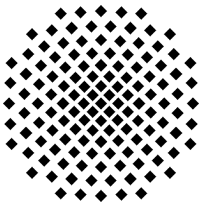
Abschlußbericht der
Projektgruppe Genetische Algorithmen
Technischer Bericht: FK 1/95



Universität Stuttgart

Projektgruppe
Genetische Algorithmen

Abschlußbericht der Projektgruppe Genetische Algorithmen

Frank Amos, Karsten Jung, Bernd Kawetzki,
Wilfried Kuhn, Oliver Pertler, Ralf Reißing, Markus Schaal

Betreuung
Prof. Dr. Volker Claus
Dipl.-Math. Nicole Weicker
Abteilung Formale Konzepte
Fakultät Informatik
Universität Stuttgart

01. Mai 1995

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

Abbildungsverzeichnis

Kapitel 1

Einleitung

Der Weg ist das Ziel
(Volksmund)

Als wir Studenten uns im April 1994 zur ersten Vorbesprechung der Projektgruppe trafen, wußte noch keiner von uns, was genau auf ihn zukommen würde. Gemeinsam war uns das Interesse, in einer größeren Gruppe (einem Team) an einem Thema zu arbeiten. Eine Möglichkeit, die das Studium der Informatik an der Universität Stuttgart sonst nicht bietet. Die einzigen Lehrveranstaltungen, bei denen eine Zusammenarbeit mehrerer Studenten vorgesehen ist, nämlich Software-Praktikum und Fachpraktikum, haben eine Gruppengröße von zwei, maximal drei Studenten.

Die Fähigkeit zur Teamarbeit gewinnt zunehmend an Bedeutung und ist heutzutage im beruflichen Alltag ebenso wichtig wie Fachkompetenz oder Einsatzbereitschaft. Die durch die Arbeit in der Gruppe erworbene soziale Kompetenz schlägt sich zwar nicht direkt in dem Endprodukt einer Projektgruppe nieder, doch ist sie eine wichtige Motivation für die Einführung solcher Lehrveranstaltungen.

Dieser Bericht ist nicht nur die Dokumentation der nach einem Jahr Projektarbeit erzielten Ergebnisse, sondern schildert auch den Ablauf der Projektgruppe und stellt die gewonnenen Einsichten der Mitglieder zusammen. Der Aufbau dieses Berichts ist wie folgt: Dieses Kapitel erläutert zuerst die Aufgaben und Ziele einer Projektgruppe im allgemeinen, dann diejenigen dieser Projektgruppe. Das Thema dieser Projektgruppe, Evolutionäre Algorithmen (EA), wird in Kapitel 2 vorgestellt. In Kapitel 3 wird die geleistete Arbeit dokumentiert. Kapitel 9.1 bietet Einblick in den historischen Ablauf, Kapitel 9.2 faßt dann unsere Erfahrungen und Eindrücke mit dieser Lehrveranstaltung zusammen.

Das Thema der Projektgruppe war für uns alle in hohem Maße interessant und motivierend, wie es auch die Arbeit in der Gruppe war. Unser Dank gilt Professor Volker Claus, der durch seinen Einsatz die Projektgruppe erst ermöglicht hat und auch

das „unternehmerische Risiko“ trug. Besonders aber möchten wir unserer Betreuerin Nicole Weicker danken, die sich stark in der Projektgruppenarbeit engagierte.

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 (nach Prof. Claus):

- Arbeiten im Team
- Selbstständige Erarbeitung von Lösungsvorschlägen und deren Vorstellung und Verteidigung in einer Gruppe
- Analyse von Problemen, Strukturierung von Lösungen und gemeinsamer Entwurf geeigneter Systeme
- 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 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 Projekt-

gruppen z.T. schon Pflichtveranstaltungen im Rahmen des Informatikstudiums.

1.2 Die Projektgruppe Genetische Algorithmen

1.2.1 Aufgabenstellung der Projektgruppe Genetische Algorithmen

Aufgabe der Projektgruppe war die Erstellung eines Systems zur Bearbeitung hartnäckiger (NP–harter) Probleme mit Hilfe von Evolutionsstrategien und Genetischen Algorithmen (siehe Kapitel 2). Dabei waren folgende Punkte besonders zu beachten:

- Beachtung von Methoden des Software–Engineerings
- Nutzung von Werkzeugen (zur Kommunikation, Planung, Software–Entwicklung, ...)
- Erstellen einer Testumgebung
- Durchführung von Experimenten, Auswertung der Ergebnisse
- Erstellung einer ausführlichen Dokumentation nicht nur des Ergebnisses, sondern auch des Arbeitsablaufes
- Restrukturierungsvorschläge zum System

Das System sollte 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 Zielsetzungen der Projektgruppe Genetische Algorithmen

Die vorgegebene Aufgabe wurde von der Projektgruppe dahingehend erweitert, daß auch andere stochastische Optimierungsverfahren (z.B. Simulated Annealing, Monte–Carlo–Verfahren ..., siehe auch hierzu Kapitel 2) leicht an das zu erstellende System anzubinden sein sollten.

Gerade im Bereich der Parametereinstellungen gibt es wenige Erkenntnisse, weder empirischer noch theoretischer Art. Ansätze zur Untersuchung der Parametereinstellungen gibt es durch das von Greffenstette [Gre86] vorgestellte Meta–Verfahren, in dem sich die Parameter in einem übergeordneten GA selbst optimieren. Ein solches

Meta–Verfahren ist allerdings sehr zeitaufwendig und gibt nur Aussagen für das konkrete Problem, auf das es angesetzt wurde. Es wäre ein System von Nutzen, das es erlaubt, beliebige Verfahren mit beliebigen Parametereinstellungen zu testen.

Vergleichende empirische Untersuchungen sollten theoretische Erkenntnisse ermöglichen und bei deren Überprüfung helfen. Die Projektgruppe setzte sich hier zum Ziel, ein benutzerfreundliches System zu entwickeln, das es ermöglichen soll, mit geringen Kenntnissen herkömmlicher Programmiersprachen (Pascal, Modula–2) ein anwenderdefiniertes Problem einzubinden und sich das Verfahren aus Operatoren selbst zusammenzustellen.

Diese Überlegungen führten schließlich zur Anforderungsspezifikation eines Systems, das von der Projektgruppe „EAGLE“ getauft wurde: Evolutionary Algorithms Gaming and Learning Environment.

Kapitel 2

Evolutionäre Algorithmen

Evolutionäre Algorithmen stellen die neue Hoffnung auf dem Gebiet der Parameteroptimierung für „schwierige“ Probleme dar. Konventionelle Optimierungsverfahren können bei vielen Problemen aufgrund der Struktur ihrer Lösungsräume nicht mehr effizient oder effektiv eingesetzt werden. Hier können Evolutionäre Algorithmen wegen der gesteuerten Zufallssuche, die im Idealfall immer bessere Lösungen (bis hin zum Optimum) findet, weiterhelfen.

In diesem Kapitel sollen kurz die Grundlagen der Optimierung erläutert werden. Darauf folgt die Herausarbeitung einer Hierarchie von gebräuchlichen Optimierungsverfahren, um die Evolutionären Algorithmen besser einordnen zu können. Auf diese speziellen evolutionären Optimierungsverfahren wird dann näher eingegangen. Dabei stehen die zwei wichtigsten Verfahren, Evolutionsstrategien (ES) und Genetische Algorithmen (GA), im Vordergrund.

2.1 Optimierung

Bei der *Optimierung* geht es darum, zu einem gegebenen Problem eine „beste“ Lösung (eine optimale Lösung) zu finden. Hier soll ein besonderer Fall der Optimierung, die Parameteroptimierung, näher erläutert werden. Bei der Parameteroptimierung läßt sich das Problem durch eine Reihe von Parametern darstellen, die verschiedene Werte aus $D = \mathbb{R} \cup \mathbb{Z} \cup \mathbb{B}$ annehmen können. Dabei ist $\mathbb{R}$ die Menge der reellen Zahlen, $\mathbb{Z}$ die Menge der ganzzahligen Zahlen und $\mathbb{B}$ die Menge der boolschen Werte 0 und 1. Dabei wird 0 mit falsch assoziiert, 1 mit wahr.

Um bei der Vorstellung der Verfahren eine einfachere Darstellung zu erhalten, wird dort $D = \mathbb{R}$ angenommen. Diese Einschränkung vermindert die Mächtigkeit der Verfahren nicht, da sich alle Werte aus $\mathbb{Z}$ oder $\mathbb{B}$ durch disjunkte Wertemengen aus $\mathbb{R}$ darstellen lassen.

Eine *Lösung* eines Parameterproblems läßt sich als Vektor $\vec{x} = (x_1, \ldots, x_n)$ von Werten $x_i \in D$ der Parameter auffassen. Dabei ist $n \in \mathbb{N}$ die Anzahl der Parameter des Problems, die hier als endlich angenommen wird. Die Menge $M \subseteq D^n$ aller möglichen Lösungen wird als *Lösungsraum* oder *Suchraum* bezeichnet. Jeder Lösung $\vec{x}$ kann über eine Zielfunktion $f : M \to \mathbb{R}$ eine Bewertung $f(\vec{x})$ zugeordnet werden. Ohne Einschränkung der Allgemeinheit ist die Bewertung $f(\vec{x})$ ist um so kleiner, je besser die Lösung $\vec{x}$ ist.

Ziel der Optimierung ist es, ein *globales Minimum* $\vec{x}_{min} \in M$ zu finden. Für dieses gilt:

$$\forall \vec{x} \in M : f(\vec{x}_{min}) \leq f(\vec{x}),$$

d.h. es gibt keine Lösung $\vec{x}$, die eine niedrigere Bewertung als $f(\vec{x}_{min})$ besitzt. Dabei ist zu beachten, daß es durchaus mehrere globale Minima geben kann, so daß die beste Lösung nicht eindeutig bestimmt ist.

Neben den globalen Minima gibt es auch *lokale Minima* $\vec{x}_l$, für die gilt:

$$\exists \varepsilon \in \mathbb{R}\ \forall \vec{x} \in M, d(\vec{x}, \vec{x}_l) \leq \varepsilon : f(\vec{x}_l) \leq f(\vec{x}).$$

Dabei ist die Funktion $d : M \times M \to \mathbb{R}$ ein Maß für den Abstand zweier Lösungen (zum Beispiel der euklidische Abstand $\sqrt{\sum_{i=1}^{n}(x_{li} - x_i)^2}$). Die Bedingung für lokale Minima bedeutet, daß in einer ε–Umgebung (definiert durch d) von $\vec{x}_l$ keine Lösung mit besserer Bewertung existiert. Nach dieser Definition sind alle globalen Minima auch lokale Minima, da ihre Minimalität für beliebige $\varepsilon \in \mathbb{R}$ gilt. Die lokalen Minima stellen bei der Optimierung ein Problem dar, da es Verfahren gibt, die in ihnen „hängen bleiben“. Dies bedeutet, daß die Suche nicht mehr aus dem lokalen Optimum herausführt, weil in der näheren Umgebung des lokalen Optimums nur schlechtere Lösungen liegen.

Diese Definition für Optimierungsverfahren läßt sich auch auf Maximierungsprobleme anwenden, indem dort statt der Zielfunktion f die Zielfunktion $f' = -f$ betrachtet wird. Dadurch wird aus dem Maximierungsproblem mit der Zielfunktion f ein Minimierungsproblem mit der Zielfunktion f'.

Bekannte Beispiele für Optimierungsprobleme gibt es in vielen Bereichen. Im Bereich des Ingenieurwesens sind dies zum Beispiel die Suche nach Fahrzeugen mit minimalen Luftwiderstandswerten (c_w–Werten) oder nach Düsen mit minimalem Strömungswiderstand. Im Bereich der Wirtschaftswissenschaften ist das am meisten erforschte Optimierungsproblem die Minimierung von Bearbeitungszeiten in der Produktion.

2.2 Klassen von Optimierungsverfahren

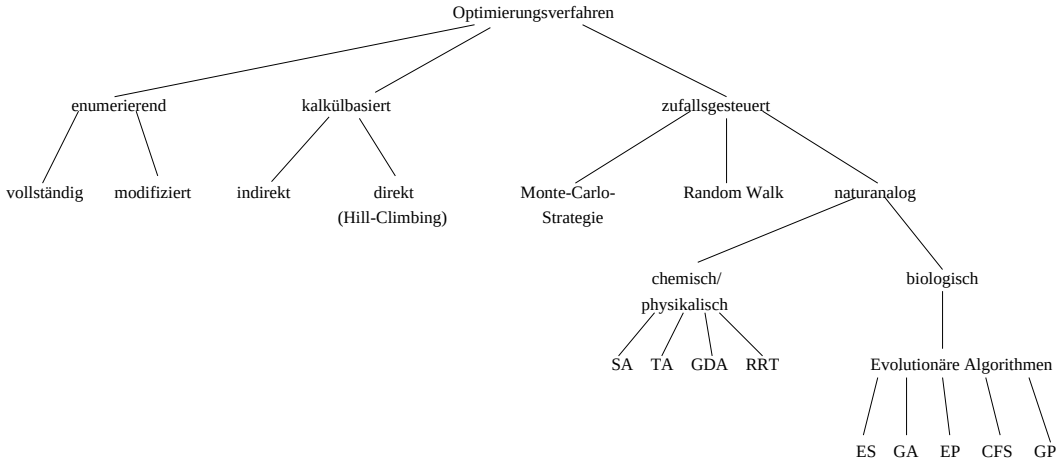


Abbildung 2.1: Hierarchie der Optimierungsverfahren

Abbildung 2.1 zeigt einen Überblick über die Hierarchie der bekannten Optimierungsverfahren, die sich an der Darstellung von Goldberg in [Gol89] orientiert. Goldberg unterscheidet drei verschiedene Ansätze: enumerative, kalkülbasierte und zufallsgesteuerte Verfahren.

Die Klassen der *enumerierenden* und der *kalkülbasierten* Verfahren sind dadurch charakterisiert, daß die Suche nach einem oder allen globalen Optima bei gleichen Anfangsbedingungen immer gleich verläuft. Bei den *zufallsgesteuerten* Verfahren spielt jedoch der Zufall eine wesentliche Rolle, so daß das Suchverhalten im allgemeinen nicht vorhergesagt und nur schwierig reproduziert werden kann.

2.2.1 enumerierende Verfahren

Beim *enumerierenden* Ansatz werden alle möglichen Lösungen untersucht. Die beste Lösung wird dadurch garantiert gefunden. Zur Untersuchung aller Lösungen müssen diese nacheinander erzeugt und bewertet werden. Da bei realen Problemen die Zahl der möglichen Lösungen jedoch sehr groß werden kann, sind die vollständig enumerierenden Verfahren dort nicht effizient einsetzbar.

Daher wurde versucht, die vollständig enumerierenden Verfahren durch Integration von Problemwissen, zum Beispiel Heuristiken, so zu modifizieren, daß möglichst wenig Lösungen betrachtet werden müssen, aber dennoch das Auffinden des globalen Optimums (der besten Lösung) garantiert bleibt. Ein bekanntes Beispiel für ein solches modifiziertes enumerierendes Verfahren ist der Branch–and–Bound–Algorithmus [Sed91].

2.2.2 kalkülbasierte Verfahren

Goldberg unterscheidet hier zwei verschiedene Ansätze: den indirekten und den direkten Ansatz.

Beim *indirekten* Ansatz werden Optima durch mathematisch fundierte Verfahren ermittelt. Dies geschieht im allgemeinen durch ein Nullsetzen des Gradienten der Zielfunktion und Lösung des sich ergebenden Gleichungssystems für die Parameter. Die sich ergebenden Lösungen sind die lokalen Optima. Durch Vergleich der Bewertungen der gefundenen Optima lassen sich die globalen Optima identifizieren. Voraussetzung für die Anwendung dieses Verfahrens ist selbstverständlich die Möglichkeit zur Bestimmung der Gradienten und die Lösbarkeit des Gleichungssystems. Die Bezeichnung „indirekt" kommt vermutlich daher, daß die Suche nicht direkt den Suchraum bearbeitet, sondern die ihn beschreibende Zielfunktion zur Suche nach den Optima verwendet.

Beim *direkten* Ansatz findet die Suche nach der besten Lösung durch gezielte Schritte im Suchraum statt. Ausgehend von einer zufällig ausgewählten Lösung wird in der Umgebung dieser Lösung die Richtung gesucht, in der der steilste Anstieg (der größte Gradientenwert) der Zielfunktion zu verzeichnen ist. In dieser Richtung wird die nächste Lösung gewählt, mit der das Verfahren weiter iteriert wird. Das Verfahren terminiert, wenn es keine Richtung mehr gibt, in der sich bessere Lösungen finden lassen. Man spricht dann davon, daß das Verfahren gegen die momentane Lösung konvergiert ist.

Dieser Ansatz ist ein *Hill–Climbing–Verfahren* [Sch81], das zielsicher das nächstgelegene lokale Optimum findet. Da dieses jedoch kein globales Optimum sein muß, ist im Gegensatz zu den enumerierenden Verfahren die Optimalität der gefundenen besten Lösung nicht mehr garantiert. Dafür müssen jedoch in den meisten Fällen weniger Lösungen bis zur Konvergenz des Verfahrens untersucht werden.

2.2.3 zufallsgesteuerte Verfahren

Bei den zufallsgesteuerten Verfahren — die auch als *stochastische* Verfahren bezeichnet werden — wird der Zufall nicht nur zur Erzeugung einer Anfangslösung eingesetzt, sondern auch bei der anschließenden Suche nach besseren Lösungen.

Die einfachste Suchstrategie ist dabei die *Monte–Carlo–Strategie* [SB92]. Hier werden zufällig verschiedene Lösungen erzeugt und bewertet. Wird eine bessere Lösung als bisher gefunden, wird sie gemerkt. Das Verfahren wird solange fortgesetzt, bis ein bestimmtes Kriterium (z.B. das Erzeugen einer bestimmten Anzahl von Lösungen) erreicht wird.

Eine alternative Vorgehensweise, die als *Random Walk* bezeichnet wird, bewegt sich mit zufälligen Schritten durch den Lösungsraum. Die nächste Lösung wird aus der

zuletzt betrachteten durch „kleine“, zufällige Änderungen erzeugt. Die beste auf dem Weg gefundene Lösung wird gemerkt. Auch hier ist das Abbruchkriterium in der Regel die Anzahl der erzeugten Lösungen.

Diese völlig zufälligen Verfahren lassen keinerlei Aussagen über die Optimalität der gefundenen Lösungen zu. Es muß sich nicht einmal um lokale Optima handeln.

Im folgenden Abschnitt sollen nun einige neuere Ansätze der zufallsgesteuerten Suche, die naturanalogen Verfahren, näher untersucht werden.

2.2.3.1 naturanaloge Verfahren

Die *naturanalogen* Verfahren sind an Optimierungsverfahren der Natur angelehnt. Bei diesen Verfahren gibt es zwei wichtige Strömungen. Die eine orientiert sich an chemischen und physikalischen Modellen, während die andere von biologischen Modellen abgeleitet sind.

chemische/physikalische Verfahren Bekannte Vertreter dieses Ansatzes sind die folgenden:

- *Simulated Annealing* (SA) [dGWH90],
- *Threshold Algorithmus* (TA) [DS90],
- *Great Deluge Algorithmus* (Sintflut–Algorithmus, GDA) [Due93] und
- *Record to Record Travel* (RRT) [Due93].

Diesen Verfahren ist gemeinsam, daß sie mit einer Arbeitslösung $\vec{x}$ arbeiten, aus der durch eine kleine, zufällige Änderung eine neue Lösung $\vec{x}'$ erzeugt wird. Danach wird anhand bestimmter Kriterien entschieden, welche der beiden Lösungen zur neuen Arbeitslösung gemacht wird, mit der das Verfahren weiter iteriert wird.

Falls die Lösung $\vec{x}'$ besser als $\vec{x}$ ist, wird sie auf jeden Fall zur neuen Arbeitslösung. Die genannten Verfahren unterscheiden sich lediglich in den Kriterien, die die Übernahme einer schlechteren Lösung $\vec{x}'$ als neue Arbeitslösung gestatten. Diese Akzeptanz schlechterer Lösungen unter bestimmten Bedingungen soll es den Verfahren erlauben, lokale Optima wieder zu verlassen.

Als ein Vertreter der chemisch/physikalischen Verfahren wird im folgenden das SA näher vorgestellt.

In Abbildung 2.2 ist ein informeller Algorithmus für das SA angegeben. Dieses Verfahren fußt auf einem Modell des Abkühlungsprozesses einer Metallschmelze.

wähle eine Ursprungslösung $\vec{x}$
berechne die Bewertung $f(\vec{x})$ von $\vec{x}$
wähle eine Anfangstemperatur $T > 0$
repeat
    repeat
        erzeuge neue Lösung $\vec{x}'$ durch kleine zufällige Änderung von $\vec{x}$
        berechne die Bewertung $f(\vec{x}')$ von $\vec{x}'$
        berechne den Bewertungsunterschied $\Delta E := f(\vec{x}') - f(\vec{x})$
        if $\Delta E < 0$ **then** // $f(\vec{x}') < f(\vec{x})$
            $\vec{x} := \vec{x}'$ // neue, bessere Lösung übernehmen
        else if Zufallszahl $< e^{-\frac{\Delta E}{k \cdot T}}$ **then** // k: Boltzmannkonstante
            $\vec{x} := \vec{x}'$ // neue, schlechtere Lösung übernehmen
    until lange keine Verbesserung der Bewertung
    verringere T, $T \geq 0$
until lange keine Verbesserung der Bewertung oder $T = 0$

Abbildung 2.2: Algorithmus des Simulated Annealing

Zu Beginn befinden sich die Atome in der Schmelze in völliger Unordnung. Beim Abkühlen streben die Atome die energetisch günstigsten Plätze an. Das sind die Plätze mit minimaler Energie, die in einer regelmässigen Struktur, meistens einem Gitter, angeordnet sind. Dadurch ergibt sich eine regelmäßige Struktur, wenn jedes Atom seinen optimalen Platz einnimmt.

Um von einem Platz in einem lokalen Energieminimum zu einem Platz niedrigerer Energie gelangen zu können, müssen die Atome einen energetisch ungünstigeren Zwischenzustand durchlaufen. Dies ist bei hoher Temperatur leichter als bei niedriger Temperatur, da hier die temperaturbedingte Bewegung der Atome höher ist. Sinkt die Temperatur, werden Lageänderungen der Atome immer unwahrscheinlicher.

Dies schlägt sich im Algorithmus wie folgt nieder: Die Position eines Atoms im Raum wird durch eine Lösung $\vec{x}$ im Suchraum repräsentiert. Die temperaturbedingte Schwingung eines Atoms wird in eine kleine zufällige Änderung der Arbeitslösung umgesetzt. Die Größe der Änderung kann dabei von T abhängig gemacht werden, um nahe am Modell zu bleiben.

Eine neu erzeugte Lösung $\vec{x}'$ wird auf jeden Fall akzeptiert, wenn sie besser als die momentane Arbeitslösung $\vec{x}$ ist. Mit einer von der Bewertungsdifferenz ΔE abhängigen Wahrscheinlichkeit wird aber auch eine schlechtere Lösung akzeptiert. Diese Wahrscheinlichkeit ist außerdem von der Temperatur T abhängig, die langsam verringert wird. Deshalb wird die Akzeptanz schlechterer Lösungen mit zunehmender Dauer

der Anwendung des Verfahrens immer unwahrscheinlicher, da gilt:

$$\lim_{T \longrightarrow 0} e^{-\frac{\Delta E}{k \cdot T}} = 0 \text{ für } \Delta E > 0, T \geq 0.$$

Für $\Delta E = 0$ ergibt sich für beliebiges $T \geq 0$ eine Übernahmewahrscheinlichkeit von 1. Dies bedeutet, daß eine gleich gute Lösung auf jeden Fall übernommen wird. Dies ist sinnvoll, um auf Ebenen gleicher Energie (Energieplateaus) dennoch den Suchvorgang fortsetzen zu können, anstatt bei der alten Lösung zu verharren.

Eine einfache, vergleichende Einführung mit der Anwendung der ersten drei genannten Verfahren (SA,TA,GDA) auf das Travelling–Salesman–Problem findet sich in [Ott94].

biologische Verfahren Die bekanntesten Verfahren, die auf biologischen Modellen beruhen, sind diejenigen, die die Evolution einer Art simulieren. Eine Menge von Lösungen wird einem Evolutionsprozeß unterworfen, bei dem die natürliche Auslese (*natural selection*), eine Gesetzmäßigkeit der Evolutionstheorie von Charles Darwin [Dar59], zugrunde liegt. Durch die Generierung neuer Lösungen aus den vorhandenen Lösungen und Auslese der besten Lösungen sollen immer bessere Lösungen gefunden werden. (Eine Einführung in die Grundlagen der Evolution findet sich in [SHF94].)

Die wichtigste Klasse der biologischen Verfahren ist die der *Evolutionären Algorithmen* (EA). Diese umfaßt die folgenden Teilbereiche [HB94]:

- *Genetische Algorithmen* (GA) [Hol75] [Gol89] [Dav91],
- *Evolutionsstrategien* (ES) [Rec73] [Sch81],
- *Evolutionäre Programmierung* (EP) [FOW66] [Fog92],
- *Classifier Systems* (CFS) [Hol75] [Gol89] und
- *Genetische Programmierung* (GP) [Koz92].

Die Verfahren der Classifier Systems und der Genetischen Programmierung lassen sich den Genetischen Algorithmen zurechnen, da sie sich zur Erzeugung neuer Lösungen eines GA bedienen [HB94]. Die CFS dienen der Optimierung von regelbasierten Entscheidungsverfahren, während bei der GP optimale (meistens: kürzeste) Programme zur Lösung einer Aufgabe gesucht werden. Diese beiden Verfahren werden hier nicht weiter betrachtet, da die von ihnen bearbeiteten Probleme keine direkten Parameteroptimierungsprobleme darstellen.

Die Evolutionäre Programmierung befaßte sich ursprünglich mit der Suche nach optimalen endlichen Automaten für eine gegebene Aufgabe, wurde jedoch später auf

Parameteroptimierungsprobleme mit reellen Parameter erweitert [Fog92]. Auf dieses Verfahren wird im folgenden ebenfalls nicht weiter eingegangen, da wir uns hier auf die universellsten und damit am häufigstigen eingesetzten Verfahren, Genetischen Algorithmen und Evolutionsstrategien, konzentrieren möchten.

2.3 Grundlagen der Evolutionären Algorithmen

Evolutionäre Algorithmen arbeiten mit einer Multimenge von Lösungen, die in Anlehnung an das zugrundeliegende biologische Modell und seiner Begriffe *Population* genannt wird. Da sich Populationen aus Individuen zusammensetzen, werden die einzelnen Lösungen auch als *Individuen* bezeichnet. Die Individuen beinhalten die konkrete Wertebelegung der Parameter des Optimierungsproblems, wobei diese Werte auch in einer kodierten Form vorliegen können.

Jedem Individuum kann eine Bewertung zugeordnet werden, die hier *Fitneß* genannt wird. Falls die Parameter des Optimierungsproblems kodiert vorliegen, muß vor der Bewertung noch eine Dekodierung durchgeführt werden.

Das Vorgehen der EA sieht wie folgt aus: Eine zufällig gewählte Anfangspopulation wird einem Evolutionsprozeß unterworfen, bei dem Individuen mit guter Bewertung bei der Fortpflanzung bevorteilt werden. Daher setzen sich — theoretisch — Individuen mit guten Eigenschaften und damit Individuen mit hoher Fitneß auf lange Sicht durch.

Bei der Fortpflanzung entstehen durch Austausch von Information zwischen Individuen und zufällige Veränderungen einzelner Individuen neue Individuen (Nachkommen), die sich bei entsprechend guter Bewertung durchsetzen können. Schlechte Individuen fallen mit großer Wahrscheinlichkeit der „natürlichen Auslese“ zum Opfer. Durch dieses Vorgehen hofft man, zumindest in die Nähe einer optimalen Lösung vordringen zu können, ohne große Bereiche des Lösungsraums absuchen zu müssen.

Der folgende Abschnitt soll die Arbeitsweise Evolutionärer Algorithmen genauer erläutern und formalisieren. Die formalisierte Darstellung ist an [BS93] angelehnt. Abbildung 2.3 enthält eine Zusammenfassung der wichtigsten Notationen.

2.3.1 allgemeines Schema

Zu Beginn wird eine zufällige Population $P(0) \in I^\mu$ der Größe μ erzeugt, wobei die Individuen $\vec{a}_i(0) \in P(0), 1 \leq i \leq \mu$, durch die Fitneßfunktion $\Phi : I \rightarrow I\!R$ bewertet werden (I ist die Menge der möglichen Individuen). Danach können sich die Individuen fortpflanzen (reproduzieren), wobei ihre Fortpflanzungschancen von ihrer Fitneß $\Phi(\vec{a}_i(0))$ abhängen.

Notation	Beschreibung
I	Individuenraum
$\vec{a} \in I$	ein Individuum
$\vec{x} \in D^n$	Vektor von Objektvariablen
$f : D^n \to \mathbb{R}$	zu optimierende Funktion
$\Phi : I \to \mathbb{R}$	Fitneßfunktion
μ	Populationsgröße (Elternpopulation)
λ	Populationsgröße (Nachkommenpopulation)
$P(t) = \{\vec{a}_1(t), \ldots, \vec{a}_\mu(t)\}$	Population zum Zeitpunkt t
Θ_r	Menge der Parameter der Rekombination
$r_{\Theta_r} : I^\mu \to I^\lambda$	Rekombinationsoperator (globale Form)
$r'_{\Theta_r} : I^\mu \to I$	Rekombinationsoperator (lokale Form)
Θ_m	Menge der Parameter der Mutation
$m_{\Theta_m} : I^\lambda \to I^\lambda$	Mutationsoperator (globale Form)
$m'_{\Theta_m} : I \to I$	Mutationsoperator (lokale Form)
Θ_s	Menge der Parameter der Selektion
$s_{\Theta_s} : I^{\mu+\lambda} \to I^\mu$	Selektionsoperator
$\iota : I^\mu \times \mathbb{N} \to \mathbb{B}$	Abbruchbedingung

Abbildung 2.3: Zusammenfassung der grundlegenden Notationen

Die Reproduktion zerfällt hier in die *Rekombination*

$$\begin{aligned} r_{\Theta_r} : I^\mu &\longrightarrow I^\lambda \\ (\vec{a}_1, \ldots, \vec{a}_\mu) &\longmapsto (\vec{a}'_1, \ldots, \vec{a}'_\lambda) \text{ mit } \vec{a}'_i = r'_{\Theta_r}(\vec{a}_1, \ldots, \vec{a}_\mu) \end{aligned}$$

und die *Mutation*

$$\begin{aligned} m_{\Theta_m} : I^\lambda &\longrightarrow I^\lambda \\ (\vec{a}'_1, \ldots, \vec{a}'_\lambda) &\longmapsto (\vec{a}''_1, \ldots, \vec{a}''_\lambda) \text{ mit } \vec{a}''_i = m'_{\Theta_m}(\vec{a}'_1, \ldots, \vec{a}'_\lambda) \end{aligned}$$

(Θ_r ist die Menge der Parameter der Rekombination, Θ_m für die Mutation). Bei der Rekombination r_{Θ_r} werden in λ Durchläufen durch die lokale Rekombination $r'_{\Theta_r} : I^\mu \to I$ mehrere Individuen (die Eltern) aus der Population ausgewählt und miteinander vermischt, wodurch ein neues Individuum $\vec{a}'_i, 1 \leq i \leq \lambda$ (der Nachkomme) entsteht. Dieser Nachkomme wird der Nachkommenpopulation P', die anfangs leer ist, hinzugefügt. Die Nachkommenpopulation $P' = (\vec{a}'_1, \ldots, \vec{a}'_\lambda)$ der Größe λ wird anschließend durch den Mutationsoperator m_{Θ_m} mutiert, indem jeder Nachkomme einer zufälligen Veränderung (Mutation) durch den Mutationsoperator $m'_{\Theta_m} : I \to I$ unterworfen wird. Das Ergebnis ist eine Population $P'' = (\vec{a}''_1, \ldots, \vec{a}''_\lambda)$ von Nachkommen. Diese Population wird bewertet, und μ der besten Individuen aus der alten

Population P und der neuen Population P'' werden durch den Selektionsoperator

$$\begin{aligned} s_{\Theta_s} : I^{\mu+\lambda} &\longrightarrow I^{\mu} \\ (\vec{a}_1(t), \ldots, \vec{a}_\mu(t), \vec{a}''_1(t), \ldots, \vec{a}''_\lambda(t)) &\longmapsto (\vec{a}_1(t+1), \ldots, \vec{a}_\mu(t+1)) \end{aligned}$$

ausgewählt (*selektiert*).

Bei der Selektion gibt es grundsätzlich verschiedene Möglichkeiten. Zum einen kann bei der Selektion nur die Nachkommenpopulation berücksichtigt werden, die Elternpopulation geht vollständig verloren. Zum anderen kann die Elternpopulation bei der Selektion berücksichtigt werden, wodurch die beste bisher gefundene Lösung bei entsprechender Wahl des Selektionsoperators erhalten bleiben kann.

Die selektierte Population ersetzt die alte Population. Der Kreislauf der Fortpflanzung und des Generationswechsels beginnt von neuem, bis die Abbruchbedingung $\iota : I^\mu \times \mathbb{N} \to \mathbb{B}$, die sowohl von der Population $P(t)$ als auch vom Generationszähler t abhängt, erfüllt ist. Abbildung 2.4 faßt dieses Vorgehen noch einmal in formalisierter Form zusammen.

$t := 0$
Initialisierung : $P(0) := (\vec{a}_1(0), \ldots, \vec{a}_\mu(0)) \in I^\mu$;
Bewertung von $P(0)$: $(\Phi(\vec{a}_1(0)), \ldots, \Phi(\vec{a}_\mu(0)))$;
while not $\iota(P(t), t)$ **loop**
    Rekombination : $P'(t) := r_{\Theta_r}(P(t))$;
    Mutation : $P''(t) := m_{\Theta_m}(P'(t))$;
    Bewertung von $P''(t)$: $(\Phi(\vec{a}''_1(t)), \ldots, \Phi(\vec{a}''_\lambda(t)))$;
    Selektion : $P(t+1) := s_{\Theta_s}(P''(t) \cup P(t))$;
    $t := t + 1$;
end loop;

Abbildung 2.4: allgemeines Ablaufschema eines EA

2.4 Evolutionsstrategien

2.4.1 Einführung

Evolutionsstrategien wurden entwickelt, um schwierige reellwertige Parameteroptimierungsprobleme lösen zu können [Rec73] [Sch81]. Die „natürliche“ Repräsentation ist dabei ein Vektor von reellwertigen „Genen“, die vornehmlich durch Mutation auf möglichst sinnvolle Weise manipuliert werden. Dies wird durch zusätzliche Strategieparameter sichergestellt, die die Mutation steuern und ebenfalls der Evolution

unterworfen sind. Auf diese Weise werden nicht nur Lösungen optimiert, sondern auch der Prozeß der Lösungsfindung. Durch die Evolution der Strategieparameter findet eine einfache Modellbildung der Fitneßlandschaft statt.

