellt werden. Die Individuen werden mutiert und gekreuzt, wobei man hofft, daß sich die guten Eigenschaften weitervererben. Die Bewertung der Individuen erfolgt durch eine *Fitneß-Funktion*, die auch die Auswahl der Individuen beeinflußt.

Genotyp ist die konkrete Belegung einer *Kodierungsstruktur*

Great-Deluge Algorithmus (Sintflut-Algorithmus) Dieses Optimierungsverfahren betrachtet einen Punkt des Lösungsraums und wählt einen neuen Punkt aus dessen Umgebung. Die *Qualität* des neuen Punkts wird berechnet und der neue Punkt übernommen, so seine Qualität über einem „Pegel“ liegt. Wird der Punkt übernommen, dann wird der Pegel erhöht.
Anschaulich kann man sich den Ablauf dieses Verfahrens als Überflutung einer Landschaft vorstellen. Während „das Wasser steigt“, versucht die Lösung sich auf den höchsten Punkt, also das Maximum, zu retten.

Individuum Element einer Population. Kann neben Problemparametern auch Strategieparameter enthalten.

Kodierung 1. Die Kodierung besteht aus Kodierungsstruktur und Kodierungsfunktion.

2. Darstellung eines Wertes (verfahrensabhängig).

Kodierungsfunktion Ist eine Abbildung von einer Problemstruktur auf eine Kodierungsstruktur.

Kodierungsstruktur Die Kodierungsstruktur besteht aus einer Anzahl von Atomen. Eine Kodierungsstruktur ist genau dann mit einem Problem verträglich, wenn eine Kodierungsfunktion existiert, die die Problemstruktur auf die Kodierungsstruktur abbildet.

Lösung Parameterbelegung, die ein Problem löst. (Keine Aussage über die Qualität.)

Lösungsraum Menge der darstellbaren (nicht notwendig möglichen) Lösungen.

Maximum Extremum mit dem größten Wert.

Minimum Extremum mit dem kleinsten Wert.

Mutation ist ein *Operator*, der ein Individuum verändert.

naturanaloge Verfahren Optimierungsverfahren, die Vorgänge in der Natur (physikalischer oder biologischer Art) zum Vorbild haben.

Operator (auch genetischer Operator) ist eine Funktion, die aus einem oder mehreren Individuen neue Individuen produziert, z.B. *Mutation, Crossover, Selektion, Fitness.*

Optimierungsverfahren Verfahren, die eine optimale Parameterbelegung für eine Funktion suchen.

Phänotyp ist die konkrete Belegung einer *Problemstruktur.*

Population Menge von Individuen.

Problem Zu optimierende/„lösende" Aufgabe, mathematisch formuliert. Besteht aus Problemstruktur und Fitneßfunktion.

Problemparameter Parameter, der in die Berechnung der Qualität eines Individuums eingeht.

Problemraum s. Lösungsraum.

Problemstruktur ist eine Struktur, die ein Individuum von der Problemseite beschreibt. Dekodierung der Kodierungsstruktur.

Record-to-Record Travel Optimierungsverfahren, Variante des *Great-Deluge Algorithmus*: Statt eines Pegels wird der bisherige Bestwert für Vergleiche herangezogen. Die *Fitneß* eines neuen Punkts im *Lösungsraum* wird als neuer Bestwert übernommen, wenn sie besser oder nur geringfügig schlechter als der bisherige Bestwert ist.

Rekombination ist ein *Operator*, der aus zwei oder mehreren Individuen ein oder mehr neue Individuen produziert.

Selektion ist ein *Operator*, der aus einer Anzahl von *Individuen* (*Population*) eine neue Anzahl von *Individuen* (*Population*) auswählt.

Simulated Annealing Naturanaloges Optimierungsverfahren, das die physikalischen Vorgänge in einer abkühlenden Schmelze nachvollzieht.
In der lokalen Umgebung eines Punktes im Suchraums wird ein neuer Kandidat bestimmt. Dieser wird übernommen, so seine Fitneß besser als die des Ausgangspunkts ist. Ist der neue Punkt schlechter, so wird er mit einer von der „Temperatur" abhängigen Wahrscheinlichkeit dennoch übernommen. Dies soll verhindern, daß die Suche in einem lokalen Optimum stecken bleibt.

Strategieparameter sind die Elemente der *Kodierungsstruktur*, die das Verhalten der *Operatoren* steuern. Diese werden eventuell zunächst als *zusätzliche Parameter* dekodiert.

Threshold Algorithmus Dieses Optimierungsverfahren ist eine Vereinfachung des *Simulated Annealing*. Hierbei wird ein Punkt des Lösungsraums betrachtet und ein weiterer Punkt aus dessen Umgebung bestimmt. Die *Qualität* dieser Punkte wird verglichen und der neue Punkt ersetzt den alten, so seine *Qualität* besser ist, oder nur um einen Schwellwert schlechter. Dieser Schwellwert wird im Laufe der Berechnung verkleinert.
Im Vergleich zum *Simulated Annealing* entfallen die Wahrscheinlichkeitsberechnungen, wodurch der Ablauf beschleunigt wird.

Verfahrensparameter sind die *Parameter*, die jeweils für ein ganzes *Experiment* in der *Laufinitialisierung* festgelegt werden.

A.2 Allgemeines

Ausdruck Ein Wort aus der Sprache, die von der in Kapitel 3.3.4.1 definierten Grammatik erzeugt wird.

DeJong Entwickelte die DeJong'schen Funktionen, die zur Klassifizierung von *Lösungsverfahren* verwendet werden.

eva Abkürzung für die Projektgruppe Evolutionäre Algorithmen.

Grunddatentyp Die vom System vordefinierten Datentypen. Wie z. B. Boolean-, Integer- und Real-Datentypen.

ICGA International Conference on Genetic Algorithms
Konferenz, die alle zwei Jahre stattfindet.

Instanz Belegung einer allgemeinen Darstellung mit konkreten Werten.

kooperatives Multitasking Verfahren zur Realisation von Parallelverarbeitung. Hierbei muß ein Thread die Kontrolle an das System explizit zurückgeben, so daß ein anderer Thread fortgesetzt werden kann. Gegensatz zu *präemptivem Multitasking*.

NP-vollständig/hart Probleme der Klasse NP können von einem nichtdeterministischen Automaten in polynomieller Zeit gelöst werden. Ein Problem ist NP-hart, wenn jedes andere Problem aus NP in dieses

Problem mit polynomiellem Aufwand transformiert werden kann. Liegt dieses Problem zudem in NP, so bezeichnet man es als NP-vollständig. Somit liegen die schwierigsten Probleme in diesen Klassen.

Optimierungsverfahren Verfahren, die möglichst optimale Parameter für eine Funktion suchen.

PPSN Parallel Problem Solving by Nature
Konferenz, die alle zwei Jahre stattfindet.

präemtives Multitasking Verfahren zur Realisation von Parallelverarbeitung. Hierbei wird ein *Thread* vom System gestoppt und die Kontrolle einem anderen übergeben. Gegensatz zu *kooperativem Multitasking*.

Typpolymorphismus Typpolymorphismus ist eine Art der Typzuweisung, bei der einem Ausdruck bei verschiedenem Auftreten verschiedene Typen zugewiesen werden können.

Index