2.4.2 Algorithmus

2.4.2.1 Repräsentation

Lösungen bei der ES sind n–dimensionale Vektoren $\vec{x} \in I\!R^n$. Die Fitneß eines Individuums entspricht dem Wert der zu optimierenden Funktion, d.h. $\Phi(\vec{a}) = f(\vec{x})$, wobei $\vec{x}$ Bestandteil eines Individuums $\vec{a}$ ist und die Problemparameter x_i enthält. Zusätzlich zu $\vec{x}$ enthält $\vec{a}$ die Strategieparameter. Dies sind die n Standardabweichungen σ_i für die Mutation der einzelnen x_i.

Darüberhinaus können noch $n(n-1)/2$ weitere Strategieparameter α_{ij} hinzugefügt werden ($1 \leq i < j \leq n$), die die Rotationwinkel für eine n–dimensionale Normalverteilung darstellen.

Damit ergibt sich ein Individuum $\vec{a} \in I = I\!R^n \times I\!R^n \times I\!R^{n(n-1)/2}$ wie folgt:

$$\vec{a} = (x_1, \ldots, x_n, \sigma_1, \ldots, \sigma_n, \alpha_{12}, \ldots, \alpha_{n-1\,n}),$$

oder in der vereinfachten Fassung, die im folgenden verwendet wird:

$$\vec{a} = (x_1, \ldots, x_n, \sigma_1, \ldots, \sigma_n),$$

mit $I = I\!R^n \times I\!R^n$ gilt.

2.4.2.2 Rekombination

Rekombination findet sowohl im Bereich der Problemparameter als auch im Bereich der Strategieparameter statt. Dabei können für jeden der beiden Bereiche verschiedene Rekombinationsverfahren

$$\begin{array}{rcl} \hat{r}' : I\!R^n \times I\!R^n & \longrightarrow & I\!R^n \\ ((w_{a,1}, \ldots, w_{a,n}), (w_{b,1}, \ldots, w_{b,n})) & \longmapsto & (w'_1, \ldots, w'_n) \end{array}$$

in der lokalen Rekombination $r' : I^\mu \to I$ gewählt werden, die in die Auswahl zweier Eltern $\vec{a}$ und $\vec{b}$, die Rekombination der Problemparameter $\hat{r}'_x$ und die Rekombination der Strategieparameter $\hat{r}'_\sigma$ zerfällt.

Die drei verschiedenen Möglichkeiten für $\hat{r}'$ sind:

1. ohne Rekombination: die Werte w_i der Parameter gehen unverändert von einem Elternindividuum $\vec{a}$ auf ein Kind $\vec{a}'$ über, d.h.

   $$w'_i = w_{a,i}.$$

2. diskrete Rekombination: jeder Parameter des Kindes $\vec{a}'$ erhält (mit gleicher Wahrscheinlichkeit, aber zufällig) entweder den entsprechenden Wert des Elternteils $\vec{a}$ oder des Elternteils $\vec{b}$, d.h.

   $$w'_i = \begin{cases} w_{a,i} & \text{falls} \quad 0 \leq \chi \leq 0.5 \\ w_{b,i} & \text{falls} \quad 0.5 < \chi \leq 1 \end{cases},$$

3. mittelnde Rekombination: die Werte w'_i der Parameter des Kindes $\vec{a}'$ bestimmen sich als Mischung der Werte der Eltern $\vec{a}$ und $\vec{b}$: dem Wert eines Elternteils wird die mit einer Zufallszahl multiplizierten Differenz der Werte der Eltern addiert, d.h.

   $$w'_i = w_{a,i} + \chi \cdot (w_{b,i} - w_{a,i}).$$

   Dadurch liegt der Wert des Kindes zufällig zwischen den Werten der Eltern.

Dabei ist $w'_i \in I\!R$ der i–te Wert des Kindes, $w_{j,i} \in I\!R$ der i–te Wert des Elternteils j und χ eine gleichverteilte Variable aus $[0, 1]$. Die Elternindividuen $\vec{a}$ und $\vec{b}$ werden bei der Rekombination zufällig aus der Elternpopulation ausgewählt, ihre Fitneß spielt dabei keine Rolle. Durch die zufällige Auswahl der Eltern und die Verwendung von Zufallszahlen im zweiten und dritten Fall ist der Rekombinationsoperator r' nicht deterministisch.

Für die Rekombination von Problemparametern und Strategieparametern werden häufig verschiedene Rekombinationsverfahren gewählt.

2.4.2.3 Mutation

Bei der lokalen Mutation

$$\begin{array}{rcl} m'_{\{\tau,\tau'\}} : I & \longrightarrow & I \\ (x_1, \ldots, x_n, \sigma_1, \ldots, \sigma_n) & \longmapsto & (x'_1, \ldots, x'_n, \sigma'_1, \ldots, \sigma'_n) \end{array}$$

werden zuerst die Strategieparameter σ_i zufällig verändert, bevor die Problemparameter x_i unter Verwendung der neuen Strategieparameter

$$\sigma'_i = \sigma_i \cdot e^{(\tau' \cdot N(0,1) + \tau \cdot N(0,\sigma_i^2))}$$

mutiert werden. Die Parameter τ und τ' sind dabei vorgegebene Steuerparameter, die in den meisten Fällen zu 1 gewählt werden. Die Mutation der x_i verwendet eine Normalverteilung mit Erwartungswert 0 und Varianz $(\sigma_i')^2$, so daß sich

$$x_i' = x_i + N(0, (\sigma_i')^2)$$

ergibt. Durch die Verwendung von Zufallszahlen ist der Mutationsoperator nicht determinisch.

2.4.2.4 Selektion

Die Selektion wird vollständig deterministisch durchgeführt. Dabei gibt es zwei Varianten. Bei der (μ, λ)–Selektion $s_{(\mu,\lambda)} : I^\lambda \to I^\mu$ werden die μ besten Individuen aus den λ Individuen der Nachkommenpopulation ausgewählt (dazu muß natürlich gelten $\lambda > \mu$). Bei der $(\mu + \lambda)$–Selektion $s_{(\mu+\lambda)} : I^{\mu+\lambda} \to I^\mu$ werden die μ besten Individuen aus der vereinigten Eltern– und Nachkommenpopulation ausgewählt. Hier gehen die bisher besten Lösungen also nicht verloren, was aber das Suchverhalten in vielen Fällen verschlechtert [BS93].

Man unterscheidet anhand der Selektionsstrategie zwischen (μ, λ)– und $(\mu + \lambda)$–ES.

2.4.2.5 Ablaufschema

Der Algorithmus in Abbildung 2.5 vereinigt die vorgestellten Operatoren im Ablaufschema einer ES. Die Bewertung der Anfangspopulation zu Beginn ist eigentlich überflüssig, wurde aber wegen des allgemeinen EA–Schemas beibehalten.

2.4.3 Beispiel

Um ES und GA besser vergleichen zu können, wird hier ein Problem vorgestellt, für das dann eine ES und ein GA als Beispiel für die jeweiligen Verfahren angegeben wird.

2.4.3.1 Problemstellung

Das zu lösende Problem sei die Mimimierung der Funktion

$$F(x_1, x_2) = (x_1)^2 + (x_2)^2.$$

$t := 0$
Initialisierung : $P(0) := (\vec{a}_1(0), \ldots, \vec{a}_\mu(0)) \in I^\mu$
    where $I = \mathbb{R}^{2n}$ **and** $\vec{a}_k = (x_{k1}, \ldots, x_{kn}, \sigma_{k1}, \ldots, \sigma_{kn})$
Bewertung von $P(0)$: $(\Phi(\vec{a}_1(0)), \ldots, \Phi(\vec{a}_\mu(0)))$
    where $\Phi(\vec{a}_k(0)) = f(\vec{x}_k(0))$;
while not $\iota(P(t), t)$ **loop**
    Rekombination : $\vec{a}'_k(t) := r'(P(t)) \quad \forall k \in \{1, \ldots, \lambda\}$;
    Mutation : $\vec{a}''_k(t) := m'_{\{\tau,\tau'\}}(\vec{a}'_k(t)) \quad \forall k \in \{1, \ldots, \lambda\}$;
    Bewertung von $P''(t) := (\vec{a}''_1(t), \ldots, \vec{a}''_\lambda(t))$: $(\Phi(\vec{a}''_1(t)), \ldots, \Phi(\vec{a}''_\lambda(t)))$
        where $\Phi(\vec{a}''_k(t)) = f(\vec{x}''_k(t))$;
    Selektion : $P(t+1) :=$ **if** (μ, λ)–*Selektion*
                then $s_{(\mu,\lambda)}(P''(t))$
                else $s_{(\mu+\lambda)}(P''(t) \cup P(t))$;
    $t := t + 1$;
end loop;

Abbildung 2.5: allgemeines Ablaufschema einer ES

Diese Funktion, deren globales Minimum $(0, 0)$ bekannt ist, stellt kein typisches Problem für Evolutionäre Algorithmen dar, da es sich mit einfachen mathematischen Mitteln lösen läßt (z.B. einem indirekten kalkülbasierten Verfahren).

Dieses Minimierungsproblem soll lediglich der Veranschaulichung dienen. Bei typischen Problemen für Evolutionäre Algorithmen ist der Suchraum nicht so einfach strukturiert wie bei dieser stetigen Funktion. Außerdem ist in den meisten Fällen weder das Optimum noch die optimale Fitneß bekannt.

2.4.3.2 Umsetzung in eine (3,6)–ES

Für die Repräsentation, die zu verwendenden Operatoren und die Parameter der ES werden folgende Festlegungen getroffen:

- Die Argumente x_1 und x_2 der Funktion F werden direkt als reelle Problemparameter verwendet, wobei der Wertebereich auf $[-20, 20]$ eingeschränkt ist. Hinzu kommen noch die Strategieparameter σ_1 und σ_2, deren Wertebereich [0,2] ist. Die Wahl der Wertebereiche erfolgte willkürlich, dabei floß Wissen über das Minimum ein. Mit diesen Festlegungen steht der Aufbau eines Individuums fest, es gilt $I = [-20, 20] \times [-20, 20] \times [0, 2] \times [0, 2]$.
- Populationsgröße $\mu = 3$, Größe der Nachkommenpopulation $\lambda = 6$. So erhält man eine (3,6)–ES. Die Wahl der Parameter bei einer (μ, λ)–ES ist hier nicht

optimal, da die „1/5–Erfolgsregel“ [Rec73] bei diesen ES ein Verhältnis von μ zu λ von 1 zu 5 empfiehlt, so daß im Beispiel eigentlich λ zu 15 gewählt werden sollte. Um das Beispiel übersichtlich zu halten, wurde darauf jedoch verzichtet.

- Rekombination: Problemparameter x_i durch diskrete Rekombination, Strategieparamter σ_i durch mittelnde Rekombination.
- Parameter der Mutation: $\tau = 1$ und $\tau' = 1$.
- Selektion: nach der (μ, λ)–Strategie.
- Abbruchkriterium ι: es wurden 100 Generationen erzeugt, d.h. $t = 100$.

Zu Beginn wird eine Population mit 3 zufällig erzeugten Individuen gefüllt. Diese werden durch die Funktion $\Phi((x_1, x_2, \sigma_1, \sigma_2)) = F(x_1, x_2)$ bewertet. Durch die Rekombination dieser 3 Individuen werden 6 Kinder erzeugt, die anschließend mutiert werden. Die Kinder werden bewertet und danach die 3 besten ausgewählt.

i	$\vec{a}_i(0)$	$\Phi(\vec{a}_i(0))$	$\vec{a}'_i(0)$	$\vec{a}''_i(0)$	$\Phi(\vec{a}''_i(0))$	$\vec{a}_i(1)$
1	(1.0,2.1,0.5,1.2)	5.41	(1.0,3.4,0.4,1.3)	(1.5,4.4,0.4,1.1)	19.36	(1.5,4.4,0.4,1.1)
2	(12.3,-1.4,1.3,0.7)	153.25	(12.3,2.1,1.0,1.1)	(11.7,2.2,1.0,1.3)	143.98	(-5.4,-1.0,0.3,0.7)
3	(-5.5,3.4,0.1,1.7)	41.81	(1.0,2.1,0.5,1.2)	(1.1,1.9,0.7,1.4)	4.82	(1.1,1.9,0.7,1.4)
4	—	—	(-5.5,-1.4,0.2,0.8)	(-5.4,-1.0,0.3,0.7)	30.16	—
5	—	—	(-5.5,3.4,0.9,1.6)	(-5.7,3.9,0.8,1.6)	47.70	—
6	—	—	(-5.5,3.4,0.1,1.7)	(-5.0,3.5,0.2,1.5)	37.25	—

Abbildung 2.6: Anwendung der ES auf die Anfangspopulation

Abbildung 2.6 zeigt eine zufällige Anfangspopulation mit ihrer Bewertung, die Nachkommenpopulation nach der Rekombination und nach der Mutation sowie die Bewertung der Kinder. Das Individuum $\vec{a}'_1$ zum Beispiel entsteht durch Rekombination der Individuen $\vec{a}_1$ und $\vec{a}_3$. Durch Mutation von $\vec{a}'_1$ entstand das Individuum $\vec{a}''_1$, dessen Bewertung in der Nachkommenpopulation $P''(0)$ die zweitbeste ist, so daß es bei der Selektion in die nächste Generation, die Population $P(1)$ übernommen wird.

Nach der Selektion befinden sich die Individuen $\vec{a}''_1$, $\vec{a}''_3$ und $\vec{a}''_4$ in der Population $P(1)$, die wiederum dem gezeigten Verfahren unterworfen wird.

2.5 Genetische Algorithmen

2.5.1 Einführung

Genetische Algorithmen verdanken ihren Namen einem ihrer wesentlichen Aspekte: sie arbeiten mit einer Kodierung nach Art eines *Genotyps*. Der Genotyp ist in der

Biologie die Gesamtheit aller Gene. Er beschreibt den Aufbau eines Individuums. Das Individuum, den *Phänotyp*, erhält man durch Dekodieren der im Genotyp enthaltenen Information. Der Genotyp setzt sich aus einfachen Bausteinen zusammen, dafür kann diese Art der Kodierung von Individuen universell eingesetzt werden. So beruht der Genotyp aller Lebewesen auf unserem Planeten (bis auf sehr wenige Ausnahmen) auf denselben Bausteinen: den Codons, Tripletts über der Menge der Basen Adenin, Thymin, Guanin und Cytosin.

Die Kodierung geschieht bei GA durch Binärstrings fester Länge, die universellen Bausteine sind hier die Bits. Auf diesen Strings werden Operatoren wie Mutation und Crossover (s.u.) ausgeführt. Diese Operatoren besitzen auf der Kodierungsebene kein Wissen über die kodierte Problemstruktur. Auf diese Weise soll eine weitreichende Problemunabhängigkeit der GA erreicht werden, so daß sie für alle möglichen Probleme einsetzbar sind. Die Mutation, der wichtigste Operator bei der ES, spielt hier nur eine untergeordnete Rolle; im Vordergrund steht der je nach Problemstruktur zu wählende Crossover–Operator.

Im folgenden wird eine möglichst allgemeine Form des GA vorgestellt. Da es sehr viele verschiedene Varianten von GA und der verwendeten Operatoren gibt, können diese jedoch nur schwer durch ein einziges Modell repräsentiert werden, so daß das folgende Modell eine unvollkommene Näherung darstellt.

2.5.2 Algorithmus

2.5.2.1 Repräsentation

Die n Problemparameter werden bei einem GA binär kodiert. Jedem Problemparameter $x_i \in I\!R, 1 \leq i \leq n$, wird ein Binärstring der Länge $l_i \in N$ zugeordnet. Der Individuenraum I ergibt sich dann zu $I = I\!B^l$ mit $l = \sum_{i=1}^{n} l_i$

Da sich mit l_i Bits maximal 2^{l_i} verschiedene Werte kodieren lassen, ist die Suche nach dem Optimum im Gegensatz zur ES diskret. Für jeden Problemparameter $x_i \in I\!R$ muß daher eine Obergrenze o_i und eine Untergrenze u_i des Wertebereichs gewählt werden, die Schrittweite bei äquidistanter Kodierung ergibt sich dann mit der Anzahl l_i der Bits zu $(o_i - u_i)/(2^{l_i} - 1)$. Die Segmentdekodierungsfunktion $\Gamma_i : I\!B^{l_i} \rightarrow I\!R$, die den Wert eines Bitstrings der Länge l_i in den Wert von x_i umwandelt, sieht wie folgt aus:

$$\Gamma_i(a_{i1}, \ldots, a_{il_i}) = u_i + \frac{o_i - u_i}{2^{l_i} - 1} \sum_{j=1}^{l_i} a_{ij} 2^{j-1} ,$$

wobei $(a_{i1}, \ldots, a_{il_i})$ das i-te Segment eines Individuums

$$\vec{a} = (a_1, \ldots, a_l) = (a_{11}, \ldots, a_{1l_1}, \ldots, a_{n1}, \ldots, a_{nl_n})$$

ist. Kombiniert man die einzelnen Segmentdekodierungsfunktionen zur Dekodierungsfunktion $\Gamma : \mathbb{B}^l \to \mathbb{R}$ mit $\Gamma = \Gamma_1 \times \cdots \times \Gamma_n$, erhält man die Fitneß eines Individuums $\vec{a}$ wie folgt:

$$\Phi(\vec{a}) = \delta(f(\Gamma(\vec{a}))).$$

Dabei ist $\delta : \mathbb{R} \to \mathbb{R}$ eine Skalierungsfunktion, die dafür sorgt, daß die Fitneßwerte positiv sind und das beste Individuum die größte Fitneß erhält.

2.5.2.2 Rekombination

Im Vordergrund steht bei GA die Rekombination, die hier durch den *Crossover–Operator* durchgeführt wird. Dieser soll brauchbare Segmente verschiedener Individuen zusammenfügen und damit bessere Individuen schaffen. Auch der Crossover–Operator $r'_{\{p_c\}} : I^\mu \to I$ arbeitet auf der Ebene des Bitstrings, besitzt also kein Wissen über dessen Segmentaufteilung und die damit assoziierten Problemparameter.

Der exogene (von außen vorgegebene) Parameter p_c, die Crossover–Wahrscheinlichkeit, gibt die Wahrscheinlichkeit an, mit der ein Individuum für die Rekombination ausgewählt wird. Typische Werte für p_c liegen im Bereich $[0.6, 1]$.

Der einfachste Crossover–Operator $r'_{\{p_c\}}$ ist der *Einpunkt–Crossover* (*one–point crossover*). Zwei Elternindividuen $\vec{a} = (a_1, \ldots, a_l)$ und $\vec{b} = (b_1, \ldots, b_l)$ werden zufällig aus der Population ausgewählt. Zuerst wird anhand der Crossover–Wahrscheinlichkeit bestimmt, ob ein Crossover stattfindet. Dies ist dann der Fall, wenn eine gleichverteilte Zufallsvariable $v \in [0, 1]$ kleiner oder gleich der Crossover–Wahrscheinlichkeit p_c ist. Dann wird eine Crossover–Stelle χ bestimmt ($\chi \in \{1, \ldots, l-1\}$ ist eine gleichverteilte Zufallsvariable). Die beiden Nachkommen $\vec{a}'$ und $\vec{b}'$ ergeben sich dann folgendermaßen:

$$\begin{aligned} \vec{a}' &= (a_1, \ldots, a_\chi, b_{\chi+1}, \ldots, b_l) \\ \vec{b}' &= (b_1, \ldots, b_\chi, a_{\chi+1}, \ldots, a_l)\,. \end{aligned}$$

Von den beiden Nachkommen $\vec{a}'$ und $\vec{b}'$ wird dann einer zufällig ausgewählt. Normalerweise werden beide Nachkommen in die Nachkommenpopulation übernommen. Hier ist die Auswahl eines Nachkommen jedoch notwendig, um im gewählten Modell für Evolutionäre Algorithmen zu bleiben, bei dem die lokale Rekombination r' nur ein Individuum erzeugt.

Ist v größer als p_c, wird ein zufällig ausgewählter Elternteil unverändert in die Nachfolgerpopulation übernommen. Auf diese Weise können auch garantiert unveränderte Individuen in die Nachfolgerpopulation gelangen.

Der Einpunkt–Crossover kann zu einem *m–Punkt–Crossover* (*m–point crossover* [DJ75]) verallgemeinert werden ($1 \leq m \leq l-1$). Dort werden statt einer m Crossover–Stellen ermittelt.

m–Punkt–Crossover mit $m = l - 1$ wird auch *uniform–Crossover* [Sys89] genannt. Dieser Operator entscheidet für jedes Bit des Kindes $\vec{a}'$, von welchem Elternteil der Wert übernommen wird, das Kind $\vec{b}'$ erhält dann den Wert des anderen Elternteils. Das *uniform crossover* ist damit der diskreten Rekombination in der ES vergelichbar, findet jedoch auf der Ebene der Kodierung statt.

Dadurch, daß zufällig entschieden wird, ob ein Crossover stattfindet und die Crossoverstelle ebenfalls zufällig bestimmt wird, ist der Rekombinationsoperator nicht determinisch.

2.5.2.3 Mutation

Die Mutation

$$\begin{array}{rcl} m'_{\{p_m\}} : I & \longrightarrow & I \\ (a_1, \ldots, a_l & \longmapsto & (a'_1, \ldots, a'_l) \end{array}$$

arbeitet auf der Ebene des Bitstrings. Einzelne Bits eines Individuums $\vec{a} = (a_1, \ldots, a_l)$ werden mit der Mutationswahrscheinlichkeit p_m invertiert, die im allgemeinen recht klein gewählt wird ($p_m \approx 0.001$). Dazu wird zu jedem Bit $a_i, 1 \leq i \leq l$, des Individuums eine gleichverteilte Zufallszahl $\chi_i \in [0,1]$ ermittelt. Ist diese Zahl kleiner oder gleich der Mutationswahrscheinlichkeit, wird a_i invertiert; ansonsten bleibt a_i unverändert:

$$a'_i = \begin{cases} \text{NOT } a_i & \text{falls} \quad \chi_i \leq p_m \\ a_i & \text{falls} \quad p_m < \chi_i \end{cases} .$$

Dadurch, daß die Invertierung von Bits vom Zufall abhängt, ist der Mutationsoperator nicht deterministisch.

Eine Variante der Mutation besagt, daß ein Bit nicht invertiert wird, sondern einen zufälligen Wert erhält. Bei dieser Variante muß die Mutationswahrscheinlichkeit doppelt so groß gewählt werden, um dasselbe Verhalten zu erreichen. Aufgrund der zwei Varianten darf die Mutationswahrscheinlichkeit nicht unabhängig vom angewendeten Mutationsverfahren gesehen werden.

Die Mutation ist bei den GA ein „Hintergrundoperator“ (*background operator* [Hol75]). Sie dient dazu, eine vorzeitige Konvergenz des GA zu verhindern, indem sie eine gewisse Inhomogenität der Population aufrecht erhält.

2.5.2.4 Selektion

Bei der proportionalen Selektion

$$\begin{aligned} s : I^{\lambda} &\longrightarrow I^{\mu} \\ (\vec{a_1}''(t), \ldots, \vec{a_\lambda}''(t)) &\longmapsto (\vec{a_1}(t+1), \ldots, \vec{a_\mu}(t+1)) \end{aligned}$$

wird μ mal ein Individuum aus der Nachkommenpopulation $P''(t)$ ausgewählt und in die nächste Population $P(t+1)$ aufgenommen. Die Wahrscheinlichkeit $p_s(\vec{a}_i'')$ für die Selektion eines Individuums $\vec{a}_i''$ ist seiner relativen Fitneß proportional, es gilt

$$p_s(\vec{a}_i'') = \Phi(\vec{a}_i'') / \sum_{j=1}^{\lambda} \Phi(\vec{a}_j'') \,,$$

d.h. $\vec{a}_i(t+1) = \vec{a}_j''(t)$ mit Wahrscheinlichkeit $p_s(\vec{a}_j'')$.

Das bedeutet, daß Individuen mit hoher relativer Fitneß in der nächsten Generation statistisch häufiger vorkommen werden als Individuen niedriger relativer Fitneß. Angenommen, eine Population setzt sich aus vier Individuen zusammen, von denen eines die Fitneß 3 und die anderen drei die Fitneß 1 haben. Dann wird das Individuum mit Fitneß 3 mit Wahrscheinlichkeit $\frac{1}{2}$ ausgewählt, während die übrigen mit Wahrscheinlichkeit $\frac{1}{6}$ ausgewählt werden.

Auch bei der Selektion gibt es Varianten. Die *Elitismus–Strategie* zum Beispiel sorgt dafür, daß die η $(1 \leq \eta \leq \mu)$ besten Individuen aus $P(t) \cup P''(t)$ auf jeden Fall in die nächste Generation übernommen werden.

Da auch bei der Selektion Individuen zufällig ausgewählt werden, ist der Selektionoperator im Gegensatz zu dem der ES auch nicht deterministisch.

2.5.2.5 Ablaufschema

Abbildung 2.7 zeigt den konzeptionellen Algorithmus für einen GA. Im Gegensatz zu dem von Holland formulierten GA entfällt hier die Selektion nach der Bewertung der Anfangspopulation $P(0)$, damit der GA in das anfangs eingeführte EA–Ablaufschema paßt. Dies ändert jedoch kaum etwas an den Resultaten des Algorithmus [BS93]. Die Eigenschaft, daß die Auswahl der Eltern für den Crossover proportional zu ihrer Fitneß erfolgt (weil ihr Anteil an der Population proportional zu ihrer Fitneß ist), ist nach dem ersten Schleifendurchlauf erfüllt. Dann nämlich wurde die proportionale Selektion durchgeführt.

$t := 0$
Initialisierung : $P(0) := (\vec{a}_1(0), \ldots, \vec{a}_\mu(0)) \in I^\mu$
where $I = \{0,1\}^l$;
Bewertung von $P(0)$: $(\Phi(\vec{a}_1(0)), \ldots, \Phi(\vec{a}_\mu(0)))$
where $\Phi(\vec{a}_k(0)) = \delta(f(\Gamma(\vec{a}_k(0))))$;
while not $\iota(P(t), t)$ **loop**
Rekombination : $\vec{a}'_k(t) := r'_{\{p_c\}}(P(t)) \quad \forall k \in \{1, \ldots, \lambda\}$;
Mutation : $\vec{a}''_k(t) := m'_{\{p_m\}}(\vec{a}'_k(t)) \quad \forall k \in \{1, \ldots, \lambda\}$;
Bewertung : $P''(t) := (\vec{a}''_1(t), \ldots, \vec{a}''_\lambda(t)) : (\Phi(\vec{a}''_1(t)), \ldots, \Phi(\vec{a}''_\lambda(t)))$
where $\Phi(\vec{a}''_k(t)) = \delta(f(\Gamma(\vec{a}''_k(t))))$;
Selektion : $P(t+1) := s(P''(t))$
where $p_s(\vec{a}''_k(t)) = \Phi(\vec{a}''_k(t)) / \sum_{j=1}^{\lambda} \Phi(\vec{a}''_j(t))$;
$t := t + 1$;
end loop;

Abbildung 2.7: allgemeines Ablaufschema eines GA

2.5.3 Beispiel

2.5.3.1 Problemstellung

Die Problemstellung aus dem ES–Abschnitt, die Mimimierung der Funktion

$$F(x_1, x_2) = (x_1)^2 + (x_2)^2,$$

wird hier wieder aufgegriffen.

2.5.3.2 Umsetzung in einen GA

Für die Repräsentation, die zu verwendenden Operatoren und die Parameter der ES werden folgende Festlegungen getroffen:

- Die Argumente x_1 und x_2 der Funktion F sollen binär kodiert werden. Der Wertebereich wird wegen dieser Art der Kodierung (willkürlich) auf $[-16, 15]$ eingeschränkt; damit lassen sich die Problemparameter bei einer Schrittweite von 1 als 5–Bit–Werte kodieren. Der Aufbau eines Individuums $\vec{a} \in I = \mathbb{B}^{10}$ ist demzufolge als Bitstring $(a_1, \ldots, a_{10})$ der Länge $l = 10$ und den Segmentlängen $l_1 = l_2 = 5$ festgelegt.
- Populationsgröße $\mu = 6$, Größe der Nachkommenpopulation $\lambda = 6$.

- Rekombination: Einpunkt–Crossover, Crossover-Rate $p_c = 0.6$.
- Mutation: invertierende Mutation, Mutationswahrscheinlichkeit $p_m = 0.001$.
- Selektion: proportional zur relativen Fitneß.
- Populationsgröße $\mu = 6$, Größe der Nachkommenpopulation $\lambda = 6$.
- Abbruchkriterium ι: es wurden 100 Generationen erzeugt, d.h. $t = 101$.

Zu Beginn wird eine Population mit 6 zufällig erzeugten Individuen gefüllt. Diese werden durch die Funktion

$$\begin{aligned} \Phi\left(\left(a_1,\ldots,a_{10}\right)\right) &= \delta\left(f\left(\Gamma\left(\left(a_1,\ldots,a_{10}\right)\right)\right)\right) \\ &= 513 - F\left(dec(a_1,\ldots,a_5)-16, dec(a_6,\ldots,a_{10})-16\right) \end{aligned}$$

bewertet.

Die Skalierungsfunktion

$$\begin{aligned} \delta : I\!R &\longrightarrow I\!R \\ x &\longmapsto 513 - x \end{aligned}$$

macht aus dem Minimierungsproblem ein Maximierungsproblem, wobei der Offset $513 = 16^2 + 16^2 + 1$ dazu dient, nur positive Werte zu erhalten. Da im allgemeinen der Wertebereich der Fitneßfunktion nicht bekannt ist, werden bei der Skalierung normalerweise die Fitneßwerte der Population mitberücksichtigt. So könnte zum Beispiel die niedrigste Fitneß in der Population von allen Fitneßwerten abgezogen werden, um eine Normalisierung der Fitneß auf Werte größer oder gleich 0 zu erhalten.

Die Segmentdekodierungsfunktionen

$$\begin{aligned} \Gamma_i : I\!B^5 &\longrightarrow I\!R \\ (a_{5i-4},\ldots,a_{5i}) &\longmapsto x_i = dec_5(a_{5i-4},\ldots,a_{5i}) - 16 \end{aligned}$$

als Bestandteil der Dekodierungsfunktion

$$\begin{aligned} \Gamma : I\!B^{10} &\longrightarrow I\!R^2 \\ (a_1,\ldots,a_{10}) &\longmapsto (\Gamma_1(\left(a_1,\ldots,5\right)), \Gamma_2(\left(a_6,\ldots,10\right))) \end{aligned}$$

sorgen für die Dekodierung des Binärstrings und liefern die Werte für x_i. Die Funktion

$$\begin{aligned} dec_k : B^k &\longrightarrow I\!R \\ (a_1,\ldots,a_k) &\longmapsto \sum_{j=1}^{k} a_j \cdot 2^{j-1} \end{aligned}$$

überführt dabei eine k-stellige Binärzahl in die äquivalente Dezimalzahl.

Durch die Rekombination dieser 6 Individuen werden 6 Kinder erzeugt, die anschließend mutiert werden. Die Kinder werden bewertet und danach durch proportionale Selektion 6 Individuen für die nächste Population ausgewählt. Abbildung 2.8 zeigt eine zufällige Anfangspopulation mit ihrer Bewertung, die Nachkommenpopulation nach der Rekombination und nach der Mutation, die Bewertung der Nachkommen und die Selektionswahrscheinlichkeit bei proportionaler Selektion. Abbildung 2.9 gibt eine Übersicht über die Kodierung der Parameterwerte aus $[-16, 15]$.

i	$\vec{a}_i(0)$	$\Phi(\vec{a}_i(0))$	$\vec{a}'_i(0)$	$\vec{a}''_i(0)$	(x''_{1_i}, x''_{2_i})	$\Phi(\vec{a}''_i(0))$	$p_s(\vec{a}''_i)$
1	(01110 11001)	428	(11101 00111)	(10101 00111)	(5,-9)	407	0.145
2	(11010 01001)	364	(01110 10101)	(01110 10110)	(-2,6)	460	0.163
3	(00101 10101)	367	(01101 01101)	(01101 01101)	(-3,-3)	495	0.176
4	(10001 01010)	476	(00101 10101)	(10101 10101)	(5,5)	463	0.164
5	(01101 00111)	423	(10001 01101)	(10001 01101)	(1,-3)	503	0.179
6	(01001 01101)	455	(10001 01010)	(10001 01011)	(1,-5)	487	0.173

Abbildung 2.8: Anwendung des GA auf die Anfangspopulation

$\Gamma_i(b)$	b	$\Gamma_i(b)$	b	$\Gamma_i(b)$	b	$\Gamma_i(b)$	b
-16	00000	-8	01000	0	10000	8	11000
-15	00001	-7	01001	1	10001	9	11001
-14	00010	-6	01010	2	10010	10	11010
-13	00011	-5	01011	3	10011	11	11011
-12	00100	-4	01100	4	10100	12	11100
-11	00101	-3	01101	5	10101	13	11101
-10	00110	-2	01110	6	10110	14	11110
-9	00111	-1	01111	7	10111	15	11111

Abbildung 2.9: Parameterwerte und ihre Kodierung

Das Individuum $\vec{a}'_1$ zum Beispiel entsteht durch Crossover der Individuen $\vec{a}_2$ und $\vec{a}_5$ nach der 2. Stelle, $\vec{a}'_2$ durch Crossover von $\vec{a}_1$ und $\vec{a}_3$ nach der 6. Stelle. Durch Mutation von $\vec{a}'_1$ entstand das Individuum $\vec{a}''_1$, dessen zweites Bit invertiert wurde.

Die mittlere Fitneß in diesem Beispiel hat sich von der Elternpopulation hin zur Nachkommenpopulation stark verbessert. Betrug sie bei der Anfangspopulation noch 419, hat die Nachkommenpopulation eine mittlere Fitneß von 469. Das Individuum $\vec{a}''_5$ liegt bereits recht nahe am Optimum. Er wird deshalb auch mit hoher Wahrscheinlichkeit in der nächsten Population zu finden sein; garantiert ist dies jedoch nicht, da die Selektion stochastisch arbeitet.

Die binäre Kodierung in diesem Beispiel hat wesentliche Nachteile, die das Finden der optimalen Lösung erschweren. Die $(0, 0)$ hat als Kodierung den Bitstring (10000 10000), während negative Zahlen grundsätzlich eine 0 an der ersten Bitposition haben. Der Bitstring (00000 00000) hingegen steht für $(-16, -16)$ und stellt die

schlechteste Lösung dar, dennoch unterscheidet er sich kaum von der besten Lösung. Durch die Wertigkeit der Bitstellen liegen benachbarte Zahlen in der Kodierung oft weit auseinander. So hat zum Beispiel -1 die Kodierung 01111, während 0 mit 10000 kodiert wird. Hingegen wird 15 mit 11111 kodiert und ist damit der -1 recht ähnlich.

Ein Ansatz für die Lösung des Wertigkeitsproblems ist der *Gray–Code* [HB94], bei dem sich benachbarte Zahlen durch ein einziges Bit unterscheiden. Auf diese Weise erzeugt die Mutation keine großen Sprünge im Lösungsraum, sondern, wie bei der ES, kleine Suchschritte. Für die großen Sprünge im Lösungsraum sorgt dann nur noch das Crossover.

2.6 Gemeinsamkeiten und Unterschiede von ES und GA

Beiden Verfahren gemeinsam ist ihre universelle Anwendbarkeit auf Optimierungsprobleme, soweit sich diese als Parameteroptimierungsprobleme fassen lassen und eine geeignete Kodierung für die Verfahren gefunden werden kann. Außerdem fußen beide auf einem vereinfachten Modell der Evolution nach der Darwinschen Evolutionstheorie. Daher arbeiten beide mit einer Population von Individuen und selektieren im Laufe des Verfahrens bevorzugt die besten Individuen.

Die wesentlichen Unterschiede bestehen zum einen in der Art der Kodierung. Erfolgt diese bei der ES durch reelle Zahlen, also kontinuierlich und problemnah (phänotypisch), so geschieht dies bei den GA durch binäre Zahlen, die diskret sind, die Kodierung ist abstrakt (genotypisch) und muß zur Bewertung erst umgewandelt werden; dafür ist diese Kodierung universell einsetzbar. Der andere wesentliche Unterschied liegt in der Art des Vorgehens. Während bei der ES die Mutation mit Selbstanpassung der Mutationsparameter im Vordergrund steht, ist der wichtigste Operator bei den GA der Crossover, der gute Lösungen zu noch besseren vereinigen soll, indem er die guten Eigenschaften dieser Lösungen vereinigt. Die Wahl des Crossover–Operators erfolgt problemabhängig, wodurch Problemwissen in das Verfahren integriert werden kann. Durch die Selbstadaption der Mutationsparameter gewinnt die ES nicht nur bessere Lösungen, sondern — in gewissem Maße — auch Wissen über die Struktur des Lösungsraums. Bei der Mutation im GA hingegen ist kein Problemwissen vorhanden, was bei ungeeigneter Kodierung zu großen Sprüngen im Lösungsraum führen kann. Auch bei der Selektion gibt es einen Unterschied: Während bei der ES die μ besten Individuen ausgewählt werden, ist dies beim GA nicht gewährleistet, da dort eine zufällige Selektion proportional zur relativen Fitneß erfolgt.

Abbildung 2.10 faßt die wichtigsten Unterschiede von ES und GA noch einmal zusammen. In [HB90] werden Gemeinsamkeiten und Unterschiede der beiden Verfahren ausführlich dargestellt und die Leistungsfähigkeit bei bestimmten Problemen empi-

Kriterium	Evolutionsstrategien	Genetische Algorithmen
Kodierung von Individuen	phänotypisch, reell	genotypisch, binär
Primärer Operator	Mutation	Crossover
Sekundärer Operator	Rekombination	Mutation
Selektionsstrategie	Selektion der Besten	proportionale Selektion
Strategieparameter	exogen/adaptiv	exogen

Abbildung 2.10: Übersicht der Unterschiede von ES und GA

risch mit Hilfe bekannter Testfunktionen untersucht. Dieser Artikel wird in [SHF94, S. 232–236] kritisch diskutiert.

Kapitel 3

Einführendes zu EAGLE

Die Kapitel 3 bis 8 sollen den Entwicklungsprozeß und den Entwicklungsstand des Software–Projekts EAGLE (Evolutionary Algorithms Gaming and Learning Environment) der Projektgruppe Genetische Algorithmen (PGA) dokumentieren.

Diese Dokumentation ist das Ergebnis unserer Arbeit an EAGLE und ist daher die Grundlage für weiterführende Realisierungspläne dieses Softwareprojekts. Sie kann aber auch Anregungen liefern für andere Entwicklungsgruppen, die sich ebenfalls mit Evolutionären Algorithmen (EA) beschäftigen.

Leider ist dieser Teil der Gesamtdokumentation in der vorliegenden Form nicht entwicklungsbegleitend entstanden. Viele Dokumente des Software-Entwicklungsprozeß lagen bei Abschluß der Projektarbeit nur fragmentarisch vor und wurden deshalb noch einmal aus heutiger Sicht überarbeitet. Dadurch verschwanden besonders in den Kapiteln 4 und 5 viele Anforderungen, die sich als nicht realisierbar im Rahmen der PGA erwiesen haben. Näheres hierzu kann im Kapitel 9.1 nachgelesen werden.

In diesem Kapitel wird eine Einführung in die Ideen gegeben, die Grundlage für die Entwicklung von EAGLE sind. In Kapitel 4 wird ein Anforderungskatalog für EAGLE vorgestellt. Kapitel 5 enthält eine Spezifikation des Kernsystems von EALGE, die sich an den verschiedenen Fenstern des Systems festmacht. Ein Teil, der in Kapitel 5 ausgelassen wird, ist die Spezifikation der eingeführten Sprache LEA (Language for Evolutionary Algorithms). Diese wird in Kapitel 6 vorgestellt. Kapitel 7 ist dann eine Formalisierung des Gesamtsystems EAGLE (einschließlich LEA). Abschließend wird in Kapitel 8 EAGLE einer kritischen Untersuchung unterzogen.

3.1 Was ist EAGLE?

Aus der Beschäftigung mit dem Thema „Genetische Algorithmen (GA)“ in der Seminarphase der PGA erwuchs sehr schnell das Bedürfnis, ein eigenes System für

die Modellierung und Durchführung Evolutionärer Algorithmen (EA) zu schreiben. Der Begriff *Evolutionäre Algorithmen* geht dabei über die Genetischen Algorithmen deutlich hinaus. Wir verstehen darunter im wesentlichen Evolutionsstrategien (ES), Genetische Algorithmen (GA), Simulated Annealing (SA) und Threshold Algorithmen (TA). Außerdem wollen wir mit EAGLE auch weitere stochastische Verfahren modellieren können, die mit Populationen arbeiten. Der PGA wurde bald klar, daß es sehr viele verschiedene Konzepte gibt, die wir von solch einem System unterstützt haben wollen.

3.1.1 Wesentliche Konzepte von EAGLE

Eine wichtige Idee, die unserem System zugrundeliegt, ist der Werkzeugcharakter der einzelnen Teile, die wir in EAGLE zu einem System zusammenstellen. Diese Idee ist sicher nicht neu, nur ist sie in keinem uns bekannten System für EA in der hier vorgestellten Konsequenz verwirklicht worden. Die andere wichtige Idee, die uns bei der Entwicklung von EAGLE am Herzen lag, ist die Benutzerfreundlichkeit. Aus diesen beiden Grundsätzen ergeben sich die folgenden Konzepte:

Kombination verschiedener Datentypen zur Beschreibung eines Problems Das Problem soll so problemnah wie möglich in das System eingegeben werden können. Dazu erscheint es uns notwendig in der Problemstruktur, das ist die mathematische Darstellung der Struktur des Problemraums, verschiedene Datentypen zuzulassen, die beliebig kombiniert werden können.

Trennung zwischen Problemstruktur und Kodierungstruktur Ausgehend von einer solchen Beschreibung eines Problems ist es notwendig, die Definitionen des Optimierungsproblems und der anzuwendenden Verfahren konsequent zu trennen, um verschiedenen Verfahren auf ein Problem anwenden zu können. Dazu werden zwischen Problem und Verfahren Kodierungen geschaltet, die die Problemstruktur auf die gewünschten Verfahren zuschneidern.

Eine solche Kodierung bildet den Problemraum in einen anderen Raum ab, der auf der einen Seite auf das Verfahren angepaßt ist, so daß das Verfahren auf das Problem überhaupt angewendet werden kann. Z.B. soll ein Problem, dessen Struktur durch einen reellwertigen Vektor dargestellt wird, in einen Bitstring kodiert werden können, so daß ein Standard–GA darauf angewendet werden kann, oder die Struktur soll durch weitere Datenstrukturen erweitert werden können, so daß eine ES mit selbstanpassenden Strategieparametern möglich ist.

Auf der anderen Seite kann eine Kodierung unschöne Eigenschaften des Problemraumes, wie z.B. Unstetigkeiten, z.T. ausgleichen. Da die Eigenschaften des Problemraumes in der Regel nicht vollständig bekannt sind, sonst bräuchte man keinen EA, um

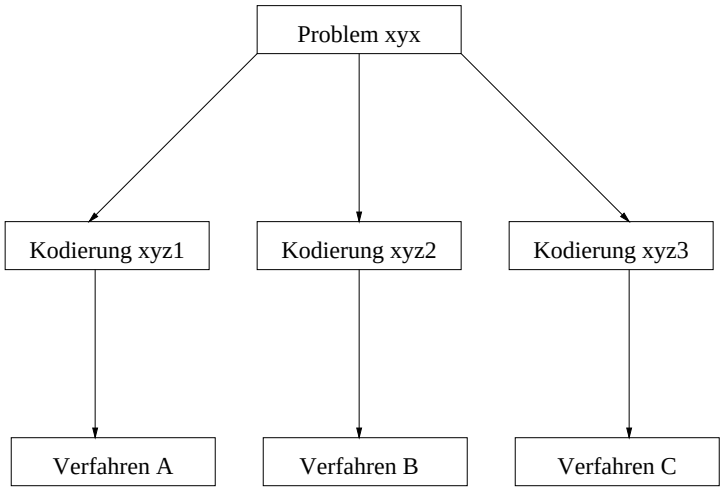


Abbildung 3.1: Verschiedene Kodierungen für verschiedene Verfahren

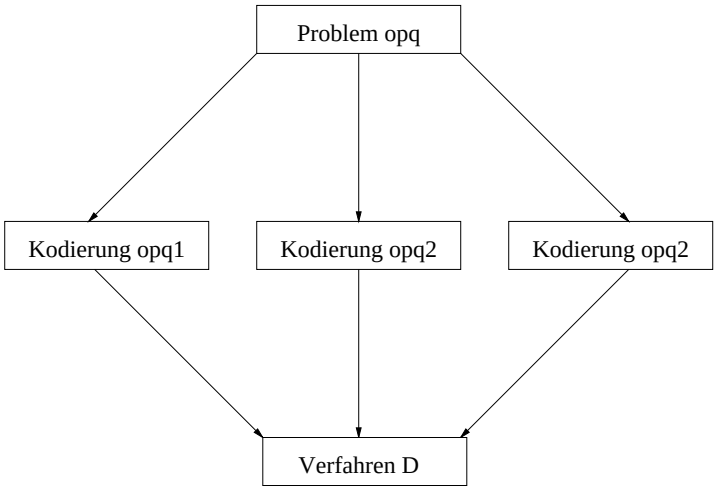


Abbildung 3.2: Verschiedene Kodierungen zur Anpassung an ein Verfahren

das Optimum in diesem Raum zu finden, ist die Wahl der Kodierung oft schwierig. Wir wollen es in unserem System EAGLE ermöglichen, einfach verschiedene Kodierungen für ein Problem auszuprobieren, um eine brauchbare herauszufinden.

Operatorenkonzept Eine andere Konsequenz aus der Idee des Werkzeugcharakters ist die Modularisierung der einzelnen Algorithmen der verschiedenen Verfahren. Die einzelnen Module eines Algorithmus werden als Operatoren betrachtet, aus denen sich der Benutzer neue Verfahren einfach zusammenstellen kann.

Es soll möglich sein, Operatoren für die Manipulation von ganzen Populationen, von einzelnen Individuen oder auch von Teilen von Individuen ineinander zu verschachteln, wobei ein rekursiver Aufruf ausgeschlossen sein soll. Ein neues Verfahren kann so aus beliebigen Kombinationen von bereits formulierten Verfahren bestehen, solange die Datenstrukturen (Kodierungen) verträglich sind. (Zwei Kodierungen können als verträglich gelten, wenn es eine Projektion von einer Kodierung auf die andere gibt.)

Leichte Entwicklung neuer Operatoren Der Pool von Operatoren, aus denen der Benutzer sich Verfahren zusammenstellen kann, soll möglichst einfach zu erweitern sein. Deshalb wird dem Benutzer die Möglichkeit bereitgestellt, eigene Operatoren zu entwerfen und mit dem System EAGLE auszuprobieren. Um eine beliebige Modularisierung von Operatoren zu zulassen und damit die Wiederverwertbarkeit der Operatoren möglichst offen zu halten, ist die Eingabe von Operatoren nicht auf spezielle EA–Operatoren, die auf Populationen bzw. auf Individuen arbeiten, beschränkt. Es können auch Operatoren, die auf anderen Datenstrukturen arbeiten, definiert werden.

Die so selbst entworfenen Operatoren werden vom System EAGLE direkt eingegliedert, so daß diese neuen Operatoren wie die im System bereits existierenden, festimplementierten Operatoren behandelt und verwendet werden.

Interaktion Ein wesentlicher Gesichtspunkt von EAGLE ist die Möglichkeit während eines Optimierungslaufes interaktiv eingreifen zu können, ohne damit den Lauf abzubrechen. Dazu ist es notwendig, während des Ablaufes eines Verfahrens die Entwicklung der Optimierung verfolgen zu können. Es ist möglich, einfach in den Ablauf eines Verfahrens einzugreifen, das Verfahren geringfügig (z.B. in der Wahl der Wahrscheinlichkeiten mit denen bestimmte Operatoren angewendet werden) zu verändern und mit diesen Veränderungen weiterlaufen zu lassen. Es ist auch möglich, ein bereits abgelaufenes Experiment exakt zu wiederholen oder auf den Daten eines alten Optimierungslaufes ein neues Experiment aufzusetzen.

3.1.2 Zusammenfassung

In dieser Form und Zusammenstellung eines Systems ist EAGLE mehr als nur das Zusammenschalten von bekannten Verfahren. Es soll ein System sein, daß es auf der einen Seite ermöglicht, Kodierung und Operatoren direkt auf ein konkretes Problem zu zuschneiden. Auf der anderen Seite soll es die Suche nach Ideen und deren Ausprobieren erleichern.

3.2 Warum eine Neuentwicklung ?

In diesem Abschnitt wird erläutert, warum wir uns zu einer Neuentwicklung entschlossen haben und kein anderes System als zu erweiternde Grundlage herangezogen haben.

Einen Überblick über bestehende nicht–kommerzielle Systeme aus den Bereichen *Evolutionsstrategien* und *Genetische Algorithmen* bietet Heitkötter[HB94] in seinem FAQ zur Newsgroup comp.ai.genetics (siehe auch Anhang B). Dort werden inzwischen 34 Systeme kurz beschrieben, die fast alle über ftp zu erhalten sind. Die meisten Systeme wenden sich dabei wohl vorwiegend an Einsteiger, da kein interaktiver Einbau eigener Operatoren vorgesehen ist. Bemerkenswert ist an dieser Auflistung vielleicht noch, daß auch andere Entwicklungsgruppen schon an einen modularen Aufbau von Genetischen oder Evolutionären Algorithmen gedacht haben. So stehen mit GAGS, GENlib und LibGA schon drei Libraries zum Zusammenbau Genetischer Algorithmen bereit. Allerdings haben wir uns mit der Analyse dieser Bibliotheken nicht weiter beschäftigt, da uns daran gelegen war, etwas grundlegend Neues zu entwickeln.

Außerdem läßt sich die Idee der Trennung von Problem und Kodierung nur dann konsequent umsetzen, wenn die Einzelindividuen gewissermaßen aus zwei Blickwinkeln betrachtet werden können. Einerseits aus Sicht der Kodierung, um die Verfahren auf sie anwenden zu können und andererseits aus Sicht des Problems, um die Fitneßberechnung durchführen zu können. Natürlich ließen sich diese Ideen auch mit einer geeigneten Schnittstelle an bestehende Systeme anpassen, allerdings schien uns der Aufwand hierfür zu hoch.

Heitkötter[HB94] nennt ein System, das einen ähnlichen Ansatz haben könnte. *Splicer* wird als System beschrieben, das wohldefinierte Schnittstellen zwischen GA kernel, Repräsentationsbibliotheken, Fitneßmodulen und Bibliotheken für Benutzungsschnittstellen besitzt. Leider können wir jedoch dieses System nicht als Lehrbeispiel heranziehen, da seine freie Verfügbarkeit auf die NASA und ihre Vertragspartner beschränkt ist. Andere Benutzer müssen zahlen.

Wir haben uns deshalb entschlossen, das Rad neu zu erfinden und ein völlig neues System zu schreiben, um unsere in Abschnitt 3.1.1 beschriebenen Konzepte ohne Kompromisse verwirklichen zu können.

3.3 Wer soll mit EAGLE arbeiten ?

EAGLE will im wesentlichen zwei Zielgruppen ansprechen:

- *Neueinsteiger* mit dem Interesse an genetischen oder anderen stochastischen Verfahren sollen die Möglichkeit haben, mit den Verfahren gewissermaßen herumzuspielen und so ein praktisches Gefühl für die Wirkungsweise und Effizienz solcher Verfahren zu entwickeln.
- *Forscher* mit einem Forschungsinteresse auf dem Gebiet der Evolutionären Algorithmen sollen die Möglichkeit haben, eine neue Idee schnell und einfach in einen EA umzusetzen, um sie in der Praxis zu bestätigen oder zu widerlegen. Das ersetzt natürlich keine formalen Methoden, kann aber helfen, vielversprechende Ansatzpunkte zu lokalisieren.

3.4 Ein Einsatzszenario

Das folgende Einsatzszenario von EAGLE soll dazu dienen, die geplanten Möglichkeiten plastischer vorzustellen.

Folgende Notation wurde verwendet:

→	Eingabe des Benutzers
←	Ausgabe des Systems
•	Erklärung

Die Benutzerin ist eine Neueinsteigerin. In der Vorlesung *Naturanaloge Verfahren* wurde das System EAGLE vorgestellt. Jeder Teilnehmer an der Vorlesung hat eine Zugangsberechtigung für EAGLE erhalten, um die Übungen bearbeiten und auch selbst mit den System herumexperimentieren zu können. Wir nennen die Benutzerin zur Vereinfachung B.. B. möchte zum Einstieg eine zweidimensionale reellwertige Funktion optimieren. B. setzt sich ans Terminal und startet das System EAGLE:

← Problem oder Verfahren eingeben ?

→ Problem.

• B. möchte zuerst das Problem eingeben, um sich dann ganz dem Verfahren widmen zu können.

← Bitte Problem eingeben.

→ Eingabe von zwei Parametern x und y, Funktion als Fitneßfunktion.

- Die Abspeicherung und Dateiverwaltung ist nicht Gegenstand dieses Szenarios.

← Bitte Verfahren eingeben.

→ Wahl der Standardkodierung (x und y als Bitstring, Reihenfolge wie bei der Eingabe, Standardgenauigkeit). Eingabe eines Verfahrens, das nur aus festimplementierten Operatoren zusammengesetzt ist (Standard-GA mit Standardcrossover, Standardmutation, Standardselektion und Standardabbruchbedingung). Start des Verfahrens.

- B. wählt Standardparameter, um sich mit dem System vertraut zu machen.

← Regelmäßig wird die Fitneß des besten Individuums, die Generation und der Erfüllungsgrad der Abbruchbedingung angezeigt. Am Ende wird das beste Paar (x,y) zusammen mit seiner Fitneß ausgegeben.

- B. stellt fest, daß das tatsächliche Optimum gefunden wurde. B. folgert, daß die Fitneßfunktion vom Standard-Crossover wohl begünstigt wird. B. überlegt sich daher gezielt einen anderen Crossover-Operator, der vielleicht ungünstiger wirkt.

→ Eingabe eines anderen Crossover-Operators, der anstelle des alten in das Verfahren eingesetzt wird. Die Kodierung bleibt zunächst erhalten. Start des Verfahrens.

← Regelmäßig wird die Fitneß des besten Individuums, die Generation und der Erfüllungsgrad der Abbruchbedingung angezeigt. Am Ende wird das beste Paar (x,y) zusammen mit seiner Fitneß ausgegeben.

- B. hat am Bildschirm verfolgt, daß zwar wieder das Optimum gefunden wurde, aber diesmal langsamer, wie auch an der größeren Zahl von Generationen abgelesen werden kann. Der gewählte Crossover-Operator war also tatsächlich ungünstiger. B. wählt nun für ein zweites Experiment ein abgewandeltes Verfahren. Statt Crossover und Mutation soll nur noch Mutation durchgeführt werden. Außerdem möchte B. statt der Standardkodierung einen gray-kodierten Bitstring haben.

→ Eingabe der Daten. Außerdem Wahl einer größeren Genauigkeit der Kodierung.

← Leider ist der eingebaute Mutationsoperator nicht in der Lage, Bitstrings mit mehr als 8 Bits zu bearbeiten.

- Durch die größere Genauigkeit wurde implizit auch die Länge des Individuums als Instanz der Kodierungsstruktur vergrößert. Daß der eingebaute Mutationsoperator nur Bitstrings mit höchstens 8 Bits bearbeiten kann, ist natürlich nicht gewollt, es soll hier nur gezeigt werden, wie eine sinnvolle Reaktion des Systems in einem solchen Fall aussehen könnte.

→ Eingabe eines besseren Mutationsoperators.

- Dieser selbstgeschriebene Mutationsoperator kann nun ohne Probleme in den Standard-GA eingebunden werden.

→ Einbindung des eigenen Mutationsoperators in den Standard-GA. Start des Verfahrens.

← Regelmäßig wird die Fitneß des besten Individuums, die Generation und der Erfüllungsgrad der Abbruchbedingung angezeigt. Am Ende wird das beste Paar (x,y) zusammen mit seiner Fitneß ausgegeben.

- B. stellt fest, daß das Optimum nun noch schneller erreicht wurde. Durch die vielen Änderungen ist jedoch nicht klar, ob die Verbesserung hauptsächlich durch die veränderte Kodierung oder durch das Weglassen des Crossover-Operators erreicht wurde. B. will diesen Zusammenhang genauer untersuchen.

Natürlich ist dieses Szenario unvollständig, vieles wurde nur gestreift, das meiste nicht einmal angeschnitten. Wenn dabei aber die wesentlichen Konzepte *Trennung zwischen Problemstruktur und Kodierungsstruktur*, *Operatorenkonzept* sowie *Leichte Entwicklung neuer Operatoren* wiederentdeckt werden konnten, hat es seinen Zweck erfüllt.

Kapitel 4

Pflichtenheft für EAGLE

4.1 Funktionale Sicht (Pflichtteil)

4.1.1 Überblick

Die Aufgabe des Systems EAGLE besteht darin, dem Benutzer die Eingabe und Verwaltung von EA und Problemen zu ermöglichen und ihn bei der Anpassung von diesen EA auf Probleme zu unterstützen. Im folgenden wird deshalb zunächst auf die Eingabe von Problemen und EA eingegangen. Anschließend werden die Interaktionsmöglichkeiten vor, nach und während dem Ablauf eines EAs beschrieben.

4.1.2 Eingabe von Problem

Probleme sollen eingegeben und geändert werden können. Folgende Teile des Problems sollen dabei frei editierbar sein:

- die Problemstruktur (das ist die mathematische Darstellung des Suchraums der möglichen Lösungen, unter denen eine optimale gefunden werden soll) soll innerhalb des Systems EAGLE interaktiv eingegeben werden. Dazu soll EAGLE die folgenden Grunddatentypen bereitstellen:
  - Boolean
  - Integer mit Wertebereich
  - Real mit Wertebereich
  - Permutation mit Größenangabe.

- die Fitneßfunktion (das ist die Funktion die jedem Element des Suchraums des Problems eine Bewertung (Fitneß) zuordnet, so daß anhand dieser Funktion beurteilt werden kann, ob ein Element des Suchraums besser ist als ein anderes) soll ebenfalls interaktiv in EAGLE eingegeben werden können.

4.1.3 Eingabe des EA

Der Evolutionäre Algorithmus soll editiert werden können. Dazu müssen folgende Teile editierbar sein:

- die Kodierung (das ist die Anpassung des Suchraums an ein spezielles Verfahren) soll in EAGLE so möglich sein, daß sich Problemwissen und Kenntnis des gewünschten Verfahrens völlig trennen lassen. Die Problemstruktur soll durch die Kodierung in eine Kodierungsstruktur abgebildet werden, auf der der EA arbeitet. Zur Bewertung einer Belegung der Kodierungsstruktur ist es notwendig, die Abbildung umzukehren, die die Problemstruktur auf die Kodierungsstruktur abgebildet hat. Die Kodierung stellt somit die jeweilige Anpassung zwischen Problem und Verfahren dar.
- die Verfahren (das sind die Ablaufalgorithmen der EA) sollen frei editierbar sein. EAGLE soll das Baukastenprinzip so verwirklichen, daß sich ein Verfahren aus frei editierbaren Operatoren zusammensetzen läßt, die wiederum aus Operatoren bestehen können. Es soll eine Reihe von festimplementierten Operatoren geben, sowie die Möglichkeit in einem Editor selbst Operatoren zu definieren. Fest implementiert sollen die folgenden Operatoren vorliegen:
  - auf Populationen:
    - Standard-EA
    - Standard-GA
    - Roulette-Wheel-Selection
    - Elitist-Selection
    - Whole-Population (Filter)
    - Best-Individual (Filter)
  - auf Individuen:
    - Hill-Climbing
    - Standard-Crossover
    - Standard-Mutate
    - Whole-Individual (Filter)
  - auf Atomen:
    - Standard-Mutate

Innerhalb eines Operators sollen die folgenden Elemente definiert werden können:

- freie Parameter, (z.B. die Wahrscheinlichkeit, mit der ein Operator angewendet wird) die leicht geändert werden können, ohne den jeweiligen Operator neu zu editieren.
- Filter, das sind Operationen, die keinerlei Veränderung bei den berechneten Daten hervorrufen, sondern lediglich vorhandene Daten in entsprechende Files schreiben.
- Label, das sind Markierungspunkte, an denen eine Berechnung eines Operators geordnet gestoppt kann. Dabei bedeutet geordnet, das eine Wiederaufnahme der Berechnung an diesem Punkt möglich ist.

Diese Elemente eines Operators sollen vor der eigentlichen Durchführung eines EA in der sogenannten Laufinitialisierung interaktiv angesprochen werden können.

4.1.4 Laufinitialisierung

Vor dem Start eines EA sollen in der Laufinitialisierung die folgenden Belegungen interaktiv eingegeben oder verändert werden können:

- die Parameterbelegung aller freien Parameter eines Verfahrens sollen gesetzt bzw. verändert werden können.
- die im Verfahren definierten Filter sollen vor dem Lauf des EA aktiviert bzw. deaktiviert werden können. Außerdem soll es wahlweise möglich sein, die Ausgabe in eine Datei oder auf die Standardausgabe zu lenken.
- die im Verfahren definierten Label sollen belegt werden können. Jeder Label kann aktiv/inaktiv sein oder zum Tracen verwendet werden. Im letzten Fall stoppt der EA bei jedem Passieren dieses Label.

Während der EA abläuft, soll es jederzeit möglich sein, zur Laufinitialisierung zurückzukehren, um die eingestellten Werte verändern zu können.

4.1.5 Abspeichern

Folgende Dinge sollen während des Laufes abgespeichert werden können:

- Die Start- bzw. aktuelle Population

- Wiederaufsetzpunkte.
  Der gesamte Zustand des Verfahrens wird dabei so abgespeichert, daß zu einem späteren Zeitpunkt das Wiederaufsetzen an gleicher Stelle möglich ist.

Diese Abspeicherungsmöglichkeiten sind statisch. Im Gegensatz dazu speichern Filter fortlaufend aktuelle Daten aus dem Verfahren heraus ab.

4.2 Funktionale Sicht (Erweiterung)

4.2.1 Überblick

Dieses Kapitel soll einige Erweiterungen enthalten, die schon einmal angedacht wurden. Diese Erweiterungen sind nicht Teil der Anforderung. Wo es jedoch möglich ist, die Voraussetzungen einer späteren Erweiterung ohne Mehraufwand zu schaffen, soll dies geschehen.

4.2.2 Eingabe von Problem (Erweiterung)

Im Problem sind folgende Erweiterungen möglich:

- Problemstruktur: weitere Grunddatentypen.
- Fitneßfunktion: Schnittstelle zum Aufruf externer Funktionen oder Datenbanken, um eine externe Fitneßberechnung zu ermöglichen.
- Constraint-Funktion: Dient zur Einschränkung des Problemraums mit Restriktionen, die über Bereichsbeschränkungen hinausgehen.
- erzeuge_korrektes_Individuum: Diese Prozedur ist insbesondere dann wichtig, wenn die Constraintfunktion stark restriktiv ist. Dann ist das Erzeugen korrekter Individuen besonders schwierig.

4.2.3 Eingabe des EA (Erweiterung)

Bei der Eingabe des EA sind folgende Erweiterungen möglich:

- Kodierung: Kodierung nicht nur in geordneter Liste, sondern z.B. als Baum, Graph, Netz, usw.
- Verfahren: weitere festimplementierte Operatoren.

4.2.4 Laufinitialisierung (Erweiterung)

Bei Start und Stop des EA sind folgende Erweiterungen möglich:

- Parameterbelegung: Auswahl von Meta–Verfahren zur Belegung der Parameter. Ein Meta–Verfahren ist hier ein Verfahren (z.B. Simulated Annealing) zur Optimierung der Belegung der freien Parameter.

4.2.5 Online-Graphiken (Erweiterung)

Man kann sich sehr viele Möglichkeiten vorstellen, das Vorankommen von EA mit Hilfe von Graphiken zu beobachten. Im ersten Schritt sind einige Standard-Online-Graphiken wie der Verlauf der besten Fitneß und der Verlauf der mittleren Fitneß, abgetragen bzgl. der Generationen, denkbar.

4.3 Weitere Eigenschaften

Die im folgenden genannten Eigenschaften haben nichts mit der eigentlichen Funktionalität des zu erstellenden Systems EAGLE zu tun. Es sind allgemeine Software-Qualitätskriterien, die bezüglich ihrer Wichtigkeit bewertet werden.

- Änderbarkeit: Hier ist insbesondere der leichte nachträgliche Einbau festimplementierter Operatoren gemeint.
- Flexibilität: Sehr wichtig, der Anwender soll sehr viele verschiedene Verfahren mit EAGLE simulieren, und Kodierungen so gut wie möglich auf die speziellen Probleme und Verfahren abstimmen können .
- Portabilität: Dies bezieht sich insbesondere auf den C++-Compiler. Der Quellcode soll sich einfach auf jedem beliebigen C++-Compiler übersetzen lassen.
- Effizienz: EAGLE braucht nicht effizient zu arbeiten. Da es vor allem ein Forschungs- und Experimentiersystem sein soll, ist Flexibilität wichtiger als Effizienz.
- Parallelisierbarkeit: EAGLE ist nicht parallelisierbar. Zwar wäre dies wünschenswert, doch eine solche Forderung würde den organisatorischen Rahmen sprengen.

Kapitel 5

Spezifikation des Kernsystems von EAGLE

5.1 Einführung

5.1.1 Zweck

Das zu erstellende Software–Paket EAGLE soll im Bereich der EA dazu dienen, brauchbare Lösungen aufzuspüren, verschiedene Verfahren miteinander zu vergleichen und Erkenntnisse über die Güte von Parametereinstellungen und verschiedenen Kodierungen zu gewinnen. Es soll und kann den Benutzer nicht bei der Erstellung effizienter und optimierter Algorithmen zur Lösung spezieller Probleme unterstützen. Als Zielgruppe stellen wir uns den Anwender vor, der schon gewisse Kenntnisse im Bereich der EA hat und diese Kenntnisse auf einfache Art und Weise anwenden möchte.

Dies erfordert, daß die Eingabe und Formulierung eines EA schnell und einfach gehen muß. Die Bedeutung der Geschwindigkeit, mit der Ergebnisse erzielt werden, rangiert dabei an zweiter Stelle. Diese Spezifikation beschreibt nicht, wie schon im Titel angedeutet, das komplette System EAGLE, sondern nur das Kernsystem. Die Sprache LEA und der Interpreter dieser Sprache werden nicht spezifiziert.

5.1.2 Verweise

Die Spezifikation der Anforderung steht in sehr engem Zusammenhang mit dem Pflichtenheft (Abschnitt 4). Der historische Abriß der Projektgruppe (vgl. Kapitel 9.1) ist als Hintergrundinformation zum besseren Verständnis der Spezifikation hilfreich.

5.1.3 Begriffsklärung

Es folgt nun eine Reihe von Begriffsklärungen, die wegen der Lesbarkeit nicht im fortlaufenden Text gegeben werden. Dabei werden Begriffe, die in diesem Abschnitt definiert werden, in der Form *Begriff* verwendet. Die Begriffsklärungen sind nicht völlig identisch mit den Begriffsklärungen im Glossar (Anhang A). Sie unterscheiden sich im Detaillierungsgrad und der inhaltlichen Gewichtung.

- Experimentdefinition:
  Eine *Experimentdefinition* setzt sich baukastenartig aus einem *Problem*, einer *Kodierung* und einem *Verfahren* zusammen.

- Experiment:
  Ein *Experiment* beinhaltet zusätzlich zu den Teilen einer *Experimentdefinition* eine Belegung der *Parameter* und eine *log-Datei* .

- log-Datei:
  Eine *log-Datei* ist eine Datei, in der die Start– und die Endpopulation eines *Laufes* und der bei diesem verwendete *seed* abgespeichert werden. Ein *Experiment* kann mit der Startpopulation aus der *log-Datei* gestartet werden (neuer *Lauf*) oder auf der Endpopulation aus der *log-Datei* aufsetzen (*Lauf* fortsetzen).

- Problem:
  Ein *Problem* besteht aus einer *Problemstruktur* und der Beschreibung der Fitneßberechnung einer konkreten Belegung der Problemstruktur.

- Problemstruktur:
  Die *Problemstruktur* bzw. die Struktur eines Problems besteht einer Liste von Grunddatentypen (Atomen). Solche Grunddatentypen sind Boolean, Integer mit Wertebereich, Real mit Wertebereich und Permutation mit Angabe der Länge der Permutation.

- Kodierung:
  Eine *Kodierung* besteht aus der *Kodierungsstruktur* und der *Kodierungsfunktion*.

- Kodierungstruktur:
  Die *Kodierungstruktur* besteht aus einer Liste von Atomen. Eine *Kodierungstruktur* ist genau dann mit einem *Problem* verträglich, wenn eine injektive, strukturerhaltende Abbildung der *Problemstruktur* auf die *Kodierungsstruktur* existiert.

- Kodierungsfunktion:
  Die *Kodierungsfunktion* beschreibt die injektive Abbildung von der *Problemstruktur* auf die *Kodierungsstruktur*. Die *Kodierungsfunktion* ...

- bildet jedes Atom der *Problemstruktur* auf ein oder mehrere Atome der *Kodierungstruktur* ab. Dabei gibt es folgende Möglichkeiten:
  - Permutation wird auf Permutation abgebildet (Identität).
  - Boolean wird auf Boolean abgebildet (Identität).
  - Integer wird auf Integer (Identität) oder einen Bitstring (gray–kodiert oder standard) abgebildet (dabei wird die Länge des Bitstrings durch den Bereich von Integer und eine anzugebende Schrittweite festgelegt).
  - Real wird auf Real (Identität) oder einen Bitstring (gray–kodiert oder standard) abgebildet (dabei wird die Länge des Bitstrings durch den Bereich und eine anzugebende Genauigkeit von Real festgelegt).
- generiert für die *Kodierungsstruktur* zusätzliche, nicht in der *Problemstruktur* enthaltene Atome als Strategieparameter.
- legt die Anordnung der Atome der *Kodierungsstruktur* in einer Liste fest.

Dadurch entsteht eine *Kodierungsstruktur*, die automatisch mit der *Problemstruktur* verträglich ist. Die Umkehrabbildung der *Kodierungsfunktion* wird Dekodierungsfunktion genannt und dient der Rückabbildung der Individuen als Instanzen der *Kodierungsstruktur* auf Elemente des Suchraums als Instanzen der *Problemstruktur*.

- Verfahren:
  Wir verstehen unter einem *Verfahren* einen Evolutionären Algorithmus, der eine Fitneßfunktion und eine *Population* übergeben bekommt und das beste gefundene *Individuum* zurückgibt. Ein *Verfahren* besteht aus einem Hauptoperator, der weitere *Operatoren* aufrufen kann.

- Operator:
  Ein *Operator* ist die Beschreibung eines Algorithmus, der auf irgendeine Art und Weise *Individuen* oder *Populationen* manipuliert. Ein *Operator* kann entweder fest im System implementiert vorliegen oder in der eigens dafür entwickelten Sprache LEA vom Benutzer eingegeben werden. In beiden Fällen kann der *Operator* vom System EAGLE abgearbeitet werden.
  Bestandteile eines *Operators* können*Parameter* , Variablen, *Label* , Operatoraufrufe, *Filter* und Kontrollkonstrukte sein.
  Die *Operatoren* können anhand der Eingabe- und Ausgabe-Datentypen klassifiziert werden. Für eine detailliertere Beschreibung sei auf Abschnitt 6 verwiesen.

- Label:
  Ein *Label* ist eine Markierung im *Verfahren*. Vom Benutzer interaktiv eingegebene Steuerungsbefehle werden bei der Erkennung der *Label* ausgeführt. *Label* werden im LEA–Code gesetzt und können in der *Laufinitialisierung* als Halte– oder Abbruchlabel definiert und aktiviert oder deaktiviert werden. Ein vom Benutzer eingegebenes „Halt“ bewirkt ein Anhalten beim Erreichen des nächsten

Haltelabels. Hingegen wird beim Erreichen eines aktiven Abbruchlabels die Abarbeitung des *Laufs* automatisch beendet.

- Filter:
  Die *Filter* werden wie *Label* im LEA–Code gesetzt und können in der *Laufinitialisierung* an ein beliebiges File gebunden und aktiviert bzw. deaktiviert werden. Ein aktiver *Filter* schreibt bei seiner Ausführung, die von ihm selektierten Daten in die mit ihm verbundene Datei.

- Individuum:
  Ein *Individuum* ist eine Belegung der *Kodierungsstruktur* mit konkreten Werten. Der Fitneßoperator berechnet die Fitneß eines *Individuums* aus den Werten der Belegung der *Problemstruktur*, die mittels der Dekodierungsfunktion aus der Belegung der *Kodierungsstruktur* errechnet werden kann. Das *Verfahren* arbeitet im Gegensatz dazu nur auf der Belegung der *Kodierungsstruktur*.

- Population:
  Eine *Population* ist eine Ansammlung von *Individuen*. Die Anzahl der in einer *Population* vorhandenen *Individuen* kann während der Abarbeitung des *Verfahrens* variieren.

- Generation:
  Eine *Generation* ist eine *Population* aus einer Folge von *Populationen* zu einem bestimmten Zeitpunkt.

- Lauf:
  Ein *Lauf* ist die Abarbeitung des LEA-Codes und die Verarbeitung der interaktiven Einflußnahme des Benutzers sowie aller auf dem Bildschirm ausgegebenen und aller in der *log-Datei* protokollierten Informationen.

- Laufinitialisierung:
  Die *Laufinitialisierung* ist die vor und auch während des *Laufs* (wobei dann eine Unterbrechung des *Laufs* nötig ist) jederzeit durchführbare Modifizierung der Parameterbelegung des *Verfahrens*, der *Filter*– und der *Label*–Belegung.

  Parameter können in der *Laufinitialisierung* mit Werten im Rahmen ihrer Wertebereiche, die im LEA-Code festgelegt werden, eingegeben werden. Die *Filter* können in der *Laufinitialisierung* aktiviert bzw deaktiviert und durch die Eingabe eines Bezeichners mit einer Datei verbunden werden. Die *Label* können in der *Laufinitialisierung* mit einem Typ versehen und aktiviert bzw. deaktiviert werden. Der Random *Seed* kann verändert und die Anfangspopulation bestimmt werden. Die Anfangspopulation kann neu (durch EAGLE generiert) oder aus einer *log-Datei* übernommen werden.

  In der Default–*Laufinitialisierung* sind die *Parameter* mit ihren Default–Werten belegt, die *Filter* deaktiviert und die *Label* aktiviert. Der Random *Seed* wird mit

einer Konstanten belegt und die Anfangspopulation ist neu. Wenn der Benutzer in einem *Experiment* bisher keine *Laufinitialisierung* vorgenommen hat, wird das *Experiment* mit der Default–*Laufinitialisierung* abgearbeitet.

- Zufallszahlengenerator und Seed:
  Ein *Zufallszahlengenerator* für EAGLE soll die Eigenschaft der Wiederholbarkeit gewährleisten. D.h. durch die Initialisierung des Generators mit einem *seed* (i.a. eine große Zahl) wird sein Zustand eindeutig beschrieben und er produziert mit demselben *seed* immer dieselbe Reihe von Zufallszahlen.

5.1.4 Übersicht

Es folgt eine kurze, allgemeine Beschreibung von EAGLE und einige Informationen, die nicht direkt zu den funktionalen Anforderungen zählen. Darauf folgt der eigentliche Kern dieses Abschnitts, die Spezifikation der funktionalen Anforderungen. Dort werden die konkreten Anforderungen an die Software beschrieben. Danach werden die Schnittstellenanforderungen nach außen, Leistungsanforderung, Design–Einschränkungen und geforderte Qualitätsmerkmale beschrieben.

5.2 Allgemeine Beschreibung

Mittels einer Beschreibungssprache für Evolutionäre Algorithmen (LEA), die innerhalb der PGA entwickelt wurde, kann ein Algorithmus mit den benötigten Datenstrukturen beschrieben werden. EAGLE bietet dem Benutzer eine graphische Oberfläche, die eine komfortable Eingabe, Konfiguration und interaktive Ablaufkontrolle eines so beschriebenen Algorithmus erlaubt. In einem festen Datenformat werden die Ergebnisse und der Ablauf des Algorithmus in geeigneter Weise protokolliert und festgehalten.

5.2.1 Produkt-Perspektive

EAGLE ist ein alleinstehendes Software–Produkt. Es ist allerdings daran gedacht, ein Auswertungs–Tool zu schreiben, das die durch einen Lauf erzeugten Daten nachträglich statistisch auswerten kann.

5.2.2 Produkt-Funktionen

Es folgt nun eine knappe Auflistung der geforderten Funktionalitäten:

- Eingabe einer Problemdatenstruktur
- Eingabe einer Kodierungsdatenstruktur
- Eingabe einer Fitneßfunktion in LEA–Code
- Eingabe von Operatoren in LEA–Code
- Eingabe von Belegungen für Parameter, Label und Filter
- Interaktive Kontrolle des Ablaufs des EA
- Ausgabe des Fortschritts der Abarbeitung des EA
- Protokollierung der Abarbeitung des EA

Innerhalb des Systems EAGLE können keine gespeicherten Strukturen (Experimentdefinition, Experiment, Problem, Kodierung, Verfahren, Operator und Laufinitialisierung) gelöscht werden. Dieses kann ausschließlich von der Betriebssystemsebene aus geschehen.

5.2.3 Hardware-Umgebung

Es liegt folgende Hardware vor:

Server: Sun SPARCstation 10/40 mit 128MB Haupspeicher und zweitem Prozessor

Terminals: Tektronix XP354 X-Terminal

5.2.4 Benutzercharakterisierung

Je nachdem, wie tief der Benutzer in EAGLE einsteigen will, sind unterschiedliche Vorkenntnisse sinnvoll. Für das einfache Ausprobieren der Wirkungsweise von EA, sind lediglich geringe Kenntnisse über die Ideen von EA notwendig. Will der Benutzer tiefer in das System einsteigen und selbst Operatoren mit Hilfe von LEA definieren, so sind Kenntnisse in Modula–2 oder einer vergleichbaren prozeduralen Programmiersprache hilfreich, um mit der Syntax und Semantik von LEA, die sich an Modula–2 orientiert, zurechtzukommen.

5.3 Spezifische Anforderungen

In diesem Abschnitt werden alle Details, die für die anschließende Design-Phase relevant sind, angegeben.

5.3.1 Funktionale Anforderungen

Es folgt eine Beschreibung der Transformation der Eingabe in die Ausgabe. Diese Beschreibung ist der wichtigste Ausgangspunkt für die Design–Phase und ist der eigentliche Kern der Spezifikation.
Die Struktur, nach der sich die Aufzählung der Funktionalitäten richtet, orientiert sich an der Fensterstruktur einer Benutzerführung. Deshalb werden die Funktionalitäten, entsprechend ihrer Zugehörigkeit zu Fenstern, gemeinsam beschrieben.

Die Beschreibung der Funktionen gliedert sich dabei jeweils in die folgenden Punkte.

- Einleitung: Eine allgemeine Beschreibung der Funktionen bzw. Beschreibung des Fensters.
- Eingabe: Eine möglichst genaue Beschreibung der Eingabewerte der Funktionen.
- Vorgangsbeschreibung: Definition der Operationen, die innerhalb der Funktionen ausgeführt werden müssen (z.B. Ausnahmebehandlung, Wertebereichsprüfung), oder Beschreibung eines zu benutzenden Algorithmus.
- Ausgabe: Eine möglichst genaue Beschreibung der Ausgabewerte der Funktionen.

Ein Übersicht über die Hierarchie der verschiedenen Fenster kann der Abbildung 5.1 entnommen werden.

1. **Fenster „Experimentdefinition“**

   (a) Einleitung: Das Fenster auf oberster Ebene der Fensterhierarchie enthält die Funktionen

      i. *Experimentdefinition laden*
      ii. *Experimentdefinition speichern*
      iii. *Experimentdefinition neu*
      iv. *edit Problem*
      v. *edit Kodierung*
      vi. *edit Verfahren*
      vii. *Experiment*
      viii. *EAGLE beenden*

      In diesem Fenster kann eine Experimentdefinition geladen, gespeichert oder neu erstellt werden. Problem, Kodierung und Verfahren können editiert werden. Zur aktuellen Experimentdefinition kann ein Experiment erstellt werden und das Programm kann beendet werden. Es gibt in diesem

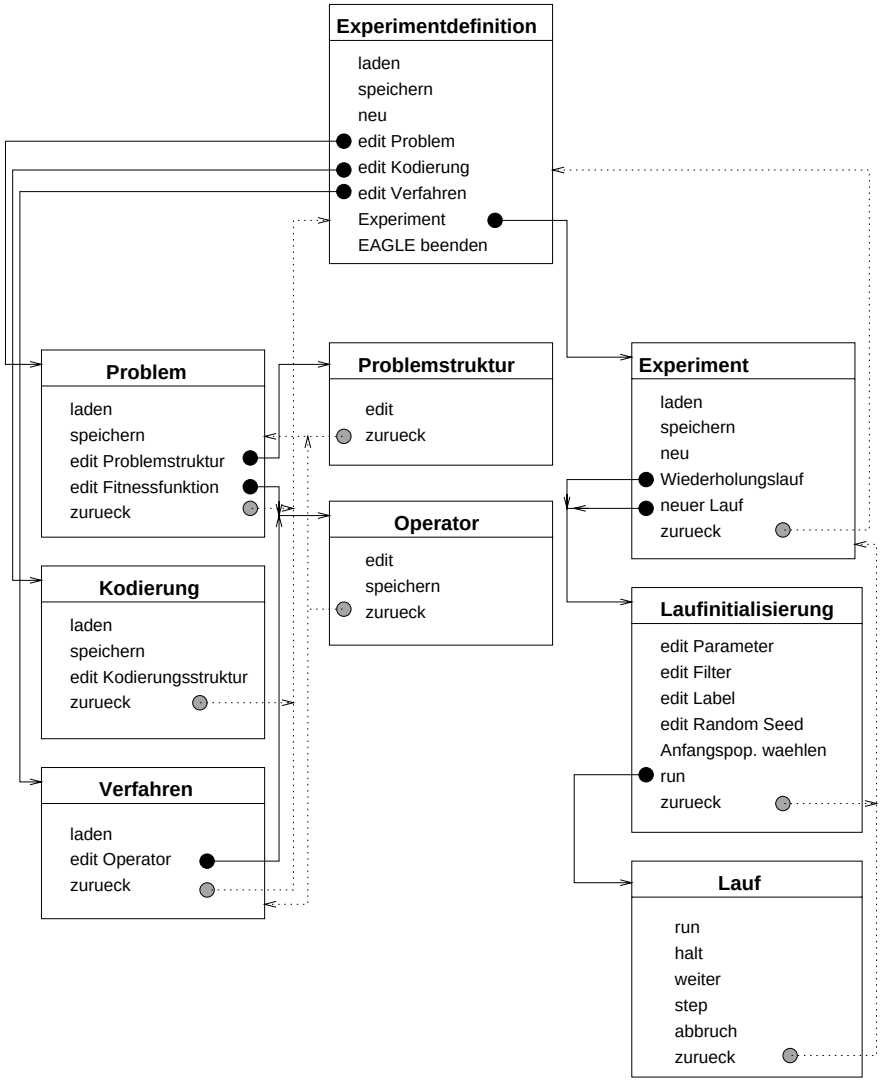


Abbildung 5.1: Hierarchie der verschiedenen Fenster

und den Fenstern „Problem“, „Kodierung“ und „Verfahren“ ein Feld, in dem der Bezeichner der aktuellen Experimentdefinition angezeigt wird. Wird in diesem Fenster „Experimentdefinition“ kein Bezeichner angezeigt, ist entweder keine Experimentdefinition vorhanden oder eine geladene Experimentdefinition wurde verändert.

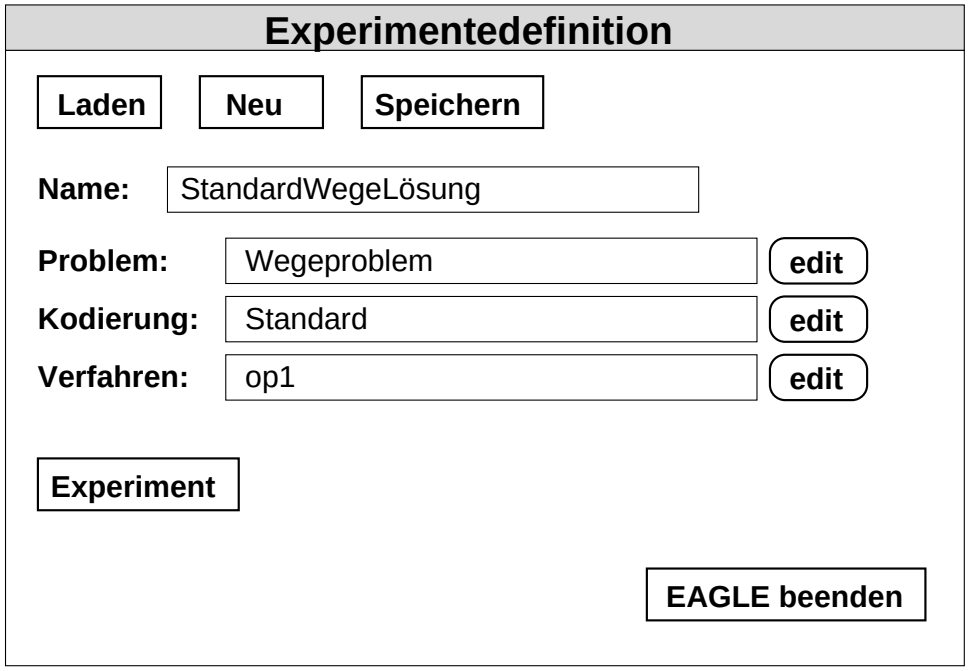


Abbildung 5.2: Fenster „Experimentdefinition“

(b) Eingabe:

i. *Experimentdefinition laden*
Aus einer Liste von vorhandenen Experimentdefinitionen wählt der Benutzer eine Experimentdefinition aus.

ii. *Experimentdefinition speichern*
Der Benutzer gibt einen Bezeichner für die zu speichernde Experimentdefinition ein. Der Bezeichner muß mit einem Buchstaben beginnen und darf keine Sonderzeichen enthalten.

iii. *Experimentdefinition neu*
keine Eingabe.

iv. *edit Problem*
keine Eingabe.

v. *edit Kodierung*
keine Eingabe.

vi. *edit Verfahren*
keine Eingabe.

vii. *Experiment*
keine Eingabe.

viii. *EAGLE beenden*
keine Eingabe

(c) Vorgangsbeschreibung:

i. *Experimentdefinition laden*
Zuerst wird geprüft, ob es eine aktuelle Experimentdefinition gibt, die noch nicht gespeichert ist. Ist dies der Fall, wird ein Warnmechanismus ausgelöst. Andernfalls werden alle Teile, die zu der ausgewählten Experimentdefinition gehören, geladen und das System damit initialisiert.

ii. *Experimentdefinition speichern*
Es wird überprüft, ob der eingegebene Bezeichner syntaktisch korrekt und noch nicht vergeben ist. Wird der Bezeichner akzeptiert, so werden unter dem Bezeichner der Experimentdefinition alle zur Experimentdefinition gehörigen Teile abgespeichert. Ansonsten wird eine Fehlerbehandlung ausgelöst.

iii. *Experimentdefinition neu*
Wie bei Experimentdefinition laden, wird überprüft, ob es eine aktuelle ungespeicherte Experimentdefinition gibt. Ist dies der Fall wird wie oben ein Warnmechanismus ausgelöst. Andernfalls wird eine neue, leere Experimentdefinition erzeugt.

iv. *edit Problem*
Es wird geprüft, ob eine Experimentdefinition geladen ist. Falls in der aktuellen Experimentdefinition ein Problem geladen ist, wird das Fenster „Problem“ mit dem aktuellen Problem aufgerufen. Andernfalls wird das Fenster „Problem“ ohne initiales Problem aufgerufen.

v. *edit Kodierung*
Es wird geprüft, ob eine Experimentdefinition geladen ist. Falls in der aktuellen Experimentdefinition ein Problem geladen ist. Ist dies nicht der Fall, wird eine Fehlermeldung ausgelöst. Gibt es in der aktuellen Experimentdefinition eine geladenen Kodierung, wird das Fenster „Kodierung“ mit der aktuellen Kodierung und dem aktuellen Problem aufgerufen. Andernfalls wird das Fenster „Kodierung“ ohne initiale Kodierung mit dem aktuellen Problem aufgerufen.

vi. *edit Verfahren*
Es wird geprüft, ob eine Experimentdefinition geladen ist. Falls in der aktuellen Experimentdefinition ein Verfahren geladen ist, wird das Fenster „Verfahren“ mit dem aktuellen Verfahren aufgerufen. Andernfalls wird das Fenster „Verfahren“ ohne initiales Verfahren aufgerufen.

vii. *Experiment*
Es wird geprüft, ob eine Experimentdefinition geladen und gespeichert ist.

viii. *EAGLE beenden*
Es wird geprüft, ob die aktuelle Experimentdefinition abgespeichert wurde. Ansonsten wird ein Warnmechanismus ausgelöst.

(d) Ausgabe:

i. *Experimentdefinition laden*
Ist die aktuelle Experimentdefinition abgespeichert, wird der Bezeichner der ausgewählten Experimentdefinition angezeigt und die ausgewählte Experimentdefinition wird zur aktuellen Experimentdefinition. Sonst erscheint eine Warnung auf dem Bildschirm, die besagt, daß die aktuelle Experimentdefinition noch nicht abgespeichert ist. Diese Warnung kann mit den Buttons *zurück*, *speichern* oder *ignorieren* verlassen werden.

ii. *Experimentdefinition speichern*
Ein Fenster erscheint, in dem der Benutzer einen Bezeichner eingeben kann. Wird der Bezeichner nicht akzeptiert, erscheint eine Fehlermeldung, die besagt, aus welchem Grund der eingegebene Bezeichner nicht akzeptiert wurde. Diese muß der Benutzer quittieren. Danach kann der Benutzer erneut eine Funktion aus dem Fenster „Experimentdefinition“ anwählen.

iii. *Experimentdefinition neu*
Ist die aktuelle Experimentdefinition abgespeichert, erscheint das Feld für den Experimentdefinitionsbezeichner als leeres Feld. Sonst erscheint dieselbe Warnung, wie unter „Experimentdefinition laden“.

iv. *edit Problem*
Ist eine Experimentdefinition geladen, so erscheint das Fenster „Problem“mit einem eventuell bereits geladenen Problem. Ansonsten erscheint eine Fehlermeldung, die vom Benutzer zu quittieren ist. Danach kann der Benutzer erneut eine Funktion aus dem Fenster „Experimentdefinition“ anwählen.

v. *edit Kodierung*
Ist eine Experimentdefinition geladen und ist ein Problem vorhanden, erscheint das Fenster „Kodierung“mit einer eventuell bereits geladenen Kodierung. Ansonsten erscheint eine Fehlermeldung, die vom Benutzer zu quittieren ist. Danach kann der Benutzer erneut eine Funktion aus dem Fenster „Experimentdefinition“ anwählen.

vi. *edit Verfahren*
Ist eine Experimentdefinition geladen, so erscheint das Fenster „Verfahren“ mit einem eventuell bereits geladenen Verfahren. Ansonsten

erscheint eine Fehlermeldung, die vom Benutzer zu quittieren ist. Danach kann der Benutzer erneut eine Funktion aus dem Fenster „Experimentdefinition“ anwählen.

vii. *Experiment*
Ist eine Experimentdefinition geladen und gespeichert, erscheint das Fenster „Experiment“. Ansonsten erscheint eine Fehlermeldung, die vom Benutzer zu quittieren ist. Danach kann der Benutzer erneut eine Funktion aus dem Fenster „Experimentdefinition“ anwählen.

viii. *EAGLE beenden*
Ist die aktuelle Experimentdefinition abgespeichert, wird das Programm EAGLE beendet. Sonst erscheint dieselbe Warnung wie unter *Experimentdefinition laden.*

2. **Fenster „Experiment“**

(a) Einleitung: Das Fenster „Experiment“ enthält die Funktionen

i. *Experiment laden*

ii. *Experiment speichern*

iii. *Experiment neu*

iv. *Wiederholungslauf*

v. *neuer Lauf*

vi. *zurück*

In diesem Fenster kann ein Experiment zu der aktuellen Experimentdefinition geladen, gespeichert oder neu erstellt werden. Es kann ein altes Experiment nachvollzogen werden (*Wiederholungslauf*) oder ein neues Experiment gestartet werden (*neuer Lauf*). Mit *zurück* gelangt man in das Fenster „Experimentdefinition“ zurück. Es gibt in diesem und den Fenstern „Run“ und „Laufinitialisierung“ ein Feld, in dem der Bezeichner des aktuellen Experiments angezeigt wird.
Wird in diesem Fenster „Experiment“ kein Bezeichner angezeigt, ist entweder kein Experiment geladen oder ein geladenes Experiment wurde verändert.

(b) Eingabe:

i. *Experiment laden*
Aus einer Liste von zur aktuellen Experimentdefinition vorhandenen Experimenten wählt der Benutzer ein Experiment aus.

ii. *Experiment speichern*
Der Benutzer gibt einen Bezeichner für das zu speichernde Experiment ein. Der Bezeichner muß mit einem Buchstaben beginnen und darf keine Sonderzeichen enthalten.

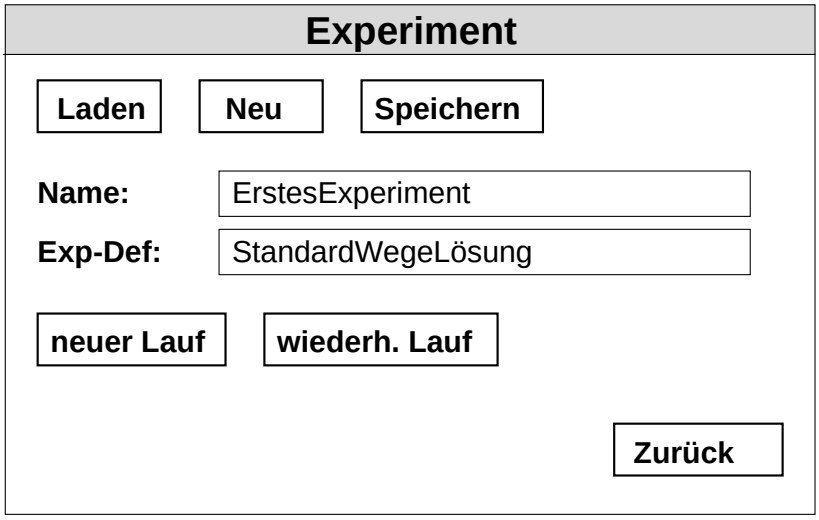


Abbildung 5.3: Fenster „Experiment“

iii. *Experiment neu*
keine Eingabe.

iv. *Wiederholungslauf*
keine Eingabe.

v. *neuer Lauf*
keine Eingabe.

vi. *zurück*
keine Eingabe.

(c) Vorgangsbeschreibung:

i. *Experiment laden*
Zuerst wird geprüft, ob es ein aktuelles Experiment gibt, das noch nicht gespeichert ist. Ist dies der Fall, wird ein Warnmechanismus ausgelöst. Andernfalls werden alle Teile, die zu dem ausgewählten Experiment gehören, geladen und das System damit initialisiert.

ii. *Experiment speichern*
Es wird überprüft, ob der eingegebene Bezeichner syntaktisch korrekt und noch nicht vergeben ist. Trifft dies zu, so werden unter dem Bezeichner des Experiments die zum Experiment gehörigen Teile abgespeichert. Es wird eine logische Verbindung zur aktuellen Experimentdefinition erstellt. Ansonsten wird eine Fehlerbehandlung ausgelöst.

iii. *Experiment neu*
Wie bei Experiment laden, wird überprüft, ob es ein aktuelles ungespeichertes Experiment gibt. Ist dies der Fall wird wie oben ein Warnmechanismus ausgelöst. Andernfalls wird ein neues Experiment

zu der aktuellen Experimentdefinition erzeugt.

iv. *Wiederholungslauf*
Es wird geprüft, ob ein altes, nicht verändertes Experiment geladen ist. Ansonsten wird eine Fehlermeldung ausgelöst.

v. *neuer Lauf*
Es wird überprüft, ob ein Experiment geladen ist. Ansonsten wird eine Fehlermeldung ausgelöst.

vi. *zurück*
Es wird geprüft, ob das aktuelle Experiment abgespeichert wurde. Sonst wird ein Warnmechanismus ausgelöst.

(d) Ausgabe:

i. *Experiment laden*
Ist das aktuelle Experiment abgespeichert, wird der Bezeichner des ausgewählten Experiment angezeigt und das ausgewählte Experiment wird zum aktuellen Experiment. Sonst erscheint eine Warnung auf dem Bildschirm, die besagt, daß das aktuelle Experiment noch nicht abgespeichert ist. Diese Warnung kann mit den Buttons *zurück*, *speichern* oder *ignorieren* verlassen werden.

ii. *Experiment speichern*
Ein Fenster erscheint, in dem der Benutzer einen Bezeichner eingeben kann. Wird der Bezeichner nicht akzeptiert, erscheint eine Fehlermeldung, die besagt, aus welchem Grund der eingegebene Bezeichner nicht akzeptiert wurde. Diese muß der Benutzer quittieren. Danach kann der Benutzer erneut eine Funktion aus dem Fenster „Experiment“ anwählen.

iii. *Experiment neu*
Ist das aktuelle Experiment abgespeichert, erscheint das Feld für den Experimentbezeichner als leeres Feld. Sonst erscheint dieselbe Warnung, wie unter Experiment laden.

iv. *Wiederholungslauf*
Ist ein altes Experiment geladen, erscheint das Fenster „Lauf“, wobei in diesem dann einige Funktionen gesperrt sind (im Gegensatz zu *neuer Lauf*). Ansonsten erscheint eine Fehlermeldung, die vom Benutzer zu quittieren ist. Danach kann der Benutzer erneut eine Funktion aus dem Fenster „Experiment“ auswählen.

v. *neuer Lauf*
Ist ein Experiment geladen, erscheint das Fenster „Lauf“, ansonsten erscheint eine Fehlermeldung, die vom Benutzer zu quittieren ist. Danach kann der Benutzer erneut eine Funktion aus dem Fenster „Experiment“ auswählen.

vi. *zurück*

Ist das aktuelle Experiment abgespeichert, erscheint wieder das Fenster „Experimentdefinition“. Sonst erscheint dieselbe Warnung wie unter *Experiment laden*.

3. **Fenster „Problem“**

   (a) Einleitung: Das Fenster „Problem“ hat die Funktionen

      i. *laden*
      ii. *speichern*
      iii. *edit Problemstruktur*
      iv. *edit Fitneßoperator*
      v. *zurück*

   Dieses Fenster ist die Instanz aus der die verschiedenen Teile des Problems eingegeben werden können. Hier kann ein Problem geladen und gespeichert werden. Die Problemstruktur kann in dem Fenster „Problemstruktur“ editiert werden, in das man gelangt, wenn man die Funktion *edit Problemstruktur* wählt. Der Fitneßoperator kann im Fenster „Operator“ editiert werden. Dorthin gelangt man durch die Wahl der Funktionalität *edit Fitneßoperator*. Mit *zurück* gelangt man wieder ins Fenster „Experimentdefinition“.

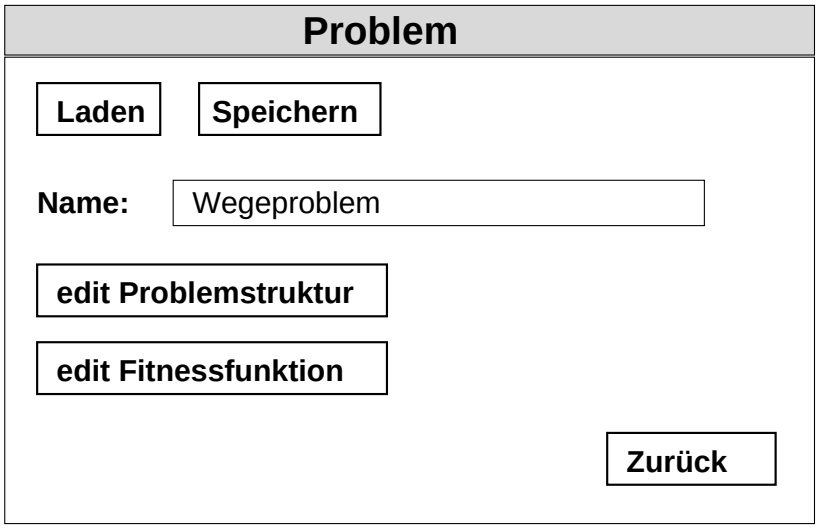


Abbildung 5.4: Fenster „Problem“

   (b) Eingabe:

      i. *laden*
      Der Benutzer wählt aus einer Liste der Bezeichner existierender Probleme einen Bezeichner für ein Problem aus.

ii. *speichern*
Der Benutzer gibt einen Bezeichner für das neue Problem ein. Der Bezeichner muß mit einem Buchstaben beginnen und darf keine Sonderzeichen enthalten.

iii. *edit Problemstruktur*
Der Benutzer wählt aus einer Liste aller Problemstrukturen, in der die zu dem aktuellen Problem gehörenden Problemstruktur markiert ist, eine zu editierenden Problemstruktur aus. In dieser Liste gibt es auch stets eine neue, leere Problemstruktur, die auch ausgewählt werden kann.

iv. *edit Fitneßoperator*
Der Benutzer wählt aus einer Liste aller Fitneßoperatoren, in der der zu dem aktuellen Problem gehörende Fitneßoperator markiert ist, einen zu editierenden Fitneßoperator aus. In dieser Liste gibt es auch stets einen neuen, leeren Fitneßoperator, der auch ausgewählt werden kann.

v. *zurück*
Keine Eingabe.

(c) Vorgangsbeschreibung:

i. *laden*
Zuerst wird geprüft, ob es ein aktuelles, noch nicht gespeichertes Problem gibt. Ist dies der Fall, wird ein Warnmechanismus ausgelöst. Andernfalls werden alle Teile, die zu dem ausgewählten Problem gehören, geladen und das System damit initialisiert.

ii. *speichern*
Es wird überprüft, ob der eingegebene Bezeichner syntaktisch korrekt und noch nicht vergeben ist. Wird der Bezeichner akzeptiert, so werden alle Teile des Problems unter diesem Bezeichner abgelegt. Es wird eine logische Verbindung zur aktuellen Experimentdefinition erstellt. Ansonsten wird eine Fehlerbehandlung ausgelöst.

iii. *edit Problemstruktur*
Das Fenster „Problemstruktur“ wird mit der ausgewählten Problemstruktur aufgerufen.

iv. *edit Fitneßoperator*
Das Fenster „Operator“ wird mit dem ausgewählten Fitneßoperator aufgerufen.

v. *zurück*
Es wird geprüft, ob das aktuelle Problem gespeichert ist. Andernfalls wird ein Warnmechanismus ausgelöst.

(d) Ausgabe:

i. *laden*
Ist das aktuelle Problem abgespeichert, wird der Bezeichner des ausgewählten Problems angezeigt und das ausgewählte Problem wird zum aktuellen Problem. Sonst erscheint eine Warnung auf dem Bildschirm, die besagt, daß das aktuelle Problem noch nicht abgespeichert ist. Diese Warnung kann mit den Buttons *zurück*, *speichern* oder *ignorieren* verlassen werden.

ii. *speichern*
Ein Fenster erscheint, in dem der Benutzer einen Bezeichner eingeben kann. Wird der Bezeichner nicht akzeptiert, erscheint eine Fehlermeldung, die besagt, aus welchem Grund der eingegebene Bezeichner nicht akzeptiert wurde. Diese ist vom Benutzer zu quittieren. Danach kann der Benutzer erneut eine Funktion aus dem Fenster „Problem“ anwählen.

iii. *edit Problemstruktur*
Es erscheint das Fenster „Problemstruktur“ mit der ausgewählten Problemstruktur.

iv. *edit Fitneßoperator*
Es erscheint das Fenster „Operator“ mit dem ausgewählten Fitneßoperator.

v. *zurück*
Ist das aktuelle Problem abgespeichert, erscheint wieder das Fenster „Experimentdefinition“. Sonst erscheint dieselbe Warnung wie unter *Problem laden*.

4. **Fenster „Problemstruktur“**

(a) Einleitung: Das Fenster „Problemstruktur“ hat die Funktionen

i. *edit*

ii. *zurück*

Mit der Funktion *edit* läßt sich die Problemstruktur in einer Tabelle editieren. Mit *zurück* gelangt man wieder ins Fenster „Problem“. Neben der Tabelle wird auch der Name des dazugehörigen Problems angezeigt. Dieses Feld ist nicht editierbar.

(b) Eingabe:

i. *edit*
Der Benutzer kann in einer Liste die einzelnen Elemente der Problemstruktur eingeben. Für jedes Element wird dabei der Problemdatentyp und eine optionale Bemerkung eingegeben. Zusätzlich sind noch bei Permutationen die Eingabe der Anzahl der Permutationselemente und bei Integer– und Real–Zahlen die Eingabe der oberen und unteren Grenze notwendig.

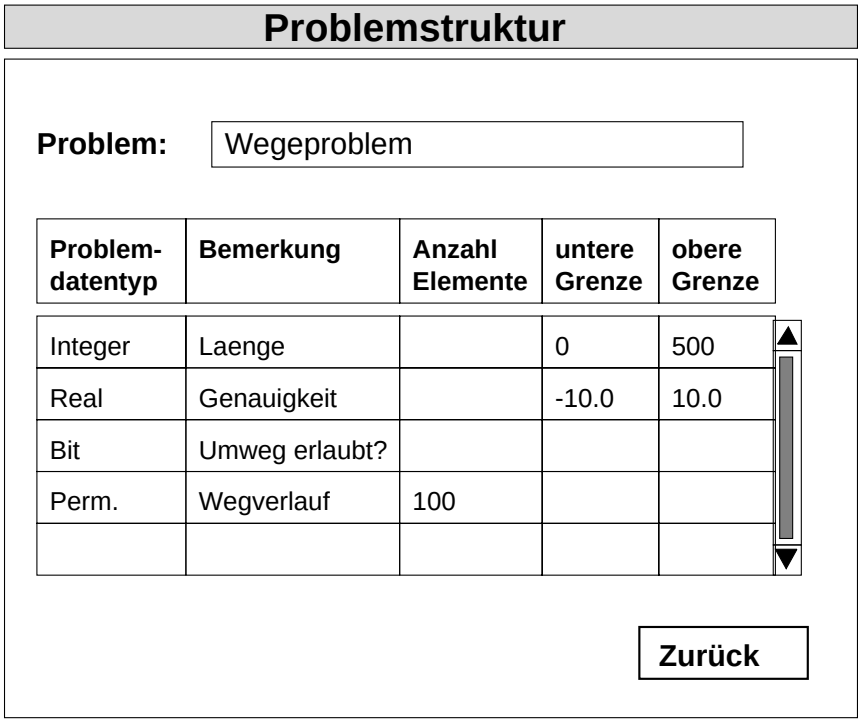


Abbildung 5.5: Fenster „Problemstruktur“

ii. *zurück*
Keine Eingabe.

(c) Vorgangsbeschreibung:

i. *edit*
Nach jeder Eingabe in einem Feld wird geprüft, ob diese Eingabe ein korrektes Format besitzt. Andernfalls wird ein Warnmechanismus ausgelöst.

ii. *zurück*
Es wird geprüft, ob die kompletten Eingaben in der Tabelle miteinander verträglich sind. Andernfalls wird ein Warnmechanismus ausgelöst.

(d) Ausgabe:

i. *edit*
Hatte die Eingabe ein falsches Format, erscheint eine Fehlermeldung und der Benutzer hat die Wahl die Eingabe erneut zu editieren oder sie zu verwerfen. War die Eingabe korrekt oder wurde eine falsche Eingabe verworfen, kann anschließend wieder eine Funktion im Fenster „Problemstruktur“ angewählt werden.

ii. *zurück*
Falls keine Fehler festgestellt wurden, erscheint wieder das Fenster „Problem“. Sonst erscheint ein Warnhinweis, bei dem der Benutzer die Wahl hat, die fehlerhafte Eingabe zu korrigieren oder die Eingaben zu verwerfen.

5. **Fenster „Kodierung“**

(a) Einleitung: Das Fenster „Kodierung“ hat die Funktionen

i. *laden*

ii. *speichern*

iii. *edit Kodierungsstruktur*

iv. *zurück*

In dem Fenster „Kodierung“ wird die aktuelle Problemstruktur angezeigt. Hier kann eine Kodierung geladen und gespeichert werden. Die Kodierungsstruktur kann in einer Tabelle editiert werden. Mit *zurück* gelangt man wieder ins Fenster „Experimentdefinition“.

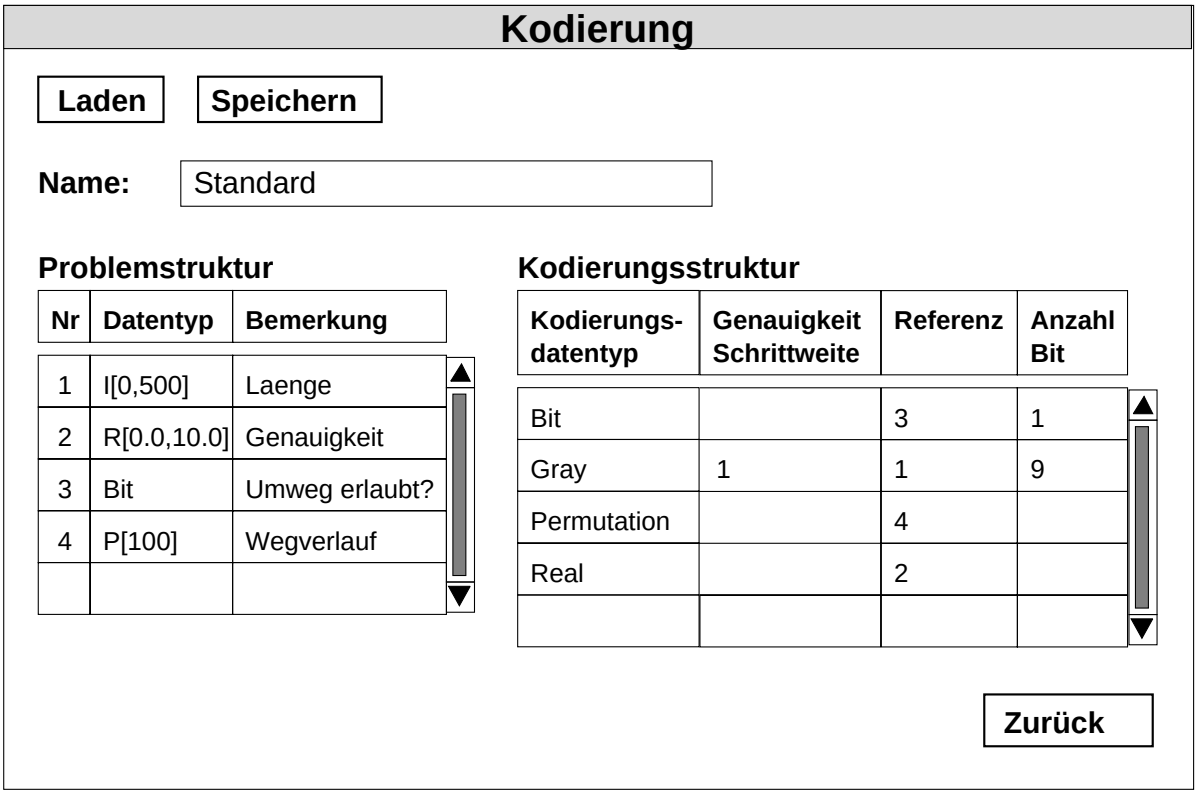


Abbildung 5.6: Fenster „Kodierung“

(b) Eingabe:

i. *laden*
Der Benutzer wählt aus einer Liste der Bezeichner aller existierender Kodierungen eine Kodierung aus. In dieser Liste sind alle Kodierungen, die speziell zu der aktuellen Problemstruktur definiert wurden, markiert.

ii. *speichern*
Der Benutzer gibt einen Bezeichner für die neue Kodierung ein. Der Bezeichner muß mit einem Buchstaben beginnen und darf keine Sonderzeichen enthalten.

iii. *edit Kodierungsstruktur*
Der Benutzer kann in einer Liste die einzelnen Elemente der Kodierungsstruktur eingeben. Für jedes Element wird dabei der Kodierungsdatentyp, ev. ein Schrittweite bzw. Genauigkeit und eine Referenz auf ein Element der Problemstruktur eingegeben.

iv. *zurück*
Keine Eingabe.

(c) Vorgangsbeschreibung:

i. *laden*
Zuerst wird geprüft, ob es eine aktuelle, noch nicht gespeicherte Kodierung gibt. Ist dies der Fall, wird ein Warnmechanismus ausgelöst. Andernfalls werden alle Teile, die zu der ausgewählten Kodierung gehören, geladen und das System damit initialisiert.

ii. *speichern*
Es wird überprüft, ob der eingegebene Bezeichner syntaktisch korrekt und noch nicht vergeben ist. Wird der Bezeichner akzeptiert, so wird die Kodierung unter diesem Bezeichner abgelegt und eine logische Verbindung zum aktuellen Problem wird erstellt. Ansonsten wird eine Fehlerbehandlung ausgelöst.

iii. *edit Kodierungsstruktur*
Nach jeder Eingabe in einem Feld wird geprüft, ob diese Eingabe ein korrektes Format besitzt. Sind zu einem Element der Kodierungsstruktur alle Felder belegt, werden die Einträge auf Konsistenz überprüft. Sind die Einträge konsistent wird die Anzahl der Datentypen durch die Kodierung neu berechnet und in der Tabelle angezeigt. Wurde kein korrektes Format eingegeben oder sind die Einträge eines Elements der Kodierungsstruktur nicht konsistent, wird ein Fehlermechanismus ausgelöst.

iv. *zurück*
Es wird geprüft, ob die aktuelle Kodierung gespeichert ist. Andernfalls wird ein Warnmechanismus ausgelöst.

(d) Ausgabe:

i. *laden*
Ist die aktuelle Kodierung gespeichert, wird der Bezeichner der ausgewählten Kodierung angezeigt und die ausgewählte Kodierung wird zur aktuellen Kodierung. Sonst erscheint eine Warnung auf dem Bildschirm, die besagt, daß die aktuelle Kodierung noch nicht gespeichert ist. Diese Warnung kann mit den Buttons *zurück*, *speichern* oder *ignorieren* verlassen werden.

ii. *speichern*
Ein Fenster erscheint, in dem der Benutzer einen Bezeichner eingeben kann. Wird der Bezeichner nicht akzeptiert, erscheint eine Fehlermeldung, die besagt, aus welchem Grund der eingegebene Bezeichner nicht akzeptiert wurde. Diese ist vom Benutzer zu quittieren. Danach kann der Benutzer erneut eine Funktion aus dem Fenster „Kodierung“ anwählen.

iii. *edit Kodierungsstruktur*
Hatte die Eingabe ein falsches Format oder waren die Einträge nicht konsistent, erscheint eine Fehlermeldung und der Benutzer hat die Wahl die Eingabe erneut zu editieren oder sie zu verwerfen. War die Eingabe korrekt oder wurde eine falsche Eingabe verworfen, kann anschließend wieder eine Funktion im Fenster „Kodierung“ angewählt werden.

iv. *zurück*
Ist die aktuelle Kodierung abgespeichert, erscheint wieder das Fenster „Experimentdefinition“. Sonst erscheint dieselbe Warnung wie unter *Kodierung laden*.

6. **Fenster „Verfahren“**

(a) Einleitung: Das Fenster Verfahren hat die Funktionen

i. *laden*

ii. *edit Operator*

iii. *neuer Operator*

iv. *zurück*

In diesem Fenster „Verfahren“ kann aus einer Liste von Hauptoperatoren ein Hauptoperator ausgewählt und das System damit geladen werden. Aus der Liste aller Operatoren, in der die zu einem Hauptoperator gehörigen Operatoren markiert sind, kann ein Operator mit *edit Operator* ausgewählt werden. Mit *zurück* gelangt man in das Fenster „Experimentdefinition“ zurück.

(b) Eingabe:

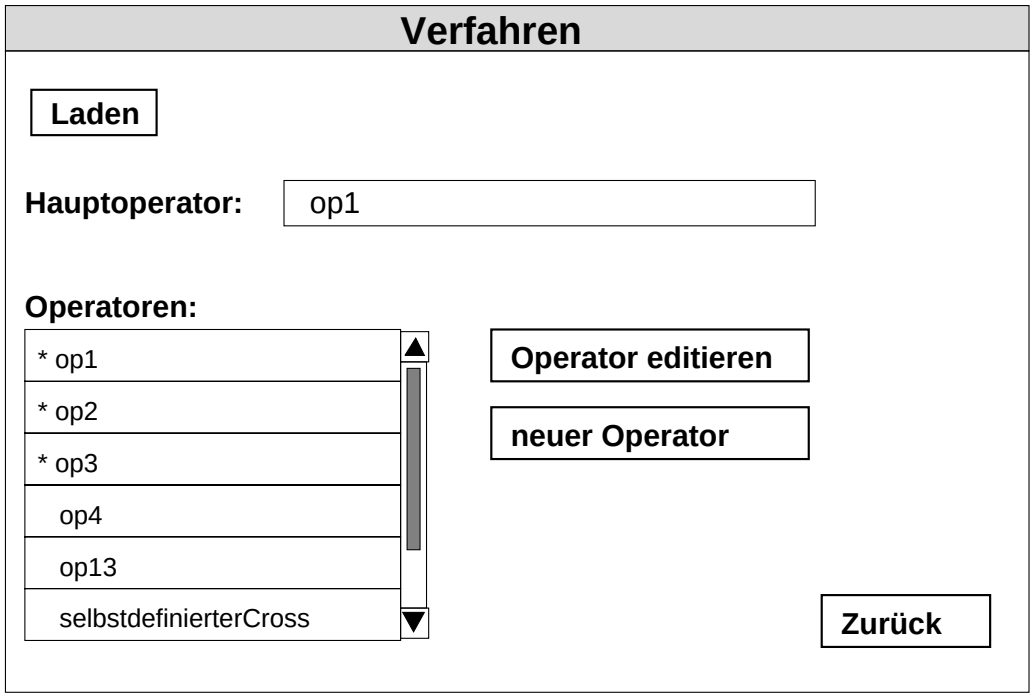


Abbildung 5.7: Fenster „Verfahren“

i. *laden*
Der Benutzer wählt aus einer Liste der Bezeichner der existierenden Hauptoperatoren einen Bezeichner für einen Hauptoperator aus.

ii. *edit Operator*
Der Benutzer wählt aus einer Liste aller Operatoren, in der die zu einem eventuell geladenen Hauptoperator gehörenden Operatoren markiert sind, einen Bezeichner für einen zu editierenden Operator aus.

iii. *neuer Operator* Keine Eingabe.

iv. *zurück*
Keine Eingabe.

(c) Vorgangsbeschreibung:

i. *laden*
Zuerst wird geprüft, ob es ein aktuelles, noch nicht gespeichertes Verfahren gibt. Ist dies der Fall, wird ein Warnmechanismus ausgelöst. Andernfalls werden alle Teile, die zu dem ausgewählten Verfahren gehören, geladen und das System damit initialisiert. In der Liste aller Operatoren werden die zu dem jetzt aktuellen Hauptoperator gehörenden Operator markiert.

ii. *edit Operator*

Das Fenster „Operator“ wird mit dem ausgewählten Operator aufgerufen.

iii. *neuer Operator* Das Fenster „Operator“ wird aufgerufen.

iv. *zurück*
Es werden , falls nicht schon erfolgt, alle zum Verfahren gehörenden Teile initialisiert.

(d) Ausgabe:

i. *laden*
Ist das aktuelle Verfahren abgespeichert, wird der Bezeichner des ausgewählten Verfahrens angezeigt und das ausgewählte Verfahren wird zum aktuellen Verfahren. Die Liste aller Operatoren wird aktualisiert und alle Operatoren, die zu dem jetzt aktuellen Hauptoperator gehören, werden markiert. Sonst erscheint eine Warnung auf dem Bildschirm, die besagt, daß das aktuelle Verfahren noch nicht abgespeichert ist. Diese Warnung kann mit den Buttons *zurück*, *speichern* oder *ignorieren* verlassen werden.

ii. *edit Operator*
Es erscheint das Fenster „Operator“ mit dem ausgewählten Operator.

iii. *neuer Operator* Es erscheint das Fenster „Operator“.

iv. *zurück*
Es erscheint das Fenster „Experimentdefinition“.

7. **Fenster „Operator“**

(a) Einleitung: Das Fenster „Operator“ hat die Funktionen

i. *edit*

ii. *speichern*

iii. *zurück*

In diesem Fenster „Operator“ kann ein Operator editiert werden. Aus dem Verfahren heraus ist ein Operator geladen, der hier verändert werden kann.

Mit *edit* kann ein Operator in der zu EAGLE gehörenden Sprache LEA eingeben werden.
Mit *laden* kann ein Operator gespeichert werden. Mit *zurück* gelangt man in das Fenster „Verfahren“.
Zur Beschreibung und Erklärung des Operatorkonzepts sei auf Abschnitt 6 verwiesen.

(b) Eingabe:

i. *speichern*
Der Benutzer gibt einen Bezeichner für den neuen Operator ein. Der Bezeichner muß mit einem Buchstaben beginnen und darf keine Sonderzeichen enthalten.

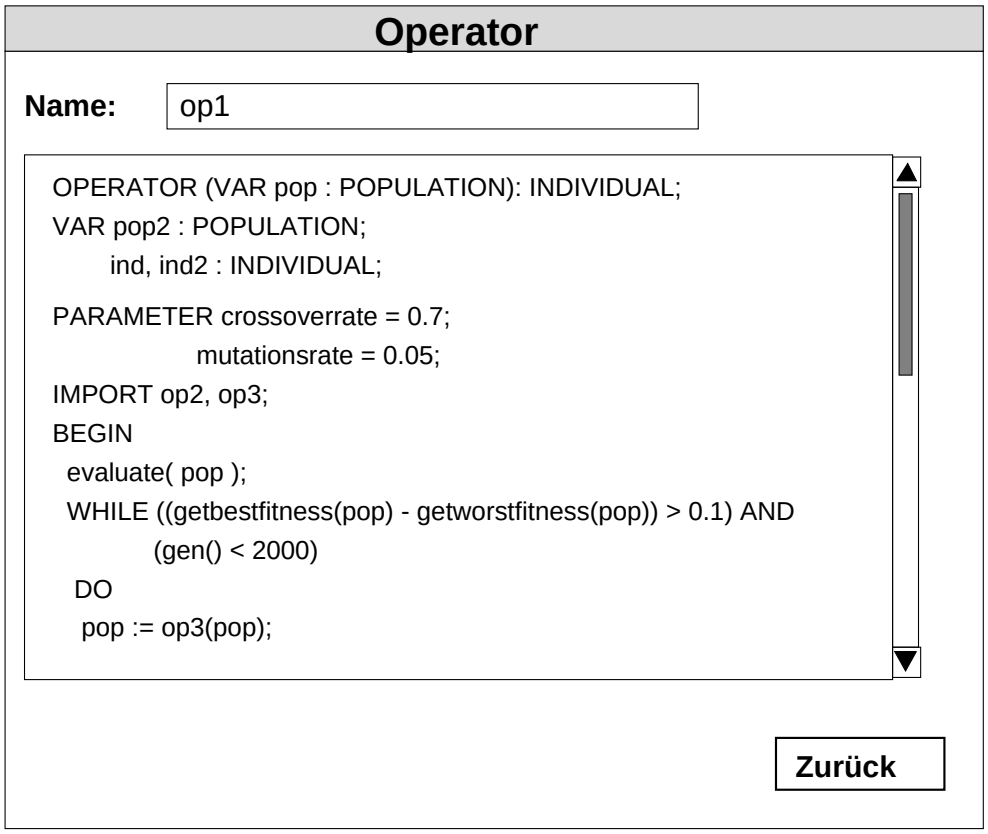


Abbildung 5.8: Fenster „Operator“

ii. *edit*
Das Fenster „Operator“ ist beim Erscheinen bereits mit einem Operator initialisiert, der in dem Editorteil des Fensters in der Programmiersprache LEA zu sehen ist. Die Funktionalität *edit* bezieht sich hier auf die Möglichkeit den vorhandenen Text des Operators zu editieren. Es stehen die gewohnten Texteditor–Funktionen zur Verfügung. Es kann allerdings nicht geladen und gespeichert werden.

iii. *zurück*
Keine Eingabe.

(c) Vorgangsbeschreibung:

i. *speichern*
Es wird überprüft, ob der eingegebene Bezeichner syntaktisch korrekt und noch nicht vergeben ist. Dann wird durch einen Testlauf des Interpreters der aktuelle Typ des Operators bestimmt und mit dem erwarteten Typ verglichen. So darf in dem Operatorfenster, wenn es vom Problemfenster aus aufgerufen wird, nur ein Fitneßoperator gespei-

chert werden. Andernfalls wird eine Fehlermeldung ausgelöst. Wird das Operatorfenster vom Verfahrensfenster aus aufgerufen, so dürfen alle Operatorentypen mit Ausnahme des Fitneßoperators gespeichert werden.

ii. *edit*
Keine Vorgangsbeschreibung.

iii. *zurück*
Es wird überprüft, ob der aktuelle Operator abgespeichert ist. Andernfalls wird ein Warnmechanismus ausgelöst.

(d) Ausgabe:

i. *speichern*
Ist der Bezeichner korrekt und stimmten die Typenvorgaben, so wird der Operator in einer Datei, die über den Bezeichner identifizierbar ist, abgespeichert. Andernfalls erscheint eine Fehlermeldung, die besagt, warum der Operator nicht abgespeichert wird. Diese ist vom Benutzer zu quittieren. Danach kann der Benutzer erneut eine Funktion aus dem Fenster „Operator“ anwählen.

ii. *edit*
In dem Texteditorteil des Fensters „Operator“ wird der jeweils aktuelle LEA–Text des Operators angezeigt.

iii. *zurück*
Ist der aktuelle Operator gespeichert , erscheint das Fenster „Verfahren“. Sonst erscheint eine Warnung auf dem Bildschirm, die besagt, daß der aktuelle Operator noch nicht abgespeichert ist. Diese Warnung kann mit den Buttons *zurück*, *speichern* oder *ignorieren* verlassen werden.

8. **Fenster „Laufinitialisierung“**

(a) Einleitung: Das Fenster „Laufinitialisierung“ hat die Funktionen

i. *edit Parameter*
ii. *edit Filter*
iii. *edit Label*
iv. *edit Random Seed*
v. *Anfangspopulation wählen*
vi. *run*
vii. *zurück*

Mit *edit Parameter* können die im Verfahren als *Parameter* deklarierten Variablen mit Werten belegt werden. Mit *edit Filter* können die Filter aktiviert werden, indem sie an Files gebunden werden. Mit *edit Label* können Labels als Halte- oder Abbruchlabel definiert und de/aktiviert werden.

Mit *edit Random Seed* kann der Seed des Zufallszahlengenerators gesetzt werden. Unter *Anfangspopulation wählen* besteht die Möglichkeit, auf eine Population in der log–Datei eines alten Experimentes zurückzugreifen. Mit *run* wird ein Lauf gestartet, d.h. es erscheint das Fenster „Lauf“ zur Kontrolle des Laufes. Mit *zurück* gelangt man in das Fenster „Experiment“. Verschiedene Funktionalitäten sind bei einem *Wiederholungslauf* nicht möglich.

(b) Eingabe:

i. *edit Parameter*
Dem Benutzer wird eine Liste mit den in den Operatoren definierten Parametern angezeigt. In dieser Liste werden jeweils der Parametername und der dazugehörige Wertebereich angezeigt. In einem weiteren Feld erfolgt als Eingabe der Wert für den Parameter. Das Feld ist bereits mit einem Default–Wert vorbelegt. Diese Funktion kann nicht bei einem Wiederhollauf aufgerufen werden.

ii. *edit Filter*
Dem Benutzer wird eine Liste mit den in den Operatoren definierten Filtern angezeigt. Zu jedem Filter wird neben dem Namen auch der Typ des Filters, also was gefiltert wird, angezeigt. In einem weitern Feld erfolgt als Eingabe der Name der Datei, in die gefiltert werden soll. Falls keine Datei angegeben wird, ist der Filter inaktiv. Diese Funktion kann nicht bei einem Wiederhollauf aufgerufen werden.

iii. *edit Label*
Dem Benutzer wird eine Liste mit den in den Operatoren definierten Labeln angezeigt. Als Eingabe kann zu jedem Label angegeben werden, ob es aktiviert ist und um welchen Typ eines Labels es sich dabei handeln soll.

iv. *Anfangspopulation wählen*
Der Benutzer kann hier eine Log–Datei eines früheren Experiments angeben und zusätzlich eingeben, ob die damalige Anfangs– oder Endpopulation als neue Anfangspopulation gewählt werden soll. Eine leere Eingabe entspricht einer zufällig gewählten Anfangspopulation.

v. *edit Random Seed*
Hier kann der Benutzer den *seed* des Zufallszahlengenerators eingeben. Diese Funktion kann nicht bei einem Wiederhollauf aufgerufen werden.

vi. *Run*
Keine Eingabe.

vii. *zurück*
Keine Eingabe.

(c) Vorgangsbeschreibung:

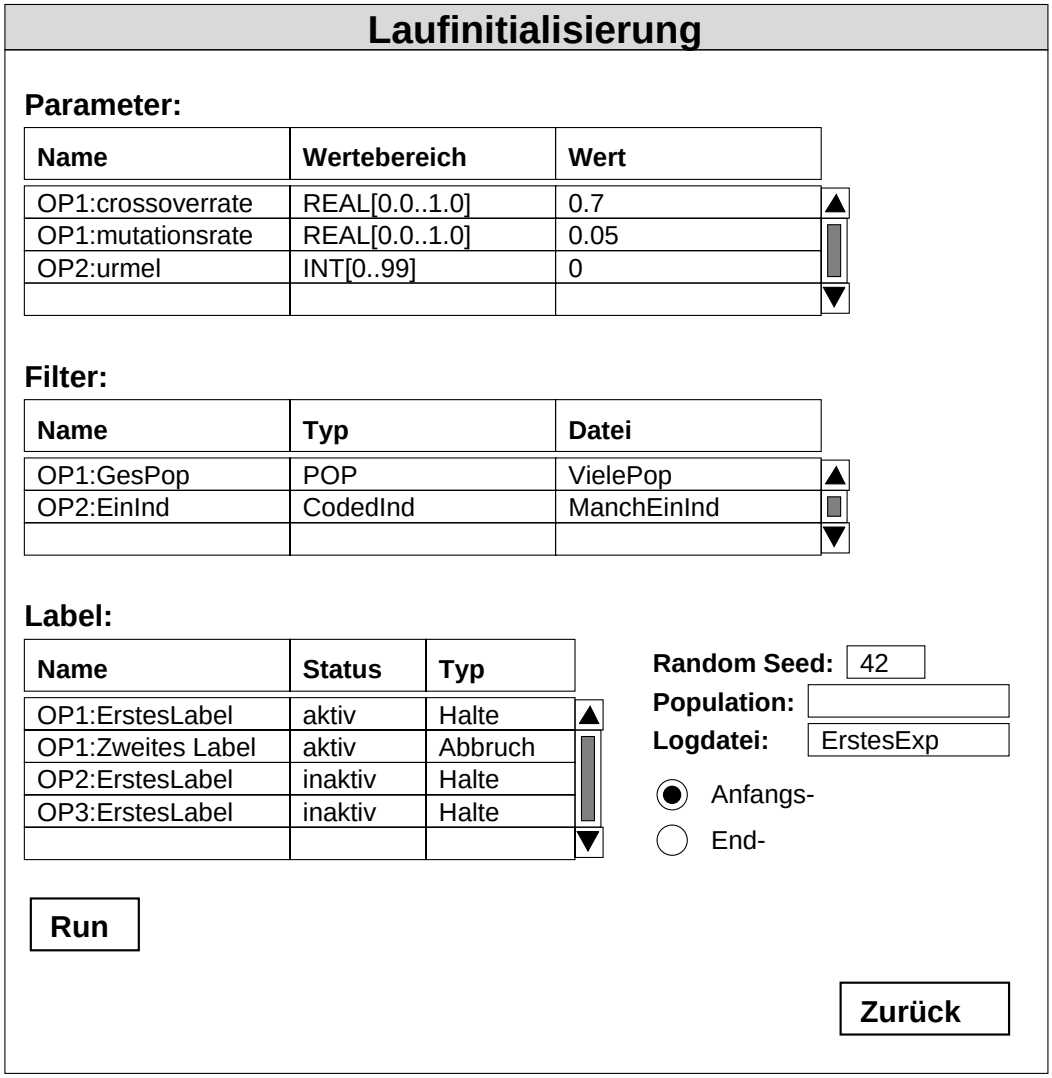


Abbildung 5.9: Fenster „Laufinitialisierung“

i. *edit Parameter*
Die für die Parameter eingegeben Werte werden auf korrektes Format, Typverträglichkeit und Einhaltung der Wertebereiche geprüft. Gegebenenfalls wird ein Fehlermechanismus ausgelöst.

ii. *edit Filter*
Bei Filtern wird der Bezeichner für eine Datei auf korrektes Format geprüft. Zusätzlich muß festgestellt werden, ob eine Datei mit diesem Bezeichner schon exisitiert. Falls dies gilt, muß der Bezeichner zurückgewiesen werden.Gegebenenfalls wird ein Fehlermechanismus ausgelöst.

iii. *edit Label*
Es wird geprüft, ob die Bezeichnung für den Typ ein korrektes Format besitzt.

iv. *Anfangspopulation wählen*
Es wird geprüft, ob das eingegebene Experiment existiert und ob es zur aktuellen Experimentdefinition paßt. Gegebenenfalls wird ein Fehlermechanismus ausgelöst.

v. *edit Random* Seed
Es wird geprüft, ob die Eingabe ein erlaubter *Seed* ist. Gegebenenfalls wird ein Fehlermechanismus ausgelöst.

vi. *Run*
Der neue Lauf wird mit den eingegebenen Daten initialisiert. Die Anfangspopulation wird in der log–Datei gesichert.

vii. *zurück*
Das Fenster „Laufinitialisierung“ wird geschlossen.

(d) Ausgabe:

i. *edit Parameter*
War die Eingabe in dem jeweiligen Feld korrekt, kann danach eine weitere beliebige Eingabe im Fenster „Laufinitialisierung“ getätigt werden. Andernfalls wird eine Meldung ausgegeben, daß die Eingabe unkorrekt war. Der Benutzer hat die Möglichkeit zurückzukehren und die Eingabe richtigzustellen oder die geänderte Eingabe zu verwerfen.

ii. *edit Filter*
War die Eingabe in dem jeweiligen Feld korrekt, kann danach eine weitere beliebige Eingabe im Fenster „Laufinitialisierung“ getätigt werden. Andernfalls wird eine Meldung ausgegeben, daß die Eingabe unkorrekt war. Der Benutzer hat die Möglichkeit zurückzukehren und die Eingabe richtigzustellen oder die geänderte Eingabe zu verwerfen.

iii. *edit Label*
War die Eingabe in dem jeweiligen Feld korrekt, kann danach eine weitere beliebige Eingabe im Fenster „Laufinitialisierung“ getätigt

werden. Andernfalls wird eine Meldung ausgegeben, daß die Eingabe unkorrekt war. Der Benutzer hat die Möglichkeit zurückzukehren und die Eingabe richtigzustellen oder die geänderte Eingabe zu verwerfen.

iv. *Anfangspopulation wählen*
War die Eingabe nicht korrekt, wird eine Meldung ausgegeben. Der Benutzer hat die Möglichkeit zurückzukehren und die Eingabe richtigzustellen oder die geänderte Eingabe zu verwerfen.

v. *edit Random Seed*
War die Eingabe in dem jeweiligen Feld korrekt, kann danach eine weitere beliebige Eingabe im Fenster „Laufinitialisierung“ getätigt werden. Andernfalls wird eine Meldung ausgegeben, daß die Eingabe unkorrekt war. Der Benutzer hat die Möglichkeit zurückzukehren und die Eingabe richtigzustellen oder die geänderte Eingabe zu verwerfen.

vi. *Run*
Es erscheint das Fenster „Lauf“, in dem die Simulation des EA–Laufes kontrolliert werden kann.

vii. *zurück*
Es erscheint wieder das Fenster „Experiment“.

9. **Fenster „Lauf“**

(a) Einleitung: Das Fenster „Lauf“ hat die Funktionen

i. *run*

ii. *halt*

iii. *weiter*

iv. *step*

v. *Abbruch*

vi. *zurück*

Nach dem betätigen des jeweiligen Buttons erfolgt für *run* der Start des EA, Für *halt* ein Anhalten des EA am nächsten Auftretens eines Halte-Labels, für *weiter* ein Fortsetzen von dieser Marke aus. Mittels *step* kann der EA veranlasst werden vom aktuellen bis zum nächsten Halte-Label zu laufen. Mit *Abbruch* kann ein EA jederzeit gestoppt werden. Mit *zurück* gelangt man in das Fenster „Experiment“. Während einem Lauf werden die Generationsnummer, die durchschnittliche Fitneß und die Fitneß des besten Individuums in der derzeitigen Population angezeigt.

(b) Eingabe:

i. *run*
Keine Eingabe. Die Funktion ist nur aktiv, wenn keine Simulation läuft.

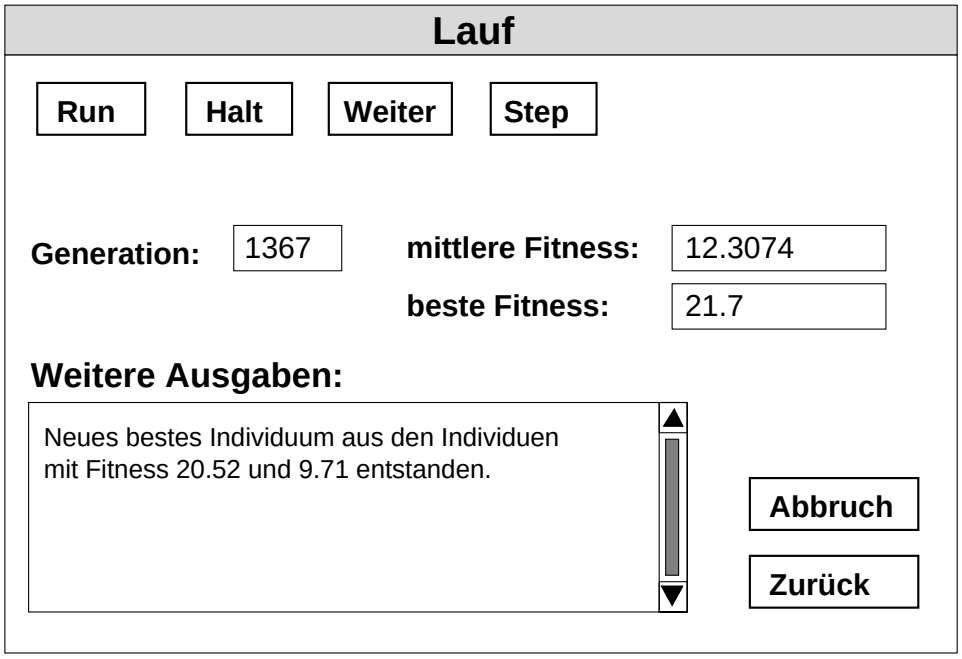


Abbildung 5.10: Fenster „Lauf“

ii. *halt*
Keine Eingabe. Die Funktion ist nur aktiv, wenn eine Simulation läuft.

iii. *weiter*
Keine Eingabe. Die Funktion ist nur aktiv, wenn eine Simulation läuft.

iv. *step*
Keine Eingabe. Die Funktion ist nur aktiv, wenn keine Simulation läuft.

v. *Abbruch*
Keine Eingabe. Die Funktion ist nur aktiv, wenn eine Simulation läuft.

vi. *zurück*
Keine Eingabe.Die Funktion ist nur aktiv, wenn keine Simulation läuft.

(c) Vorgangsbeschreibung:

i. *run*
Der EA wird abgearbeitet, bis der EA regulär beendet wird, ein aktives Abbruch-Label erreicht wird oder der Benutzer in den Ablauf eingreift. Während des Laufs werden die Ausgabedaten regelmäßig aktualisiert.

ii. *halt*
Die Abarbeitung des EA wird am nächsten aktiven Halte-Label angehalten.

iii. *weiter*
Die Abarbeitung des EA wird vom aktuellen Halte-Label aus fortgesetzt.

iv. *step*
Die Abarbeitung des EA wird bis zum nächsten aktiven Halte-Label fortgesetzt.

v. *Abbruch*
Der EA wird sofort unterbrochen.

vi. *zurück*
Die Endpopulation und alle für einen Wiederhollauf notwendigen Daten werden in die log–Datei geschrieben.

(d) Ausgabe:

i. *run*
Während des Laufs ist immer ein Ausgabefenster zu sehen, in dem der Generationenzähler, die durchschnittliche Fitneß der Population und die Fitneß des besten Individuums anzeigt werden. Die Werte werden einmal pro Generation aktualisiert.

ii. *halt*
Es ist das Fenster „Lauf“ zu sehen.

iii. *weiter*
Es ist das Fenster „Lauf“ zu sehen.

iv. *step*
Falls eine Generation beendet wurde werden die Ausgaben wie bei der Funktion *run* modifiziert. Es ist das Fenster „Lauf“ zu sehen.

v. *Abbruch*
Es erscheint eine Warnung, daß die Population u.U. in einem inkonsistenten Zustand ist. Der Benutzer hat die Wahl, ob er abbricht — dann hat er auch keine Ergebnisdaten zur Verfügung —, oder ob er bis zu einem Halte–Label weiterlaufen möchte.

vi. *zurück*
Es erscheint das Fenster „Experiment“.

5.3.2 externe Schnittstellenanforderungen

- Benutzerschnittstelle:
Die Benutzerschnittstelle wird durch eine **G**rafical **U**ser **I**nterface (GUI) realisiert. Das Programm ist mausgesteuert und nur die Eingabe von Bezeichnern und Zahlenwerten erfordert die Tastatur.
Hilfefunktionen und Funktionstasten sind nicht vorgesehen.
Für weitere Details sei auf die Konventionen des X-Windows-Systems verwiesen.

- Softwareschnittstellen:
  verwendete Systemsoftware:

  Betriebssystem: Solaris2.3 (SunOS5.3 & Zubehör)

  Oberfläche: X11 Window-System Release 5

  Compiler: SPARCompiler C++ 4.0

- verwendete Anwendungssoftware bzw. Tools:

  Interpreter für LEA: Zur Abarbeitung der in der Programmiersprache LEA spezifierten EAs wird in der PGA ein Interpreter entwickelt. Zur Beschreibung der Schnittstellen wird auf 6 verwiesen.

  Oberfläche - GUI für EAGLE: Die für EAGLE verwendete GUI wird ebenfalls in der PGA entwickelt. Hierfür existiert aber noch keine SchnittstellenBeschreibung.

5.3.3 Leistungsanforderung

Der Aspekt der Leistungsanforderung in bezug auf Laufzeit spielt für dieses Softwareprojekt eine untergeordnete Rolle.
Die Laufzeit eines in LEA formulierten EA darf auf EAGLE höchstens um den Faktor 100 langsamer sein, als die Ausführung des Codes des EA, der analog (unoptimiert) zu der Beschreibung des LEA-Codes übertragen wird in C++.

5.3.4 Designeinschränkungen

Da das Design objektorientiert sein soll, wurde C++ als Programmiersprache bestimmt. Die Entscheidung für eine bestimmte Programmiersprache fiel in der PGA bevor die Designphase begann, um den Projektmitgliedern die Zeit zu geben, mit dieser Programmiersprache vertraut zu werden.

Kapitel 6

LEA — Language for Evolutionary Algorithms

6.1 Motivation

Laut Pflichtenheft (Abschnitt 4) und Anforderungsspezifikation (Abschnitt 5) soll es möglich sein, in EAGLE völlig beliebige, verschiedene Evolutionäre Algorithmen (EA) zu untersuchen, weiterzuentwickeln und auch neue Algorithmen zu entwerfen. Daher war es notwendig, eine Möglichkeit zu entwickeln, mit der Evolutionäre Algorithmen auf einfache Art und Weise eingegeben werden können. Durch ein „bausteinartiges“ Prinzip sollten Teile aus einem Evolutionären Algorithmus in einem zweiten Algorithmus wiederverwendbar sein — auch wenn dort ein anderes zu optimierendes Problem mit einer andersgearteten Repräsentation und Kodierung vorliegt.

Daher wurde eine Entscheidung für eine getrennte Eingabe des Problems (mit der Problemstruktur und der Fitneßfunktion), der Kodierung und des darauf arbeitenden Evolutionären Algorithmus gefällt. Die Eingabe eines Algorithmus sollte aufgrund des „Baustein“-Denkens weiter aufgespaltet werden in die Eingabe der einzelnen Operatoren. Dabei wird ein gesamter Algorithmus als ein Operator aufgefaßt, der als Eingabe eine Population erhält und diese durch Aufruf weiterer Operatoren verändert, um eine optimierte Lösung des Problems zu finden. Die einzelnen Algorithmenteile sollten in einer Sprache formuliert werden können und dann von einem Simulator abgearbeitet werden.

Die Sprache LEA (*Language for Evolutionary Algorithms*) wurde in der Projektgruppe entwickelt, um die Eingabe von verschiedenen Operatoren zu ermöglichen. Während des Entwicklungsprozesses wurde die Syntax und die Semantik dieser Sprache stark von der geplanten Realisierung des Simulators für LEA beeinflußt. Dies ist auch im letzten Abschnitt dieses Unterkapitels näher erläutert.

Wir haben uns bei unserem Sprachentwurf dafür entschieden, daß sich LEA stark an

den Programmiersprachen Pascal [JW74] und Modula–2 [Wir82] orientiert. Daher wird für das Verständnis einiger der folgenden Abschnitte auch die Kenntnis einer solchen Programmiersprache vorausgesetzt.

In den folgenden Abschnitten werden die Idee der Operatoren und die Sprachelemente von LEA vorgestellt. In Abschnitt 6.17 wird kurz die Realisierung der Sprache LEA betrachtet, da sie zu diesem Sprachentwurf geführt hat.

Die Syntax von LEA kann Anhang C entnommen werden.

6.2 Die Idee hinter LEA und das Zusammenspiel mit EAGLE

Bevor auf die genauere Syntax und Semantik von LEA eingegangen wird, soll in diesem Abschnitt noch ein kurzer Überblick über die zugrundeliegenden Konzepte und das Zusammenwirken von LEA und EAGLEeingeschoben werden, da diese Grundlagen das Verständnis von LEA erleichtern.

In EAGLEwird bezüglich der Eingabe unterschieden zwischen den Teilbereichen Problem, Kodierung und Operatoren.

Das Problem besteht im wesentlichen aus einer Fitneßfunktion und der Repräsentation. Letztere ist eine (ungeordnete) Menge von Datentypen, welche im Individuum bestimmte Werte annehmen können. Auf der Repräsentation arbeitet die Fitneßfunktion. Außerdem ist noch eine Funktion zur Überprüfung spezieller Beschränkungen für die Individuen und eine Funktion zur effizienten Erzeugung von korrekten Individuen optional möglich.

Die Kodierung ist eine „maschinennähere“ Sichtweise der Repräsentation. Auf ihr arbeiten im Normalfall die Evolutionären Algorithmen.

Die Algorithmen werden als Operatoren eingegeben. Allerdings sollen Operatoren so allgemein formuliert werden können, daß sie auf verschiedene Probleme, Repräsentationen und Kodierungen anwendbar sind.

Wenn im weiteren von Operatoren die Rede ist, sind damit nicht nur die eigentlichen Operatoren zur Eingabe eines Evolutionären Algorithmus gemeint, sondern auch die Fitneßfunktion und die weiteren möglichen Funktionen bei der Problemeingabe.

Um ein Experiment ablaufen zu lassen, ist es notwendig, die Experimentdefinition aus den folgenden Teilen zusammenzustellen:

- einem Problem, bestehend aus Fitneßfunktion und Repräsentation
- einer Kodierung

- einem Operator, der den Evolutionären Algorithmus darstellt

Solche Operatoren, die eine Population als Argument bekommen und das beste Individuum zurückliefern, werden im weiteren auch *Hauptoperatoren* genannt. Der Hauptoperator kann noch verschiedene andere Operatoren aufrufen, d. h. es wird ein Algorithmus aus verschiedenen Operatoren „zusammengestöpselt". Dadurch ist es möglich die Operatoren auch noch in anderen EA weiterzuverwenden. Die Funktionsweise dieses „Baukasten"-Prinzips ist in der Abbildung 6.1 veranschaulicht.

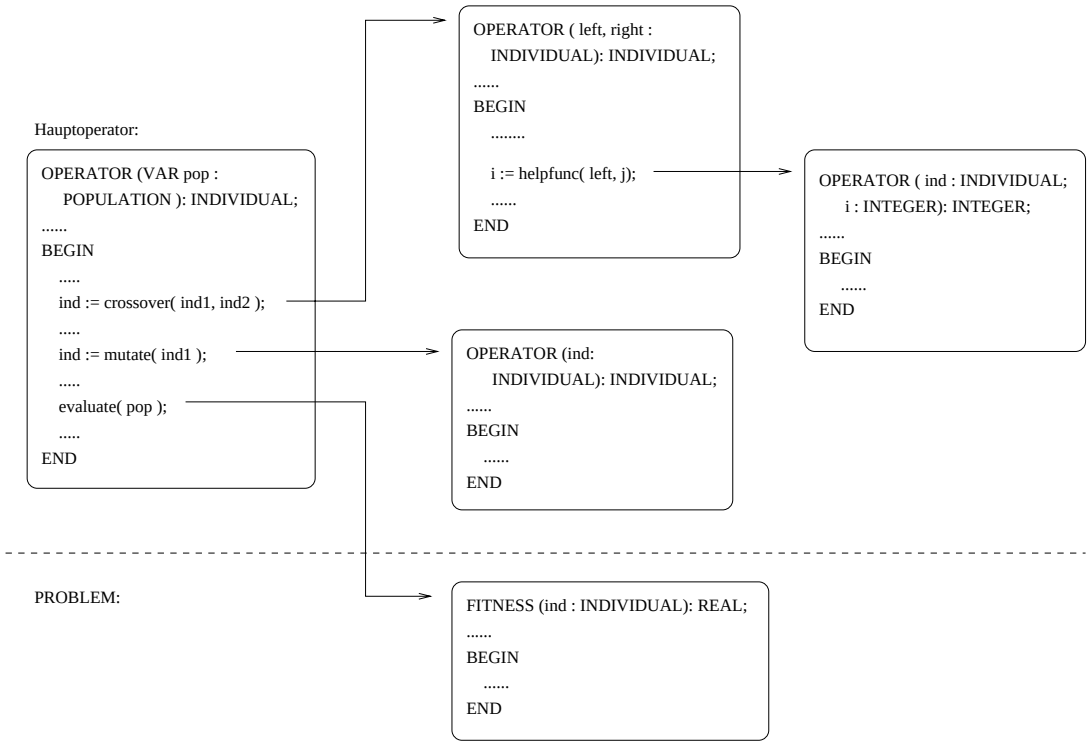


Abbildung 6.1: Beispielhierarchie von Operatoren

Bevor ein solches Experiment ablaufen kann, sind noch verschiedene Eingaben in der Laufinitialisierung notwendig. Es handelt sich dabei vor allem um Eingaben und Wahlmöglichkeiten, die im LEA-Text des Hauptoperators und den verwendeten Unteroperatoren noch offengelassen wurden:

Parameter In LEA können verschiedene Parameter deklariert werden. Es handelt sich dabei um Konstanten für diesen Operator, deren Wert im Operatortext noch offen gelassen wird. Die Parameter können dann außerhalb des Operators vor jedem einzelnen Experiment in der Laufinitialisierung mit Werten belegt

werden. Dafür bietet sich z. B. die Crossover-Wahrscheinlichkeit an. Es handelt sich also vor allem um Einstellungen, die für einzelne Experimente schnell verändert werden sollen, ohne den gesamten Operatortext abzuändern.

Filter In LEA kann an beliebigen Stellen im Operatortext angegeben werden, welche Variablen dort gefiltert werden können. Diese Filter können dann vor einem Experiment aktiviert werden. Für die aktivierten Filter wird außerdem angegeben, in welche Dateien die Speicherung der Daten erfolgen soll.

Label In LEA können durch Labels bestimmte Stellen markiert werden, an denen dann in der Laufinitialisierung Abbruch- oder Haltepunkte gesetzt werden. An diesen Punkten hält die Simulation des Experiments immer bzw. nur bei Benutzerabbruch an.

6.3 Datenstrukturen

Um ein schnelles Arbeiten mit LEA zu ermöglichen, ist es notwendig, daß neben den normalen Standarddatentypen wie Integer und Real auch spezifische evolutionäre Datentypen wie Individuum und Population zur Verfügung stehen. Diese können durch einfache Funktionen (bzw. Methoden) manipuliert und verändert werden. Auf die Einführung von komplizierten benutzerdefinierten „Structs“ oder Records wurde verzichtet, da dies u. U. den Umfang der anstehenden Arbeit gesprengt hätte. Allerdings wurde zur Betrachtung von speziellen Permutationsproblemen auch ein Datentyp Permutation eingeführt. Auch eine geordnete Ansammlung von Datenstrukturen identischen Typs in einem Feld („Array“) wurde ermöglicht.

Damit ergeben sich die folgenden Datentypen:

INTEGER umfaßt die ganzen Zahlen zwischen einer minimalen und einer maximalen Zahl. Dieser Bereich ist maschinenabhängig.

REAL umfaßt die reellen Zahlen zwischen einer minimalen und einer maximalen Zahl. Der Bereich und die Darstellungsgenauigkeit sind maschinenabhängig.

BIT kann die Werte 0 und 1 annehmen

PERMUTATION hat als Wert eine Permutation der Werte von 1 bis zu einem festen Wert größer als 1. Die Größe der Permutation wird bei der Deklaration angegeben und kann nicht geändert werden. Auf die einzelnen Werte der Permutation kann mittels der im Abschnitt 6.10 beschriebenen Funktionen zugegriffen werden.

INDIVIDUAL hat als Wert ein komplettes Individuum. Ein Individuum besteht aus seiner Kodierung, den dadurch dargestellten Werten in der Repräsentation

und der zuletzt berechneten Fitneß. Auf die einzelnen Werte der Kodierung bzw. der Repräsentation und auf die Fitneß kann mittels der Funktionen im Abschnitt 6.8 zugegriffen werden. Das Aussehen eines Individuums wird nicht in einem Operatortext eindeutig festgelegt, sondern wird erst bei der Simulation eines Ablaufes durch das Problem, mit dem der definierte Operator benutzt wird, und seiner Kodierung bestimmt.

POPULATION hat als Wert eine komplette Population. Die Population ist eine (geordnete) Ansammlung von Individuen, d. h. Daten vom Typ `INDIVIDUAL`. Die Größe der Population ist dynamisch. Auf die einzelnen Individuen und weitere Informationen über die Population kann mittels der Funktionen im Abschnitt 6.9 zugegriffen werden.

ARRAY ist ein ein- oder mehrdimensionales Feld mit Werten eines festen Datentyps aus `INTEGER`, `REAL`, `BIT`, `INDIVIDUAL`, `POPULATION` und `PERMUTATION`. Die Dimensionen sind fest und können nicht nachträglich geändert werden. Schachtelung von `ARRAY`s ist nicht möglich, d. h. mehrdimensionale Felder sollten durch direkte Angabe aller Dimensionen definiert werden.

Analog zu Pascal kann auf die einzelnen Einträge des Feldes in Zuweisungen und Ausdrücken (siehe auch Abschnitt 6.7) mittels der Angabe des Index in rechteckigen Klammern zugegriffen werden.

6.4 Arten von Operatoren

In LEA können verschiedene Arten von Operatoren definiert werden. In der Kopfzeile jedes Operators wird festgelegt, um welchen Operatortyp es sich handelt, welche Aufrufparameter nötig sind und welchen Rückgabetyp der Operator hat. Aufrufparameter können auf zwei verschiedene Arten übergeben werden: einmal per default als call-by-value, d. h. der Wert wird kopiert und ändert sich nicht außerhalb des aufgerufenen Operators, oder als call-by-reference, d. h. es wird mit dem Wert außerhalb des Operators gearbeitet, und dieser ändert sich auch z. B. bei Zuweisungen. Für einen call-by-reference-Parameter wird das Schlüsselwort `VAR` vor den betreffenden Aufrufparameter gestellt.

Eine Kopfzeile sieht z. B. folgendermaßen aus:

```
OPERATOR (laenge,durchmesser: INTEGER; VAR pop: POPULATION):INDIVIDUAL;
```

Es gibt die folgenden, verschiedenen Operatortypen:

FITNESSOP Fitneßoperatoren bekommen ein Individuum übergeben, berechnen die Fitneß dieses Individuums auf der Grundlage der Repräsentation und liefern den berechneten Wert als `REAL`-Wert zurück. In Fitneßoperatoren darf

nicht auf kodierte Werte des Individuums zugegriffen werden. Bei der Eingabe eines Problems in EAGLEdürfen nur Operatoren vom Typ `FITNESSOP` als Fitneßfunktionen benutzt werden.

PROCEDURE Prozeduren sind Operatoren, die keine evolutionären Datenstrukturen als Ein- oder Ausgabe bekommen, d. h. sie hängen nicht vom konkreten Problem und der Kodierung ab, in dessen Kontext sie verwendet werden.

OPERATOR Hier sind nur Operatoren im eigentlichen evolutionären Sinn gemeint, also Operatoren, die als Ein- oder Ausgabe u. a. Individuen bzw. Populationen haben.

MAIN Hauptoperatoren sind diejenigen Operatoren, die ein gesamtes Evolutionäres Verfahren darstellen. Sie erhalten als Eingabe eine Population als `VAR`-Parameter und liefern ein — das beste — Individuum am Ende der Laufzeit zurück.

Zusätzlich gibt es noch die Operatorentypen `CHECKOP` zur Überprüfung von Konsistenzbedingungen und `CREATEIND` zur Erzeugung korrekter Individuen. Sie finden im derzeitigen System EAGLEallerdings keine Berücksichtigung.

6.5 Variablen, Konstanten, Parameter

Direkt nach der Kopfzeile eines Operators können Variablen, Konstanten und Parameter deklariert werden. Für alle drei gilt, daß sie ausschließlich in dem Operator verwendet werden, in dem sie deklariert sind — also auch nicht in aufgerufenen Unteroperatoren.

Variablen Variablen können von beliebigem Datentyp sein. Sie werden mit dem Schlüsselwort `VAR` definiert. Es wird zwischen zwei verschiedenen Variablenarten unterschieden: normalen Variablen und statischen Variablen.

Normale Variablen sind bei jedem Aufruf des Operators wieder mit dem Nullwert (also 0 bei Zahlen, Identität bei Permutationen, leere Population) bzw. dem Default-Wert vorbelegt. D. h. der Wert der Variable ist nicht von früheren Aufrufen des Operators abhängig. Falls ein Operator sich selbst rekursiv aufruft, haben die Variablen gleichen Namens in den verschiedenen Instanziierungen des Operators keinerlei Einfluß aufeinander.

Zum anderen gibt es statische Variablen. Hier wird nur eine Variable angelegt, die für alle Operatoraufrufe dieselbe ist und ihren Wert beibehält. Dies bedeutet für rekursive Aufrufe eines solchen Operators, daß sich der Variablenwert in allen Instanzen ändert, wenn im innersten Aufruf der Rekursion eine Zuweisung stattfindet. Statische Variablen werden durch das Schlüsselwort `STATIC` vor dem Namen der Variable gekennzeichnet.

Optional kann bei allen Variablen ein Initialisierungswert angegeben werden.

Die Definitionen lauten z. B.:

```
VAR individual1 : INDIVIDUAL;
    temp = 15, STATIC counter : INTEGER;
```

Konstanten Konstanten gibt es nur für direkte Zahlenwerte. Sie werden mit dem Schlüsselwort `CONST` definiert, wobei immer dem Namen der Konstante der Zahlenwert mit einem Gleichheitszeichen zugewiesen wird.

Beispiel:

```
CONST pi = 3.1416;
      e = 2.7183;
```

Parameter Parameter sind Konstanten, deren Wert noch offengelassen wird. Sie werden außerhalb von LEA in der Laufinitialisierung direkt vor einem Experiment mit festen Werten belegt werden. Dieses Prinzip bietet sich an für Werte, die man gerne schnell für verschiedene Experimente ändern möchte, ohne den gesamten Operator zu ändern (also z. B. bei einer Mutationswahrscheinlichkeit). Parameter werden direkt hinter dem Schlüsselwort `PARAMETER` aufgelistet. Sie erhalten neben dem Typ einen Default-Wert. Hier können als Parametertyp auch eingeschränkte Wertebereiche angegeben werden.

Beispiel:

```
PARAMETER crossover    = 0.5 : REAL[0.0 .. 1.0];
          optimization = 1   : BIT;
```

6.6 Ablaufkontrollkonstrukte

Der eigentliche Ablauf des Operators wird nach den Deklarationen angegeben. Er wird durch `BEGIN` und `END` umklammert. In diesem Teil des Operators werden einzelne Befehle durch ; voneinander getrennt.

Um einen Ablauf einzugeben, stehen die folgenden Sprachkonstrukte zur Verfügung:

IF-THEN-ELSE Syntax und Semantik der `IF-THEN-ELSE`-Verzweigung entsprechen genau der Verwendung in Modula-2.

Beispiel:

```
IF counter = 0 THEN counter := 100
               ELSE counter := counter - 1 END;
```

WHILE-Schleife Syntax und Semantik der `WHILE`-Schleife entsprechen genau der Verwendung in Modula-2.

Beispiel:

```
WHILE counter < 100 DO counter := counter + 1 END;
```

FOR-Schleife Syntax und Semantik der `FOR`-Schleife entsprechen genau der Verwendung in Modula-2.

Beispiel:

```
FOR i := 1 TO n DO mutation_nr_10( individual1, i ) END;
```

dabei ist `mutation_nr_10` ein benutzerdefinierter Mutationsoperator (siehe auch Abschnitt 6.11).

FOREACH-Schleife Hiermit können eine (oder auch mehrere) Aktionen auf alle Individuen in einer Population angewendet werden. Dabei steht direkt hinter dem `FOREACH` eine Variable vom Typ `INDIVIDUAL`, die nacheinander als Wert alle Individuen in der Population, die nach dem `IN` steht, annimmt. Die auszuführenden Aktionen stehen direkt nach dem `DO`. Das Individuum ist exakt das Individuum, das in der Population liegt. Es wird also nicht auf einer Kopie gearbeitet, sondern etwaige Änderungen werden direkt am Individuum in der Population vorgenommen.

Beispiel:

```
FOREACH individual1 IN population5
     DO mutation_nr_5( individual1 ) END;
```

RETURN Hiermit wird die Abarbeitung des Operators beendet, und der Wert (bzw. der Wert der Variable) direkt hinter dem `RETURN` zurückgegeben.

Beispiel:

```
RETURN individual1;
```

ERROR Mit dem Schlüsselwort `ERROR` kann die Interpretation des Operators mit einem Laufzeitfehler abgebrochen werden.

6.7 Wertzuweisungen und Ausdrücke

Die Zuweisung eines Werts an eine Variable geschieht analog zu Pascal mit dem Zuweisungsoperator `:=`, wobei links der Variablenname und rechts der zuzuweisende Ausdruck steht.

Beispiel:

```
integer1 := 2 * integer2;
feld[1,1,5] := feld[1,1,4];
```

LEA unterstützt über die von Pascal her bekannten Zuweisungen hinaus auch Zuweisungen an Variablen der Typen `INDIVIDUAL`, `POPULATION` und `PERMUTATION`. Dabei wird z. B. im Falle eines Individuums der Inhalt des Individuums auf der rechten Seite in das Individuum auf der linken Seite kopiert.

Ein zuzuweisender Ausdruck kann aus den folgenden Teilen aufgebaut werden:

- `INTEGER`-, `REAL`- und `BIT`-Zahlen
- Variablen mit beliebigem Typ
- Konstanten
- Funktionen zum Zugriff auf Werte von Permutation (siehe Abschnitt 6.10)
- Funktionen zum Zugriff auf Individuen bzw. Populationen (siehe Abschnitte 6.8 und 6.9)
- Funktionen zur Erzeugung von Zufallsvariablen (siehe Abschnitt 6.15)
- Aufruf von anderen benutzerdefinerten Operatoren (siehe Abschnitt 6.11)
- Vergleichsoperatoren `=`, `<>`, `<`, `>`, `<=` und `>=`
- Additionsoperatoren `+`, `-` und (falls es sich um binäre Ausdrücke handelt) `OR`
- Multiplikationsoperatoren `*`, `/` und (falls es sich um binäre Ausdrücke handelt) `AND`
- Funktionen zur Typkonvertierung (siehe unten)

Mit den Vergleichs-, den Additions- und den Multiplikationsoperatoren können keine Daten der Typen `PERMUTATION`, `INDIVIDUAL`, `POPULATION` oder `ARRAY` miteinander verglichen bzw. verknüpft werden.

Zwischen den einzelnen Typen sind explizite Typkonvertierungen notwendig, d. h. LEA konvertiert nicht automatisch zwischen den verschiedenen Datentypen. Konvertierungen sind zwischen `INTEGER`, `REAL` und `BIT` möglich. Der Datentyp in den der Ausdruck konvertiert werden soll, wird vor den geklammerten Ausdruck gesetzt. Dabei gelten die folgenden Vereinbarungen:

- Bei einer Konvertierung nach `BIT` wird genau dann 0 zurückgegeben, wenn der Ausdruck 0 war. Andernfalls ist der Rückgabewert 1.
- Bei einer Konvertierung nach `REAL` wird der Ausdruck als reelle Zahl zurückgegeben.
- Bei einer Konvertierung nach `INTEGER` werden reelle Zahlen abgerundet.

Beispiel:

```
bit1 := BIT(0.0);
bit2 := BIT(17);
integer1 := INTEGER(14.9);
real1 := REAL(23);
```

Die Variable `bit1` hat den Wert 0, `bit2` den Wert 1, `integer1` den Wert 14 und `real1` den Wert 23.0 .

6.8 Manipulation von Individuen

Da auf den Inhalt von Individuen nicht in direkter Form zugegriffen werden kann und der Operatortext unter bestimmten Umständen von der Struktur des Individuums unabhängig sein soll, werden mehrere Funktionen zur Verfügung gestellt, die es ermöglichen, sowohl Informationen über die Struktur als auch über die Wertebelegung des Individuums zu erhalten. Es wird zwischen der problemorientierten Repräsentation und der Kodierung unterschieden. In einem Individuum werden die Werte der Repräsentation und der Kodierung immer automatisch konsistent gehalten.

Die Funktionen gliedern sich folgendermaßen:

Informationen über die Struktur der Repräsentation

Da die Struktur der Repräsentation bei allen Individuen in einem Operator gleich ist, wird bei diesen Funktionen auf die Übergabe eines Individuums verzichtet.

Die folgenden Informationen können abgefragt werden:

- Länge der Repräsentation

  ```
  replength()
  ```

- Anzahl der direkt aufeinanderfolgenden Atome mit identischem Typ ab einer bestimmten Position. Liegt an der Position kein Atom vom gefragten Typ vor, wird der Wert 0 zurückgegeben.

  ```
  numberrepbit( position )
  numberrepint( position )
  numberrepreal( position )
  numberrepperm( position )
  ```

Informationen über die Struktur der Kodierung

Ganz analog können auch Informationen über die Kodierung der Repräsentation abgefragt werden. Auch hier wird kein Individuum übergeben.

Folgende Informationen stehen zur Verfügung:

- Länge der Kodierung

  ```
  length()
  ```

- Anzahl der direkt aufeinanderfolgenden Atome mit identischem Typ ab einer bestimmten Position. Liegt an dieser Position kein Atom vom gefragten Typ vor, wird der Wert 0 zurückgegeben.

  ```
  numberbit( position )
  numberint( position )
  numberreal( position )
  numberperm( position )
  ```

Lesen und Setzen der Werte der Repräsentation eines Individuums

Hier wird bei allen Funktionen auf eine bestimmte Position in einem Individuum lesend oder schreibend zugegriffen. D. h. sowohl das betroffene Individuum als auch die Position müssen übergeben werden. Bei einem Schreibzugriff wird zusätzlich auch der neue Wert übergeben. Falls an der entsprechenden Stelle kein Atom des gewünschten Typs ist, bricht der Simulator mit einem Laufzeitfehler ab.

Folgende Funktionen sind vorhanden:

- Lesen eines Werts. Der Wert wird als Resultat der Funktion zurückgegeben.

  ```
  getrepbit( individual1, position )
  getrepint( individual1, position )
  getrepreal( individual1, position )
  getrepperm( individual1, position )
  ```

- Setzen eines Werts

```
setrepbit( individual1, position, wert )
setrepint( individual1, position, wert )
setrepreal( individual1, position, wert )
setrepperm( individual1, position, wert )
```

Lesen und Setzen der Werte der Kodierung eines Individuums

Der Zugriff auf die Werte der Kodierung funktioniert analog zum Zugriff auf die Werte der Repräsentation. Unter dem Wert einer Permutation verstehen wir dabei den Inhalt einer Variablen des Typs Permutation.

Die Funktionen sind im einzelnen:

- Lesen eines Werts. Der Wert wird als Ergebnis der Funktion zurückgegeben.

```
getbit( individual1, position )
getint( individual1, position )
getreal( individual1, position )
getperm( individual1, position )
```

- Setzen eines Werts

```
setbit( individual1, position, wert )
setint( individual1, position, wert )
setreal( individual1, position, wert )
setperm( individual1, position, wert )
```

Berechnung und Zugriff auf die Fitneß

Die Fitneß eines Individuums wird nur dann berechnet, wenn dies explizit verlangt wird. Dann wird der Fitneßoperator mit dem übergebenen Individuum ausgeführt. Der Befehl lautet

```
evaluate( individual1 )
```

Dabei wird kein Rückgabewert zurückgegeben.

Zum Zugriff auf die zuletzt berechnete Fitneß steht die folgende Funktion zur Verfügung:

```
fitness( individual1 )
```

6.9 Manipulation von Populationen

Auch auf den Inhalt von Populationen kann nicht in direkter Form zugegriffen werden. Daher stehen mehrere Funktionen zur Verfügung, die es ermöglichen, sowohl Informationen über die Population als auch über die einzelnen in ihr enthaltenen Individuen zu erhalten. Dies sind genauer die folgenden Funktionen:

Informationen über die Population

Die folgenden Daten können von einer Population abgefragt werden:

- Größe der Population, d. h. die Anzahl der in ihr enthaltenen Individuen

  ```
  sizeofpop( population1 )
  ```

- Verschiedene Fitneßwerte: durchschnittliche, beste und schlechteste in der Population vorkommende Fitneß.

  ```
  getavgfitness( population1 )
  getbestfitness( population1 )
  getworstfitness( population1 )
  ```

Lesen, Löschen und Hinzufügen von Individuen

Mit folgenden Funktionen können die Individuen gelesen bzw. geschrieben werden:

- Löschen der Population
  Alle Individuen in der Population werden gelöscht, die Population ist anschließend leer.

  ```
  clearpop( population1 )
  ```

- Vereinigen zweier Populationen
  Alle Individuen aus der Quellenpopulation werden in die Zielpopulation kopiert.

  ```
  mergepop( destinationpopulation, sourcepopulation )
  ```

- Lesen eines Individuums aus der Population
  Es wird eine Kopie des Individuums mit der entsprechenden Nummer zurückgegeben.

  ```
  getind( population1, number )
  ```

- Einfügen eines Individuums in die Population
  Das neue Individuum wird an der Stelle mit der entsprechenden Nummer in der Population eingefügt. Die Numerierung der nachfolgenden Individuen verschiebt sich dabei.

  ```
  insertind( population1, individual1, number )
  ```

- Löschen eines Individuums aus der Population
  Das entsprechende Individuum wird gelöscht, die Numerierung der nachfolgenden Individuen verschiebt sich.

  ```
  killinpop( population, number )
  ```

- Anfrage nach einem besten Individuum
  Es wird die Nummer eines Individuums mit der besten Fitneß zurückgegeben.

  ```
  getbest( population )
  ```

- Anfrage nach einem schlechtesten Individuum
  Es wird die Nummer eines Individuums mit der schlechtesten Fitneß zurückgegeben.

  ```
  getworst( population )
  ```

Berechnen der Fitneß

Analog zu der Berechnung der Fitneß eines einzelnen Individuums kann auch die Fitneß zu jedem Individuum in einer Population berechnet werden. Die Funktion lautet genau gleich und wird nur mit einer Population als Argument aufgerufen:

```
evaluate( population1 )
```

6.10 Manipulation von Permutationen

Auch auf den Inhalt von Permutationen kann nicht in direkter Form zugegriffen werden. Daher stehen mehrere Funktionen zur Verfügung, die es ermöglichen, die Werte der Permutation zu lesen und zu verändern. Die Funktionen sind die folgenden:

Lesen eines Wertes der Permutation

Es wird der Wert, der einer bestimmten Stelle der Permutation zugeordnet ist, zurückgeliefert. Falls die Stelle nicht im erlaubten Bereich liegt, wird eine 0 zurückgegeben.

```
getpermvalue( permutation1, position )
```

Setzen eines Wertes der Permutation

Der Wert der Permutation wird an der gewünschten Stelle auf den angegebenen Wert gesetzt. Dabei werden die verbleibenden Werte in der Permutation so verschoben, daß die Permutation in einem konsistenten Zustand bleibt. Der Rückgabewert der Funktion ist 1, falls sie erfolgreich war, andernfalls 0 (z. B. wenn der Wert oder die Stelle nicht im erlaubten Bereich liegen).

```
setpermvalue( permutation1, position, value )
```

Falls z. B. die Permutation vor dem Aufruf die Belegung `( 4 2 1 3 )` hatte, ändert sich ihre Wertebelegung bei einem Aufruf mit den Werten `3` für `position` und `4` für `value` zu `( 2 1 4 3 )`.

Vertauschen zweier Werte

Es werden beim Aufruf der Funktion zwei Stellen angegeben, an denen die Werte in der Permutation vertauscht werden. Auch hier wird wieder 1 im Erfolgsfall und 0 sonst zurückgegeben.

```
xchangeperm( permutation1, position1, position2 )
```

Falls der Aufruf mit denselben Werten wie bei `setpermvalue` erfolgt, hat die Permutation anschließend die Wertebelegung `( 1 2 4 3 )`.

Inversion eines Teils der Permutation

Es wird das Stück zwischen zwei Positionen einer Permutation gespiegelt. Hierfür werden die erste und die letzte Position des zu spiegelnden Teilstücks angegeben. Rückgabewert ist 1 im Erfolgsfall und 0 sonst.

```
reverseperm( permutation1, firstposition, lastposition )
```

Mit der anfänglichen Belegung `( 4 2 1 3 )` und einem Aufruf mit den Positionen `2` und `4` ändert sich die Permutation zu `( 4 3 1 2 )`.

6.11 Verwendung anderer Operatoren

Wenn in einem Operator andere benutzerdefinierte Operatoren verwendet werden sollen, ist es notwendig, daß dem Operator zunächst mitgeteilt wird, welche Operatoren benötigt werden. Dann kann der Unteroperator auf verschiedene Arten aufgerufen werden. Jeder Operator kann sich selbst rekursiv ohne vorherige Deklaration aufrufen.

Operatoren werden im Deklarationsteil, also vor dem `BEGIN`, deklariert. Es reicht anzugeben, welche Operatoren hier im Algorithmenteil aufgerufen werden. Dies geschieht durch das Schlüsselwort `IMPORT` und eine anschließende Aufzählung der Operatoren.

Beispiel:

```
IMPORT mutation1, crossover3;
```

Dann können die Operatoren im Operatortext beliebig verwendet werden. Beim Aufruf eines Operators werden die Argumente für diesen Operator in Klammern hinter dem Operatornamen angegeben. Also z. B.:

```
individual1 := crossover3( individual2, individual1 );
```

Die Argumente werden als reference-by-value- oder reference-by-argument-Parameter dem Operator übergeben, so wie es in seiner Kopfzeile festgelegt wurde (siehe Abschnitt 6.4).

Wenn Individuen als Argument übergeben werden, ist in LEA allerdings ein weiterer, besonderer Übergabemechanismus möglich. Es kann sein, daß der aufgerufene Operator nur auf einem Teil der Individuen arbeiten soll und auch nur für ein Individuum geschrieben ist, bei dem der restliche Teil fehlt. Dann können beim Aufruf des Operators die überflüssigen Teile ausgeblendet werden, so daß der aufgerufene Operator darauf paßt.

Beispiel:

Das kodierte Individuum besteht aus fünf `REAL`-Werten, wobei nur drei Werte das eigentliche Individuum darstellen und zwei z. B. Strategieparameter sind. Dann kann ein Unteroperator `suboperator`, der nur auf dem Kodierungsteil — also drei `REAL`-Werten — arbeitet, folgendermaßen aufgerufen werden:

```
suboperator( individual1{ REAL(1 .. 3)} )
```

Hier werden das vierte und fünfte Element in der Kodierung ausgeblendet. Bei den verbleibenden drei Werten handelt es sich um `REAL`-Zahlen.

Durch Kommata getrennt werden die einzelnen Datentypen und in Klammern ihre Position(en) im ursprünglichen Individuum angegeben. Die Reihenfolge, in der sie angegeben werden, entspricht der Reihenfolge, in der der Unteroperator auf ihnen arbeitet.

6.12 Prinzip der Labels

Labels können fest im Operatortext gesetzt werden. Sie kennzeichnen Stellen, an denen Breakpoints usw. vor dem Start eines EA-Ablaufes gesetzt werden können. Ein Label besteht aus dem Schlüsselwort `LABEL` und dem Namen des Labels, also z. B.

```
LABEL firstlabel;
```

6.13 Prinzip der Filter

Filter sind Stellen, an denen Werte abgespeichert werden können. Grundsätzlich kann jede Variable in LEA gefiltert werden, also auch Individuen, ganze Populationen und Permutationen. Das eigentliche Format der Abspeicherung wird nicht hier in LEA festgelegt, sondern vor dem einzelnen Ablauf eines EA. Dort können die einzelnen Filter aktiv oder inaktiv gesetzt werden. Falls sie aktiv sind, kann angegeben werden, in welches File sie filtern sollen.

Filter bestehen aus dem Schlüsselwort `FILTER`, dem Namen des Filters, dem Typ des Filters (s. u.) und dem Wert, der gefiltert werden soll.

Der Typ des Filters gibt sowohl an, welcher Datentyp gefiltert werden soll, als auch bei Individuen und Populationen, in welcher Form dies geschehen soll. Die verschiedenen Typen sind:

INT Beim nachfolgenden Ausdruck handelt es sich um einen beliebigen `INTEGER`-Wert.

REAL Beim nachfolgenden Ausdruck handelt es sich um einen beliebigen `REAL`-Wert.

BIT Beim nachfolgenden Ausdruck handelt es sich um einen beliebigen `BIT`-Wert.

PERMUTATION Beim nachfolgenden Ausdruck handelt es sich um eine Permutation.

FITNESS Beim nachfolgenden Ausdruck handelt es sich um ein Individuum oder eine Population. Es wird der Fitneßwert des Individuums bzw. jedes Individuums in der Population abgespeichert.

INDIVIDUAL Beim nachfolgenden Ausdruck handelt es sich um ein Individuum oder eine Population. Es wird das komplette Individuum in der Darstellung der Repräsentation bzw. jedes Individuums in der Population in dieser Form abgespeichert.

CODEDIND Beim nachfolgenden Ausdruck handelt es sich um ein Individuum oder eine Population. Es wird das komplette Individuum in der Darstellung der Kodierung bzw. jedes Individuums in der Population in dieser Form abgespeichert.

Beispiel:

```
FILTER filter1 FITNESS individual1;
```

6.14 Ausgabe während des Laufs

Über ein spezielles Dialogfenster können während des Laufs eines EA Informationen auf dem Bildschirm ausgegeben werden.

Es stehen zur Ausgabe die Funktion `WRITE` zur Verfügung, wobei als Argument ein beliebiger Ausdruck oder ein Textstring in Anführungszeichen (") möglich sind.

Beispiele:

```
WRITE( "Geben Sie den Wert ein:" );
WRITE( real1 );
```

6.15 Von Zufallsgeneratoren und Generationenzählern

In LEA steht ein globaler Zufallszahlengenerator zur Verfügung. Der Seed des Generators kann nicht von LEA aus beeinflußt werden, sondern wird vor einem Ablauf eingegeben.

Folgende zwei Funktionen stehen zur Verfügung:

- reellwertige Zufallszahlen
  Mit der Funktion `getrandomreal` können reellwertige Zufallszahlen zwischen 0 und 1 (einschließlich) erzeugt werden.

  Beispiel:

  ```
  real1 := getrandomreal();
  ```

- ganzzahlige Zufallszahlen
  Mit der Funktion `getrandomint` können ganzzahlige Zufallszahlen zwischen (einschließlich) einer unteren und einer oberen Grenze erzeugt werden.

  Beispiel:

  ```
  integer1 := getrandomint( 1, 100 );
  ```

Außerdem steht noch ein globaler Generationenzähler in allen Operatoren zur Verfügung. Auf ihn kann lediglich lesend oder inkrementierend mit `gen` bzw. `incgen` zugegriffen werden.

Beispiel:

```
integer1 := gen();
incgen();
```

6.16 Ein Beispieloperator

Als Beispiel soll hier der LEA-Text für einen Genetischen Algorithmus vorgestellt werden. Der erste Operator, der angegeben wird, ist der Hauptoperator, der die Population erhält. Der zweite Operator ist der Crossover-Operator. Der Mutationsoperator, die Selektion und die (beliebige) Fitneßfunktion sind hier nicht angegeben.

Text des Hauptoperators:

```
OPERATOR (VAR pop : POPULATION):INDIVIDUAL;
VAR pop2       : POPULATION;
    ind, ind2  : INDIVIDUAL;
PARAMETER generationnumber = 2000 : INTEGER;
          crossoverrate = 0.3 : REAL;
          mutationrate =  0.05 : REAL;
IMPORT select, crossover, mutate;
BEGIN
  evaluate(pop);
  WHILE ((getbestfitness(pop) - getworstfitness(pop)) > 0.1) AND
        (gen() < generationnumber)
  DO
    pop := select(pop);
    FOREACH ind IN pop DO
      IF getrandomreal() <= crossoverrate THEN
        ind2 := crossover( ind,
                   getind(pop,getrandomint(1,sizeofpop(pop))));
      ELSE
        ind2 := ind;
      END
      insertind(pop2,ind2,1);
    END;
    FOREACH ind IN pop2 DO
      IF getrandomreal() <= mutationrate THEN
        ind := mutate( ind );
      END;
    END;
    pop := pop2;
    clear( pop2 );
    evaluate(pop);
  END;
  ind := getind( pop, getbest(pop) );
  RETURN ind;
END
```

Text des Operators „crossover“:

```
OPERATOR ( ind1, ind2 : INDIVIDUAL): INDIVIDUAL;
VAR ind : INDIVIDUAL;
    pos, counter : INTEGER;
BEGIN
  IF length() <> numberbit(1) THEN
    ERROR;
  ELSE
    pos := getrandomint(1,length());
    ind := ind1;
    FOR counter := pos TO length() DO
      setbit(ind, counter, getbit(ind2, counter));
    END;
    RETURN ind;
  END
END
```

6.17 Realisierung von LEA

Um die Eingabe und die spätere Abarbeitung von Operatoren in der Sprache LEA zu ermöglichen, wurden verschiedene Ansätze zur ihrer Implementation diskutiert. Je nachdem, welcher Ansatz für die Realisierung gewählt wird, ergeben sich verschiedene Konsequenzen bezüglich der Syntax und der Semantik der Sprache LEA. Daher sollen die verschiedenen Ansätze hier kurz vorgestellt werden.

Um eine Eingabesprache abzuarbeiten, bieten sich folgende Möglichkeiten an:

- Verwendung einer existierenden Hochsprache und eines existierenden Compilers
- Verwendung eines existierenden Simulators, der leicht angepaßt werden kann
- Entwicklung eines eigenständigen Simulators

Die Verwendung einer Hochsprache und eines externen Compilers hat den großen Vorteil, daß eine stabile und schnelle Abarbeitung des Codes in der Regel gewährleistet ist. Ein Problem liegt nun darin, daß der vom Benutzer geschriebene Code mit dem restlichen Code von EAGLEzur Laufzeit zusammenwirken soll. Dies ließe sich dadurch realisieren, indem verschiedene „Module“ zur Laufinitialisierung und zur interaktiven Kontrolle der Simulation entwickelt würden. Die von der Projektgruppe geforderte hohe Interaktivität, d. h. die direkte Eingabe eines Operators in

EAGLEund seine anschließenden sofortige Simulation, führt dabei zu dem diskutierten Compiler-Modell, d. h. die Überführung eines eingegebenen Operatortextes in übersetzbaren Source-Code einer Hochsprache, seine anschließende Übersetzung in ausführbaren Code, Zusammenbinden mit der Initialisierung und der Laufkontrolle von EAGLEund dem abschließenden Start des neu entstandenen Programms. Die Projektgruppe hatte sich gegen diese Realisation von LEA entschieden, da befürchtet wurde, daß der Zeitfaktor vom Zusammenstellen eines Experiments zum eigentlichen Experiment durch den Überführungsprozeß zu groß werden würde. Bei diesem Ansatz hätten sich Syntax und Semantik von LEA sehr stark an der verwendeten Hochsprache orientiert.

Die Verwendung eines existierenden Simulators hätte dieselben Vorteile wie bei dem vorigen Lösungsansatz. Zusätzlich würde auch die eher komplizierte und u. U. langsame Erstellung des lauffähigen Experiments entfallen. Hier bot es sich an, z. B. den Interpreter der Rapid-Prototyping-Sprache Tcl zu verwenden (siehe auch Abschnitt 9.1.2.2). Dieser Interpreter könnte leicht durch Definition neuer und Uminterpretation vorhandener Befehle in den gewünschten Simulator umgewandelt werden. Allerdings ergaben sich hier Probleme bei der Anbindung von `C++`-Code an den Tcl-Interpreter. Insbesondere wäre ein komplizierter Übergabemechanismus für die verwendeten Daten notwendig geworden. Bei dieser möglichen Realisation hätte sich LEA in Syntax und z. T. auch Semantik an Tcl angelehnt. Weitere existierende Interpreter oder Simulatoren wurden von uns nicht betrachtet.

Die Entscheidung fiel schließlich für den komplett selbstgeschriebenen Simulator, da dieser konkret für die verschiedenen verwendeten Datentypen entwickelt werden konnte. Dadurch ist ein schneller Zugriff auf die Daten gewährleistet. In der Entwicklung der Sprache LEA waren wir an keine Sprache gebunden, die zu verwenden notwendig gewesen wäre. Es konnten auch die neuen Konzepte wie die „eingeschränkte“ Übergabe von Individuen an Unteroperatoren (siehe Abschnitt 6.11) problemlos formuliert werden. Der Simulator setzt sich zusammen aus den Phasen der Erzeugung eines Strukturbaums für jeden verwendeten Operator und der späteren Simulation eines Experiments anhand dieses Strukturbaums.

Kapitel 7

Ein formaler Ansatz

In diesem Kapitel wird ein funktionaler Ansatz zur formalen Spezifikation des Systems EAGLE vorgestellt. Er entstand nach dem Scheitern der Implementierungsphase im Rahmen der Enddokumentation in einer Untergruppe. Es handelt sich dabei um einen Versuch, die komplette Funktionalität von EAGLE einschließlich LEA in eine mathematische Form zu bringen.

Dieses Kapitel gliedert sich wie folgt: In Abschnitt 7.1 werden Datentypen definiert, aus denen sich Problem– und Kodierungsstruktur, sowie Individuen und Populationen zusammensetzen lassen. Abschnitt 7.2 und 7.3 geben eine Auflistung der Eingaben des Benutzers bzw. der Ausgaben des Systems. In Abschnitt 7.4 wird die Problemstruktur eingeführt. Ein Schwerpunkt des formalen Ansatzes liegt in der Herleitung der Kodierungsfunktion und –struktur (Abschnitt 7.5). Abschnitt 7.6 stellt die Individuen und Populationen formal vor. Der zweite Schwerpunkt dieses Kapitels liegt in der Formalisierung der Operatoren in Abschnitt 7.7. Den Abschluß bildet der Abschnitt 7.8, in dem ein Experiment formal definiert wird.

Ohne Beschränkung der Allgemeinheit werden für diesen formalen Ansatz Maximierungsprobleme zugrundegelegt.

7.1 Grunddatentypen

Es werden zunächst die Grunddatentypen definiert, die dem Benutzer zur Verfügung stehen, um das Problem und die Kodierung zu beschreiben.

Reelle Datentypen
Die Systemkonstanten $min_{I\!R}$ und $max_{I\!R}$ kennzeichnen die minimale und die maximale vom System akzeptierte reelle Zahl.

Damit lassen sich die folgenden reellen Datentypen definieren. Insbesondere zur Beschreibung der binären Kodierung von reellen Zahlen wird zusätzlich eine *Genauigkeit* s eingeführt.

$$\mathcal{R}_{m,n} := \left\{ r \in I\!R \;\middle|\; m \leq r \leq n \right\} \text{ für } min_{I\!R} \leq m < n \leq max_{I\!R},\ m,n \in I\!R.$$
$$\mathcal{R}_{m,n,s} := \left\{ r \in \mathcal{R}_{m,n} \;\middle|\; \exists i \in I\!N_0 : r = m + i * s \right\} \cup \{n\} \text{ für } m,n,s \in I\!R.$$

Es gilt offensichtlich $\mathcal{R}_{m,n,s} \subset \mathcal{R}_{m,n}$.

Diese werden jeweils zusammengefaßt in den Mengen der reellen Datentypen:

$$\mathcal{R} := \left\{ \mathcal{R}_{m,n} \;\middle|\; min_{I\!R} \leq m < n \leq max_{I\!R},\ m,n \in I\!R \right\}.$$
$$\tilde{\mathcal{R}} := \left\{ \mathcal{R}_{m,n,s} \;\middle|\; min_{I\!R} \leq m < n \leq max_{I\!R},\ s > 0,\ m,n,s \in I\!R \right\}.$$

Die Zuweisung von $r \in I\!R$ an eine Variable vom Datentyp $\mathcal{R}_{m,n,s}$ soll implizit durch die Funktion

$$RoundReal_{m,n,s}(r) := \begin{cases} \max\left\{ r' \in \mathcal{R}_{m,n,s} \;\middle|\; \mid r' - r \mid \leq \frac{1}{2}s \right\}, & \text{falls } \ m \leq r \leq n \\ \text{undef.}, & \text{sonst} \end{cases}$$

erfolgen.

Ganzzahlige Datentypen

Die Systemkonstanten $min_{Z\!\!Z}$ und $max_{Z\!\!Z}$ kennzeichnen die minimale und die maximale vom System akzeptierte ganzzahlige Zahl.

Damit lassen sich die folgenden ganzzahligen Datentypen definieren. Insbesondere zur Beschreibung der binären Kodierung wird zusätzlich eine *Schrittweite* s eingeführt.

$$\mathcal{I}_{m,n} := \left\{ z \in Z\!\!Z \;\middle|\; m \leq z \leq n \right\} \text{ für } min_{Z\!\!Z} \leq m < n \leq max_{Z\!\!Z},\ m,n \in Z\!\!Z.$$
$$\mathcal{I}_{m,n,s} := \left\{ z \in \mathcal{I}_{m,n} \;\middle|\; \exists i \in I\!N_0 : z = m + i * s \right\} \cup \{n\},\ m,n \in Z\!\!Z,\ s \in I\!N.$$

Es gilt offensichtlich $\mathcal{I}_{m,n,s} \subset \mathcal{I}_{m,n}$.

Diese werden jeweils zusammengefaßt in den Mengen der ganzzahligen Datentypen:

$$\mathcal{I} := \left\{ \mathcal{I}_{m,n} \;\middle|\; min_{Z\!\!Z} \leq m < n \leq max_{Z\!\!Z},\ m,n \in Z\!\!Z \right\}.$$
$$\tilde{\mathcal{I}} := \left\{ \mathcal{I}_{m,n,s} \;\middle|\; min_{Z\!\!Z} \leq m < n \leq max_{Z\!\!Z},\ s > 0,\ m,n \in Z\!\!Z,\ s \in I\!N \right\}.$$

Die Zuweisung von $z \in Z\!\!Z$ an eine Variable vom Datentyp $\mathcal{I}_{m,n,s}$ soll implizit durch die Funktion

$$RoundInt_{m,n,s}(z) := \begin{cases} \max\left\{ z' \in \mathcal{I}_{m,n,s} \;\middle|\; \mid z' - z \mid \leq \frac{1}{2}s \right\}, & \text{falls } \ m \leq z \leq n \\ \text{undef.}, & \text{sonst} \end{cases}$$

erfolgen.

Binäre Datentypen
Es wird der folgende *binäre Grunddatentyp* definiert:

$$\mathcal{B} := \{0, 1\}.$$

Dann ist $\{\mathcal{B}\}$ die *Menge aller binären Grunddatentypen.* Weiter werde noch der *Bitstring* definiert als $\mathcal{B}^k := \underbrace{\mathcal{B} \times \cdots \times \mathcal{B}}_{k\text{–mal}}$.
Dann ist die *Menge aller Bitstrings* $\tilde{\mathcal{B}} := \left\{ \mathcal{B}^k \;\middle|\; k \geq 1 \right\}$.

Permutation
Es sind folgende Datentypen für Permutationen definiert:

$$\mathcal{P}_n := \left\{ \pi : \mathcal{I}_{1,n} \to \mathcal{I}_{1,n} \;\middle|\; \pi \; ist \; bijektiv \right\}.$$

Die Bezeichnung $\mathcal{P}_n$ dient hier ausschließlich für die Bezeichnung der Datentypen, die dem Benutzer für die Eingabe der Problemstruktur zur Verfügung stehen. Sie wird hier nicht zur weiteren Beschreibung des Systems benutzt. Zur Beschreibung der Kodierungsfunktion wird die Menge der Permutationen der Länge $n \in I\!N$ mit Π_n bezeichnet. Dabei ist natürlich $\Pi_n = \mathcal{P}_n$. Die Unterscheidung dient lediglich dem besseren Verständnis.

Dann ist $\mathcal{P} := \left\{ \mathcal{P}_n \;\middle|\; n > 1 \right\}$ die *Menge aller Permutationen.*

Mengen der Datentypen
Die *Menge der Grunddatentypen* ist

$$\mathcal{A} := \mathcal{R} \cup \mathcal{I} \cup \{\mathcal{B}\} \cup \mathcal{P}.$$

Für die Kodierung werden noch die beiden folgenden Mengen definiert:

$$\hat{\mathcal{A}} := \mathcal{A} \cup \tilde{\mathcal{R}} \cup \tilde{\mathcal{I}},$$
$$\tilde{\mathcal{A}} := \hat{\mathcal{A}} \cup \tilde{\mathcal{B}}.$$

Dabei ist $\hat{\mathcal{A}}$ die *Menge aller eingeschränkten Grunddatentypen* und $\tilde{\mathcal{A}}$ die *Menge aller kodierten Grunddatentypen.*

Ein konkreter Datentyp $A \in \mathcal{A}$ kann damit als eine Menge von Belegungen gesehen werden. Eine solche Belegung $a \in A$ ist dann ein bestimmter Wert des Datentyps A.

In den folgenden Abschnitten werden mit Hilfe der Problemstruktur und der Kodierungsfunktion noch die Datentypen aller möglichen Individuen $\mathcal{INDV}$ und aller

möglichen Populationen $\mathcal{POP}$ definiert. Damit ergeben sich die *Basisdatentypen für die Operatoren*:

$$\mathcal{DAT} := \mathcal{A} \cup \{\mathcal{IND}\} \cup \{\mathcal{POP}\}.$$

In den Operatoren ist noch zusätzlich der Datentyp `ARRAY` mit n Dimensionen definiert:

$$\mathcal{AR} := \left\{ \mathcal{DAT}^{k_1 \cdots k_n} \;\middle|\; n, k_1, \ldots, k_n \in I\!N \right\}.$$

Damit ergibt sich schließlich die *Menge aller Datentypen für Operatoren*:

$$\overline{\mathcal{DAT}} := \mathcal{DAT} \cup \mathcal{AR}.$$

7.2 Eingaben des Benutzers

EAGLE erwartet vom Benutzer die folgenden Eingaben zur Initialisierung des Systems:

- Problemstruktur $SP \in \mathcal{A}^l$ (siehe Abschnitt 7.4), wobei $l \in I\!N$ die Länge der Problemstruktur ist.
- Kodierungsfunktion, bestehend aus (siehe Abschnitt 7.5)
  - Kodierungsfunktion $C \in \mathcal{K}^l$ für die Problemstruktur.
  - zusätzlichen Atomen $ZP \in \mathcal{A}^{l'}$, wobei $l' \in I\!N$ die Anzahl der zusätzlichen, kodierungsbedingten Atome (z.B. Strategieparameter) ist.
  - Anordnung aller Atome $\pi \in \Pi_{l''}$, wobei $l'' := l + l'$.
- mehreren Operatoren:
  - ein Operator $OP_{fitness} = OP_0$ für die Fitneßfunktion.
  - ein Operator $OP_{main} = OP_1$ für den Hauptoperator, welcher den Evolutionären Algorithmus darstellt.
  - eine beliebige Anzahl $N \in I\!N$ weiterer Operatoren OP_i mit $2 \leq i \leq N+1$.

  Jeder eingegebene Operator OP_i besteht aus den folgenden Bestandteilen:
  - einer Menge von Variablen

    $$VAR_i \in \overline{\mathcal{DAT}}^{l_{VAR_i}}$$

    und einer Default–Belegung $var_i^* \in VAR_i$.

- einer Menge von statischen Variablen

  $$STATVAR_i \in \overline{\mathcal{DAT}}^{l_{STATVAR_i}}$$

  und einer Default–Belegung $statvar_i^* \in STATVAR_i$.
- einer Menge von Konstanten

  $$CONST_i \in (\mathcal{I} \cup \mathcal{R} \cup \{\mathcal{B}\})^{l_{CONST_i}}.$$

- einer Menge von Parametern

  $$PARAM_i \in (\mathcal{I} \cup \mathcal{R} \cup \{\mathcal{B}\})^{l_{PARAM_i}}$$

  und einer Default–Belegung $param_i^* \in PARAM_i$.
- einer Menge von call–by–value–Aufrufparametern

  $$CALLVAL_i \in \mathcal{DAT}^{l_{CALLVAL_i}}.$$

- einer Menge von call–by–reference–Aufrufparametern

  $$CALLREF_i \in \mathcal{DAT}^{l_{CALLREF_i}}.$$

- dem Rückgabetyp

  $$RETURN_i \in \mathcal{DAT}.$$

- der Funktion F_i, welche durch den Algorithmen–Teil eines Operators definiert wird.
- einer Menge von Label

  $$LABEL_i := \{label_{i,1}, \ldots, label_{i,l_{LABEL_i}}\}.$$

- einer Menge von Filtern

  $$FILTER_i := \{filter_{i,1}, \ldots, filter_{i,l_{FILTER_i}}\}.$$

Vor einem Experiment sind in der Laufinitialisierung noch weitere Eingaben notwendig:

- eine Belegung der Parameter $param_i \in PARAM_i$, sonst gilt $param_i := param_i^*$.
- eine globale Menge von Filestreams

  $$STREAMS := \{stream_1, \ldots, stream_{l_{STREAMS}}\}$$

  für die Filterung.
- eine Zuordnung der Filter zu den Filestreams

  $$FSTREAM_i : FILTER_i \rightarrow STREAMS \cup \{NOSTREAM\},$$

  wobei $NOSTREAM$ dafür steht, daß der Filter inaktiv ist.

- eine Eingabe des Status für die Label durch

$$LABSTAT_i : LABEL_i \rightarrow \{Inaktiv, Halt, Abbruch\}.$$

- eine Anfangspopulation $P_0 \in \mathcal{POP}$, hier kann eine zufällige neue oder eine bereits vorhandene gewählt werden.

Weiter gibt es noch die Rechenzustände

$$VALID := \{NORMAL, HALT, ABBRUCH\}.$$

Anfangs ist der Zustand $valid := NORMAL$. Während eines Experiments, d.h. der Berechnung eines Ergebnisses, ist es dem Benutzer jederzeit möglich, diesen Zustand zu ändern.

7.3 Ausgaben von EAGLE

EAGLE berechnet bei einem Experiment die folgenden Ergebnisse:

- ein bestes Individuum $bestind \in \mathcal{IND}$.
- eine Endpopulation $P_{end} \in \mathcal{POP}$.
- die gefilterten Werte

$$filval \in FILVAL := FILVAL_1 \times \ldots \times FILVAL_{l_{STREAMS}},$$

$$\text{mit} \quad FILVAL_j := \left\{ filval_j \;\middle|\; filval_j \in \{\mathcal{DAT}\}^* \right\}, \quad 1 \leq j \leq l_{STREAMS}.$$

- die Bildschirmausgabe $scrn \in SCRN := \hat{\Sigma}^*$.

Dabei sei $\hat{\Sigma}$ ein der Bildschirmausgabe zugrundeliegendes Alphabet.

7.4 Problemstruktur

Mittels der eingeführten Grunddatentypen $A \in \mathcal{A}$ kann sich der Benutzer eine Problemstruktur konstruieren, auf der die zu optimierende Fitneßfunktion arbeitet.

Sei $l \in I\!N$ die *Länge der Problemstruktur*, so ist eine spezielle Problemstuktur $SP = (SP_1, \ldots, SP_l) \in \mathcal{A}^l$ ein Tupel von gewählten Grunddatentypen $SP_i \in \mathcal{A}$. Die Komponenten dieses Tupels werden im folgenden als *Atome* bezeichnet.

7.5 Kodierungsstruktur

Aus einer Problemstruktur $SP \in \mathcal{A}^l$ läßt sich mittels einer Kodierungsfunktion eine zugehörige Kodierungsstruktur konstruieren. Nachfolgend wird schrittweise die Kodierungsfunktion und damit auch die Kodierungsstruktur hergeleitet, vgl. auch die Abbildung 7.1.

Dazu werden zunächst die verschiedenen Kodierungsarten für einzelne Atome vorgestellt (Abschnitt 7.5.1). Dann wird die Problemstruktur eingeschränkt auf einen Bereich, der zur jeweils gewählten Kodierungsart passend ist (Abschnitt 7.5.2). Die eingeschränkten Atome der Problemstruktur werden in Abschnitt 7.5.3 kodiert. In Abschnitt 7.5.4 werden zusätzliche Atome eingeführt und das so erhaltene Tupel von kodierten Atomen wird umsortiert. Schließlich wird dann die Kodierungsstruktur aus Sicht der Operatoren eingeführt (Abschnitt 7.5.5). Der Abschnitt 7.5.6 gibt eine abschließende Übersicht über die gesamte Kodierungsfunktion.

Im gesamten Abschnitt 7.5 wird die Kodierung auf zwei Ebenen durchgeführt. Zum einen werden die Datentypen, die die Problemstruktur beschreiben, betrachtet. Parallel dazu wird auch die Kodierungsfunktion für die konkrete Belegung fester Datentypen angegeben.

7.5.1 Kodierungsarten für einzelne Atome

Es sind die folgenden *Kodierungsarten* vorgesehen:

$$\begin{aligned}\mathcal{K} := \{id\} &\cup \left\{(realbinary, s) \;\middle|\; 0 \leq s \leq max_{\boldsymbol{R}}\right\} \\ &\cup \left\{(realgray, s) \;\middle|\; 0 \leq s \leq max_{\boldsymbol{R}}\right\} \\ &\cup \left\{(intbinary, s) \;\middle|\; 1 \leq s \leq max_{\boldsymbol{Z}}\right\} \\ &\cup \left\{(intgray, s) \;\middle|\; 1 \leq s \leq max_{\boldsymbol{Z}}\right\},\end{aligned}$$

dabei kennzeichnet s die Genauigkeit bzw. die Schrittweite bei der Kodierung.

Diese Kodierungsarten ermöglichen es, die Datentypen $A = \mathcal{R}_{m,n}$ und $A = \mathcal{I}_{m,n}$ binär zu kodieren, indem jeweils der Wertebereich, aus dem die konkreten Belegungen gewählt werden dürfen, diskretisiert wird.

Sei jetzt für die weiteren Abschnitte $C = (c_1, \ldots, c_l) \in \mathcal{K}^l$ das vom Benutzer gewählte Tupel der Kodierungsarten c_i für die Atome SP_i aus $SP = (SP_1, \ldots, SP_l) \in \mathcal{A}^l$.

7.5.2 Eingeschränkte Problemstuktur

Die folgende Funktion schränkt Grunddatentypen entsprechend einer gewählten Kodierungsart so ein, daß sie binär kodiert werden können.

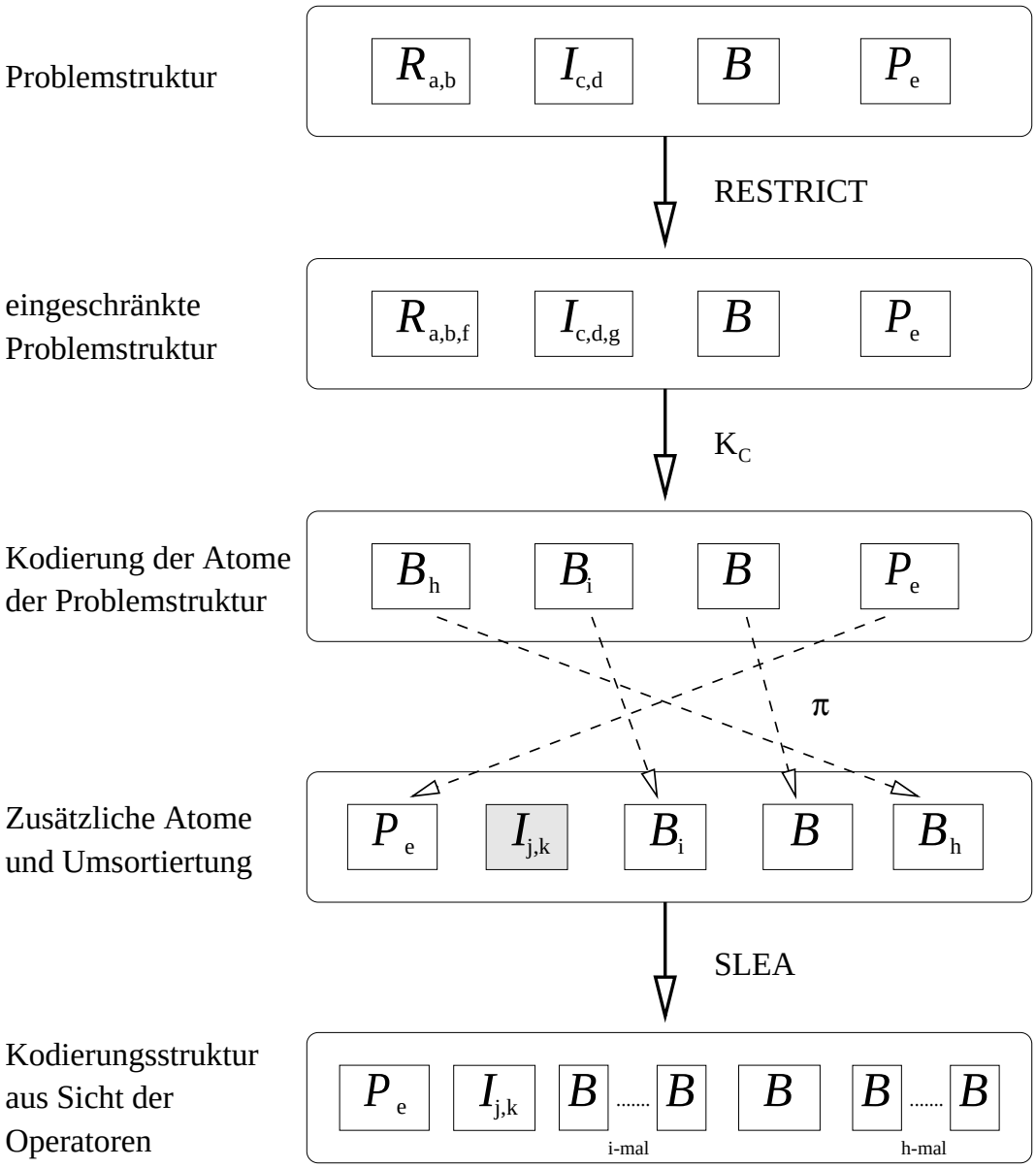


Abbildung 7.1: Schrittweise Herleitung der Kodierungsstruktur

$$\begin{aligned} RESTRICT_{elem} : \mathcal{A} \times \mathcal{K} &\longrightarrow \hat{\mathcal{A}} \\ (A, c) &\longmapsto \hat{A}, \end{aligned}$$

$$\text{mit} \quad \hat{A} := \begin{cases} \mathcal{R}_{m,n}, & \text{falls } A = \mathcal{R}_{m,n} \text{ und } c = id \\ \mathcal{R}_{m,n,s}, & \text{falls } A = \mathcal{R}_{m,n} \text{ und } c = (realbinary, s) \\ & \quad \text{bzw. } c = (realgray, s) \\ \mathcal{I}_{m,n}, & \text{falls } A = \mathcal{I}_{m,n} \text{ und } c = id \\ \mathcal{I}_{m,n,s}, & \text{falls } A = \mathcal{I}_{m,n} \text{ und } c = (intbinary, s) \\ & \quad \text{bzw. } c = (intgray, s) \\ \mathcal{B}, & \text{falls } A = \mathcal{B} \text{ und } c = id \\ \mathcal{P}_n, & \text{falls } A = \mathcal{P}_n \text{ und } c = id \\ \text{undef.}, & \text{sonst.} \end{cases}$$

Durch komponentenweises Anwenden der Funktion $RESTRICT_{elem}$ auf $\mathcal{A}^l \times \mathcal{K}^l$ ergibt sich die Funktion $RESTRICT$:

$$\begin{aligned} RESTRICT : \mathcal{A}^l \times \mathcal{K}^l &\longrightarrow \hat{\mathcal{A}}^l \\ ((A_1, \ldots, A_l), (c_1, \ldots, c_l)) &\longmapsto (\hat{A}_1, \ldots, \hat{A}_l), \end{aligned}$$

$$\text{mit} \quad \hat{A}_j := RESTRICT_{elem}(A_j, c_j), \quad 1 \leq j \leq l.$$

Somit ergibt sich die eingeschränkte Problemstruktur $SP|_C := RESTRICT(SP, C)$.

7.5.3 Kodierung der Atome der Problemstruktur

Für ein einzelnes Atom $A \in \hat{\mathcal{A}}$ der eingeschränkten Problemstruktur wird durch ein gewähltes $c \in \mathcal{K}$ das kodierte Atom bestimmt. Dies wird durch die *Kodierungsfunktion für eingeschränkte Atome* beschrieben:

$$\begin{aligned} K_{elem} : \hat{\mathcal{A}} \times \mathcal{K} &\longrightarrow \tilde{\mathcal{A}} \\ (\hat{A}, c) &\longmapsto \tilde{A}. \end{aligned}$$

Für ein festes $c \in \mathcal{K}$ definiert die Kodierungsfunktion K eine injektive Zuordnung der Belegung $a \in A$ eines Datentyps $A \in \hat{\mathcal{A}}$ zu einer konkreten Belegung $\tilde{a} \in \tilde{A}$. Die Zuordnung

$$\begin{aligned} Kod_c : \hat{A} &\longrightarrow \tilde{A} = K_{elem}(\hat{A}, c) \\ \hat{a} &\longmapsto \tilde{a} \end{aligned}$$

heiße *elementare Kodierung*. Durch $c = id$ wird dabei immer die Identität induziert.

- $K_{elem}(\mathcal{R}_{m,n}, c) := \begin{cases} \mathcal{R}_{m,n}, & \text{falls } c = id \\ \text{undef.}, & \text{sonst.} \end{cases}$

- $K_{elem}(\mathcal{R}_{m,n,s}, c) := \mathcal{B}_k, \quad$ wobei $k \in I\!N$ mit $k := \lceil \log_2(\lfloor \frac{n-m+1}{s} \rfloor) \rceil$.

  - Falls $c = (realbinary, s)$, wird die folgende elementare Kodierung induziert:

$$\begin{aligned} Kod_{realbinary,s} : \mathcal{R}_{m,n,s} &\longrightarrow \mathcal{B}_k \\ r &\longmapsto b_{k-1} \cdots b_1 b_0 \end{aligned}$$

$$\text{mit} \quad r = m + s * \sum_{i=0}^{k-1} b_i * 2^i.$$

  - Für $c = (realgray, s)$ analog, wobei $b_{k-1} \cdots b_1 b_0$ die Gray–Kodierung von r ist.

- $K_{elem}(\mathcal{I}_{m,n}, c) := \begin{cases} \mathcal{I}_{m,n}, & \text{falls } c = id \\ \text{undef.}, & \text{sonst.} \end{cases}$

- $K_{elem}(\mathcal{I}_{m,n,s}, c) := \mathcal{B}_k, \quad$ wobei $k \in I\!N$ mit $k := \lceil \log(\lfloor \frac{n-m+1}{s} \rfloor) \rceil$.

  - Falls $c = (intbinary, s)$ wird die folgende elementare Kodierung induziert:

$$\begin{aligned} Kod_{intbinary,s} : \mathcal{I}_{m,n,s} &\longrightarrow \mathcal{B}_k \\ z &\longmapsto b_{k-1} \cdots b_1 b_0 \end{aligned}$$

$$\text{mit} \quad z = m + s * \sum_{i=0}^{k-1} b_i * 2^i.$$

  - Für $c = (realgray, s)$ analog, wobei $b_{k-1} \cdots b_1 b_0$ die Gray–Kodierung von z ist.

- $K_{elem}(\mathcal{B}, c) := \begin{cases} \mathcal{B}, & \text{falls } c = id \\ \text{undef.}, & \text{sonst.} \end{cases}$

- $K_{elem}(\mathcal{P}_n, c) := \begin{cases} \mathcal{P}_n, & \text{falls } c = id \\ \text{undef.}, & \text{sonst.} \end{cases}$

Solche $c \in \mathcal{K}$, für die die Kodierungsfunktion nicht definiert ist, sind als Eingabe nicht zulässig.

Dadurch, daß bei den Datentypen $A = \mathcal{I}_{m,n}$ und $A = \mathcal{R}_{m,n}$ die Schrittweite bzw. Genauigkeit frei wählbar sind, ergibt sich bei einer beliebigen binären Kodierung das Problem, daß die induzierte Funktion Kod_c nicht immer surjektiv ist. D.h. es können

beim Verändern des Wertes eines kodierten Atoms, auf dem ja nachher der Evolutionäre Algorithmus auch arbeitet, Werte entstehen, für die es keinen entsprechenden Wert im Atom der Problemstruktur gibt. Dies wird vom System EAGLE automatisch ausgeschlossen.

D.h. folgendes: wird ein Bit (oder mehrere Bits) eines kodierten Atoms so geändert, daß ein unzulässiger Wert entsteht, wird diese Änderung nicht akzeptiert — es bleibt also der alte Wert bestehen. Durch diese Einschränkung des Bildbereichs wird die injektive Funktion Kod_c zur Bijektion.

Aus den Funktionen K_{elem} und Kod_c auf einzelnen Atomen lassen sich ebenfalls komponentenweise die Funktionen K und Kod_C definieren:

$$\begin{aligned} K : \hat{\mathcal{A}}^l \times \mathcal{K}^l &\longrightarrow \tilde{\mathcal{A}}^l\Big|_{Kod_C} = \tilde{\mathcal{A}}^l\Big|_{Kod_{(c_1,\ldots,c_l)}} \\ \Big((\hat{A}_1,\ldots,\hat{A}_l),(c_1,\ldots,c_l)\Big) &\longmapsto (\tilde{A}_1,\ldots,\tilde{A}_l) \end{aligned}$$

mit $\tilde{A}_j := K_{elem}(\hat{A}_j, c_j)$ und

$$\begin{aligned} Kod_C : (\hat{A}_1,\ldots,\hat{A}_l) &\longrightarrow (\tilde{A}_1,\ldots,\tilde{A}_l) \\ (\hat{a}_1,\ldots,\hat{a}_l) &\longmapsto (\tilde{a}_1,\ldots,\tilde{a}_l), \end{aligned}$$

mit $\tilde{a}_j := Kod_{c_j}(\hat{a}_j), \quad 1 \le j \le l.$

Durch die Einschränkung des Bildbereichs wird Kod_C eine bijektive Abbildung.

Somit ergibt sich die *kodierte Problemstruktur*

$$SK^* := K(SP\big|_C, C)$$

und die Kodierungsfunktion für die konkreten Belegungen der Problemstruktur

$$Kod_C : SP\big|_C \longrightarrow SK^*.$$

7.5.4 Zusätzliche Atome und Umsortierung

Neben den kodierten Atomen aus der Problemstruktur können in der Kodierungsstruktur auch noch weitere Atome (z.B. als Strategieparameter) zugefügt werden.

Sei $l' \in I\!N$ die *Anzahl dieser zusätzlichen Atome* und $ZP \in \mathcal{A}^{l'}$ die *zusätzlichen Atome*. Da $\mathcal{A}^{l'} \subset \tilde{\mathcal{A}}^{l'}$ ist $ZP \in \tilde{\mathcal{A}}^{l'} = \tilde{\mathcal{A}}^{l'}\Big|_{Kod_{(c'_1,\ldots,c'_{l'})}}$, mit $c'_1 := \cdots := c'_{l'} := id$.

Die endgültige Kodierungsstruktur erhält man nun durch beliebiges Umsortieren von SK^* und ZP. Mit Hilfe einer vom Benutzer gewählten Permutation $\pi \in \Pi_{l+l'}$, wird die Kodierungsstruktur aus der kodierten Problemstruktur und den zusätzlichen Atomen erstellt. Sei dazu $l'' := l + l'$ die Länge der Kodierungsstruktur.

$$\begin{aligned} SORT : \tilde{\mathcal{A}}^l\Big|_{Kod_{(c_1,\ldots,c_l)}} \times \tilde{\mathcal{A}}^{l'}\Big|_{Kod_{(c'_1,\ldots,c'_{l'})}} \times \Pi_{l''} &\longrightarrow \tilde{\mathcal{A}}^{l''} \\ \Big((\tilde{A}_1,\ldots,\tilde{A}_l),(\tilde{A}_{l+1},\ldots,\tilde{A}_{l+l'}),\pi\Big) &\longmapsto (A^*_1,\ldots,A^*_{l''}), \end{aligned}$$

$$\text{mit} \quad A^*_j := A^{**}_{\pi(j)} \quad \text{und} \; (A^{**}_1,\ldots,A^{**}_{l''}) := (\tilde{A}_1,\ldots,\tilde{A}_l) \circ (\tilde{A}_{l+1},\ldots,\tilde{A}_{l+l'}).$$

Der eigentliche Bildbereich der Abbildung wird durch die Kodierung der einzelnen Atome und die Umordnung der Atome bestimmt: $\tilde{\mathcal{A}}^{l''}\Big|_{Kod_{(c^*_1,\ldots,c^*_{l''})}}$, wobei $c^*_j := c^{**}_{\pi(j)}$ mit $(c^{**}_1,\ldots,c^{**}_{l''}) := (c_1,\ldots,c_l) \circ (c'_1,\ldots,c'_{l'})$.

Für eine feste Permutation $\pi \in \Pi_{l''}$ läßt sich auf Ebene der konkreten Datentypen

$$((\tilde{A}_1,\ldots,\tilde{A}_l),(\tilde{A}_{l+1},\ldots,\tilde{A}_{l+l'})) = (\tilde{A}^l, A^{l'}) \in \tilde{\mathcal{A}}^l\Big|_{Kod_C} \times \mathcal{A}^{l'}\Big|_{Kod_{C'}},$$

mit $C = (c_1,\ldots,c_l)$ und $C' = (c'_1,\ldots,c'_{l'})$ eine analoge Abbildung einführen:

$$\begin{aligned} Sort_\pi : (\tilde{A}_1,\ldots,\tilde{A}_l) \times (\tilde{A}_{l+1},\ldots,\tilde{A}_{l+l'}) &\longrightarrow (A^*_1,\ldots,A^*_{l''}) \\ ((\tilde{a}_1,\ldots,\tilde{a}_l),(\tilde{a}_{l+1},\ldots,\tilde{a}_{l+l'})) &\longmapsto (a^*_1,\ldots,a^*_{l''}), \end{aligned}$$

mit $a^*_j := a^{**}_{\pi(j)}$ und $(a^*_1,\ldots,a^*_{l''}) := (\tilde{a}_1,\ldots,\tilde{a}_l) \circ (\tilde{a}_{l+1},\ldots,\tilde{a}_{l+l'})$.

Somit ergibt sich die Kodierungsstruktur

$$SK := SORT(SK^*, ZP, \pi) \in \tilde{\mathcal{A}}^{l''}\Big|_{Kod_{(c^*_1,\ldots,c^*_{l''})}}$$

und die einzelnen Elemente aus SK durch

$$sk = (sk_1,\ldots,sk_{l''}) := Sort_\pi(sk^*, zp) \in (\tilde{A}_1,\ldots,\tilde{A}_{l''}),$$

mit $sk^* = (sk^*_1,\ldots,sk^*_l) \in (\tilde{A}_1,\ldots,\tilde{A}_l) \in \tilde{\mathcal{A}}^l$
und $zp = (zp_1,\ldots,zp_{l'}) \in (A_1,\ldots,A_{l'}) \in \tilde{\mathcal{A}}^{l'}$.

7.5.5 Kodierungsstruktur aus Sicht der Operatoren

Auf der Kodierungsstruktur SK soll schließlich der Evolutionäre Algorithmus arbeiten. Allerdings wird bei der Ausführung eines EA nur auf einzelne Grunddatentypen zugegriffen und nicht etwa auf Bitstrings, wie sie in SK durch die binäre Kodierung der Atome der Problemstruktur enthalten seien können. Die verschiedenen Bits eines Bitstrings werden als einzelne Atome behandelt, was z.B. aus der Sicht eines Genetischen Algorithmus sinnvoll ist, da dort die Bedeutung eines einzelnen Bits für das Verfahren völlig gleichgültig ist.

Dies motiviert die folgenden Definitionen:

Sei für $SK = (SK_1, \cdots, SK_{l''})$:

$$l''' := \sum_{\substack{1 \le i \le l'' \\ SK_i \in \mathcal{A}}} 1 + \sum_{\substack{1 \le i \le l'' \\ SK_i \notin \mathcal{A}}} L(SK_i), \text{ wobei } L(SK_i) := m \text{ für } SK_i = \mathcal{B}_m.$$

So entspricht l''' der Anzahl der Grunddatentypen $A \in \mathcal{A}$ in der Kodierungsstruktur $SK \in \tilde{\mathcal{A}}^{l''}\big|_{Kod_{(c^*_1, \cdots, c^*_{l''})}}$. Die folgende Funktion $SLEA$ führt nun die Kodierungsstruktur SK in die Struktur über, die für die Operatoren sichtbar ist. Diese Struktur wird im folgenden *Kodierung aus Sicht der Operatoren*, $SL \in \mathcal{A}^{l'''}\big|_{Kod} \subseteq \mathcal{A}^{l'''}$ genannt. Dabei soll $|_{Kod}$ hier lediglich noch kennzeichnen , daß verschiedene Elemente durch die induzierte Kodierung ausgeschlossen wurden.

$$\begin{aligned} SLEA : \tilde{\mathcal{A}}^{l''}\big|_{Kod_{(c^*_1, \cdots, c^*_{l''})}} &\longrightarrow \mathcal{A}^{l'''}\big|_{Kod} \\ (\tilde{A}_1, \ldots, \tilde{A}_{l''}) &\longmapsto (A_1, \ldots, A_{l'''}), \end{aligned}$$

$$\text{mit} \quad A_k := \begin{cases} \tilde{A}_j & \text{falls } \tilde{A}_j \notin \tilde{\mathcal{B}} \text{ und } k = \sum_{\substack{1 \le i \le j \\ \tilde{A}_i \in \mathcal{A}}} 1 + \sum_{\substack{1 \le i \le j \\ \tilde{A}_i \notin \mathcal{A}}} L(\tilde{A}_i) \\ \hat{\mathcal{B}} & \text{falls } \tilde{A}_j = (b_1, \cdots, b_n) = \mathcal{B}_n,\ \hat{\mathcal{B}} = b_{j'} \\ & \quad \text{und } k = \sum_{\substack{1 \le i < j \\ \tilde{A}_i \in \mathcal{A}}} 1 + \sum_{\substack{1 \le i < j \\ \tilde{A}_i \notin \mathcal{A}}} L(\tilde{A}_i) + j', \end{cases}$$

und $1 \le k \le l'''$.

Auf der Ebene der konkreten Belegung der Datentypen gibt es eine analoge Funktion $Slea$:

$$\begin{aligned} Slea : (\tilde{A}_1, \ldots, \tilde{A}_{l''}) &\longrightarrow (A_1, \ldots, A_{l'''}) \\ \tilde{a} = (\tilde{a}_1, \ldots, \tilde{a}_{l''}) &\longmapsto (a_1, \ldots, a_{l'''}), \end{aligned}$$

$$\text{mit} \quad a_k := \begin{cases} \tilde{a}_j & \text{falls } \tilde{A}_j \notin \tilde{\mathcal{B}} \text{ und } k = \sum_{\substack{1 \le i \le j \\ \tilde{A}_i \in \mathcal{A}}} 1 + \sum_{\substack{1 \le i \le j \\ \tilde{A}_i \notin \mathcal{A}}} L(\tilde{A}_i) \\ b_{j'} & \text{falls } \tilde{A}_j = (b_1, \cdots, b_n) = \mathcal{B}_n, \\ & \quad \text{und } k = \sum_{\substack{1 \le i < j \\ \tilde{A}_i \in \mathcal{A}}} 1 + \sum_{\substack{1 \le i < j \\ \tilde{A}_i \notin \mathcal{A}}} L(\tilde{A}_i) + j', \end{cases}$$

und $1 \le k \le l'''$.

Daraus ergibt sich dann die Kodierungsstruktur aus Sicht der Operatoren:

$$SL := SLEA(SK)$$

mit der Zuweisungsfunktion auf der Ebene der konkreten Datentypen

$$sl = (sl_1, \ldots, sl_{l'''}) := Slea(sk) \in (A_1, \ldots, A_{l'''}),$$

mit $sk = (sk_1, \ldots, sk_{l''}) \in (\tilde{A}_1, \ldots, \tilde{A}_{l''})$.

7.5.6 Übersicht über die gesamte Kodierungsfunktion

Der Benutzer gibt bei der Eingabe des Problems die Problemstruktur $SP \in \mathcal{A}^l$ vor. Zur Konstruktion der Kodierungsfunktion gibt er

- die Kodierung $C \in \mathcal{K}^l$ der einzelnen Atome der Problemstruktur,
- die zusätzlichen Atome $ZP \in \mathcal{A}^{l'}$,
- die Anordnung $\pi \in \Pi_{l+l'}$ aller Atome

ein. Durch das $C \in \mathcal{K}^l$ wird die Problemstruktur weiter eingeschränkt auf

$$SP|_C := RESTRICT(SP, C) \in \tilde{\mathcal{A}}^l.$$

Aus dieser eingeschränkten Problemstruktur ergibt sich die Kodierungsstruktur aus Sicht der Operatoren folgendermaßen:

$$\begin{aligned} CODING : \hat{\mathcal{A}}^l \times \mathcal{K}^l \times \mathcal{A}^{l'} \times \Pi_{l+l'} &\longrightarrow \mathcal{A}^{l'''} \Big|_{Kod} \\ (\hat{A}, C, A', \pi) &\longmapsto A'', \end{aligned}$$

mit $A'' := SLEA\left(SORT\left(K(\hat{A}, C), A', \pi\right)\right)$.

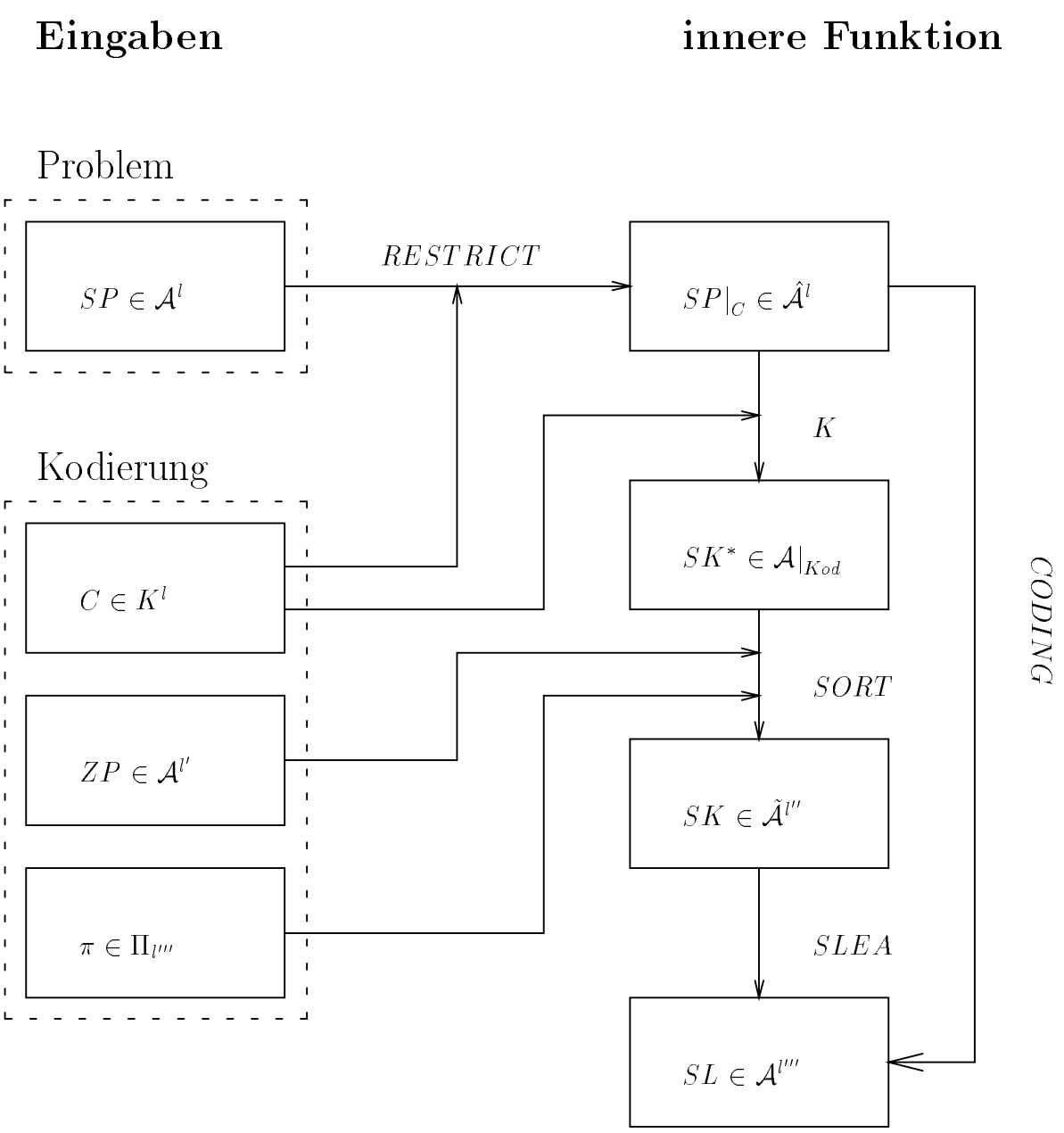


Abbildung 7.2: Übersicht über die einzelnen Schritte der Kodierung auf Ebene der Datentypen

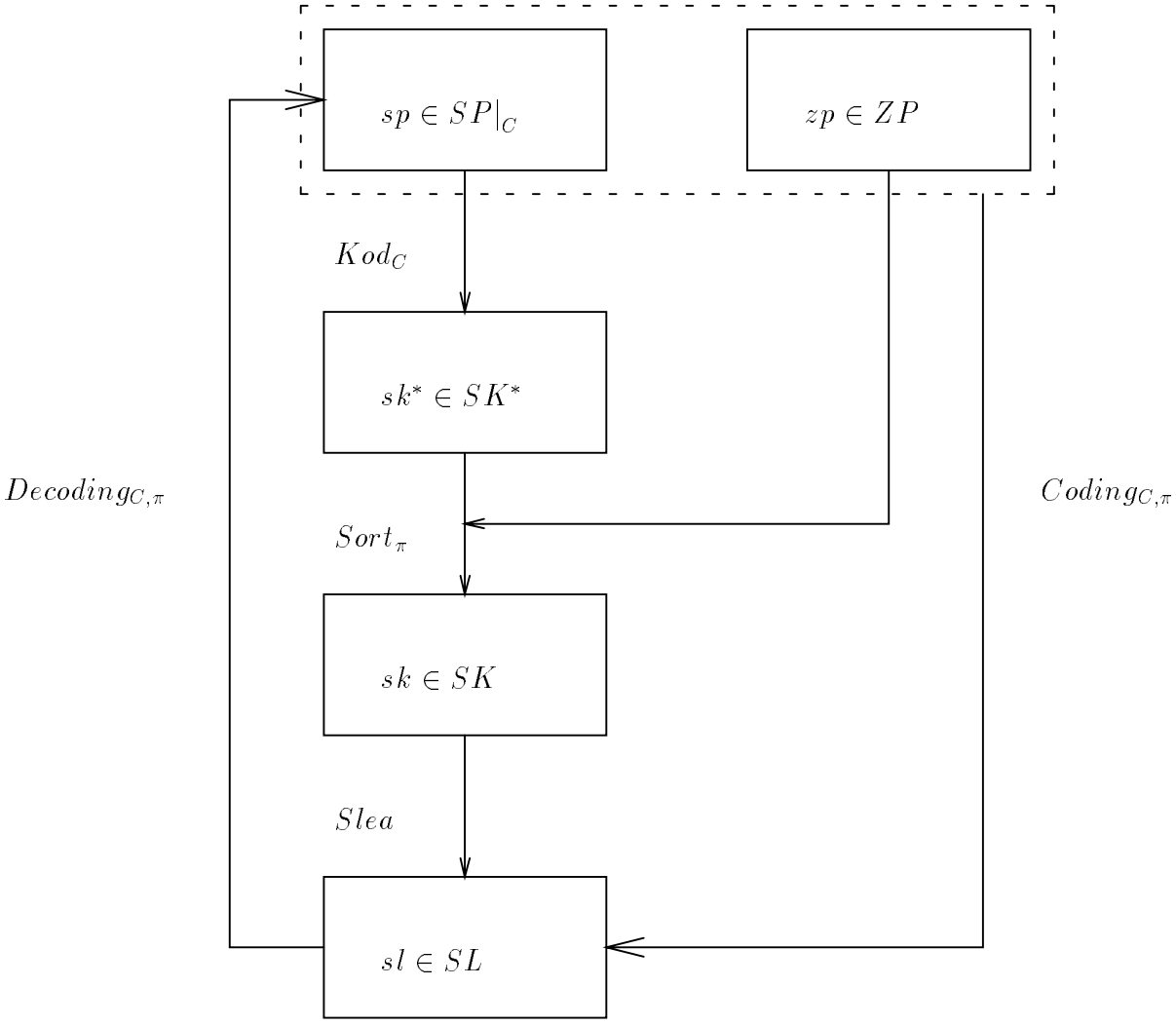


Abbildung 7.3: Übersicht über die einzelnen Schritte der Kodierung auf Ebene der konkreten Belegungen

D.h. für $SP|_C \in \hat{\mathcal{A}}^l$, $C \in \mathcal{K}^l$, $ZP \in \mathcal{A}'$ und $\pi \in \Pi_{l+l'}$ erhält man die folgende Kodierungsstruktur aus Sicht der Operatoren:

$$SL := CODING(SP|_C, C, ZP, \pi).$$

Zwischen $\hat{A} \times A' \in \hat{\mathcal{A}}^l \times \mathcal{A}^{l'}$ und $A'' \in \mathcal{A}^{l'''}|_{Kod}$ gibt es dann eine bijektive Funktion:

$$\begin{aligned} Coding_{C,\pi} : \hat{A} \times A^{l'} &\longrightarrow A'' \\ (\hat{a}, a') = ((\hat{a}_1, \ldots, \hat{a}_l), (a'_1, \ldots, a'_{l'})) &\longmapsto a'' := (a''_1, \ldots, a''_{l'''}), \end{aligned}$$

mit $a'' := Slea(Sort_\pi(K_C(\hat{a}, a')))$.

Da wir während der Konstruktion gesehen haben, daß K_C durch Einschränkungen im Definitions– und Bildbereich bijektiv ist und die Funktionen $Sort_\pi$ und $Slea$ ohnehin bijektiv sind, gibt es zu $Coding_{C,\pi}$ eine Umkehrfunktion $Coding^{-1}_{C,\pi}$. Bei der Dekodierung sind wir allerdings nur an den Elementen des eingeschränkten Problembereichs interessiert. Deshalb definieren wir die folgende Dekodierungsfunktion:

$$\begin{aligned} Decoding_{C,\pi} : A'' &\longrightarrow \hat{A} \\ a'' &\longmapsto \hat{a}, \end{aligned}$$

wobei $Coding^{-1}_{C,\pi}(a'') = (\hat{a}, a')$.

Konkret für das eingegebene Problem heißt das wiederum:

$$\begin{aligned} &Coding_{C,\pi} : SP\Big|_C \times ZP \longrightarrow SL \\ &Decoding_{C,\pi} : SL \longrightarrow SP\Big|_C. \end{aligned}$$

7.6 Individuen und Populationen

Wie man im Abschnitt 7.5 sieht, wird die Kodierungsstruktur SL ausschließlich von der Wahl der Problemstruktur SP, der Kodierung der einzelnen Atome C, der Wahl der zusätzlichen Atome ZP und der Permutation π bestimmt.

Es existiert sowohl eine Funktion $Coding$, die ein $x \in SP|_C$ und ein $y \in ZP$ auf das entsprechende $z \in SL$ abbildet, als auch eine Funktion $Decoding$, die aus $z \in SL$ das dazugehörige $x \in SP|_C$ errechnet.

D.h. wir können die Individuen als Elemente der Kodierungsstruktur auffassen. Die Belegung der Problemstruktur ist dann implizit gegeben. Zusätzlich merken wir uns bei jedem Individuum die zuletzt berechnete Fitneß.

Sei jetzt also

$$\mathcal{IND} := SL \times \mathcal{R}_{min_R, max_R}$$

die *Menge aller Individuen* , wobei in der zweiten Komponente die zuletzt berechnete Fitneß steht.

Eine Population ist ein Tupel von Individuen beliebiger Länge.

$$\mathcal{POP} := \bigcup_{0 \leq k} \mathcal{IND}^k$$

ist die *Menge aller möglichen Populationen.*

7.7 Operatoren

In diesem Abschnitt werden die Operatoren formalisiert. Nach einer kurzen Beschreibung der Bestandteile eines Operators (Unterabschnitt 7.7.1) werden in Unterabschnitt 7.7.2 die einzelnen Befehle von LEA vorgestellt. Unterabschnitt 7.7.3 beschreibt dann die definierten Ablaufkontrollkonstrukte in LEA. Die Funktionsweise eines Operators wird in Unterabschnitt 7.7.4 dargestellt.

7.7.1 Beschreibung eines Operators

Ein Operator Op_i, mit $i \in \mathcal{I}_{0,N}$ und $N \in I\!N$ Anzahl der Operatoren, besteht aus den bereits im Abschnitt 7.2 vorgestellten Bestandteilen:

- einer Menge von Variablen

  $$VAR_i \in \overline{\mathcal{DAT}}^{l_{VAR_i}}$$

  und einer Default–Belegung $var_i^* \in VAR_i$.

- einer Menge von statischen Variablen

  $$STATVAR_i \in \overline{\mathcal{DAT}}^{l_{STATVAR_i}}$$

  und einer Default–Belegung $statvar_i^* \in STATVAR_i$.

- einer Menge von Konstanten

  $$CONST_i \in (\mathcal{I} \cup \mathcal{R} \cup \{\mathcal{B}\})^{l_{CONST_i}}.$$

- einer Menge von Parametern

  $$PARAM_i \in (\mathcal{I} \cup \mathcal{R} \cup \{\mathcal{B}\})^{l_{PARAM_i}}$$

  und einer Default–Belegung $param_i^* \in PARAM_i$.

- einer Menge von call–by–value–Aufrufparametern

  $$CALLVAL_i \in \mathcal{DAT}^{l_{CALLVAL_i}}.$$

- einer Menge von call–by–reference–Aufrufparametern

  $$CALLREF_i \in \mathcal{DAT}^{l_{CALLREF_i}}.$$

- dem Rückgabetyp

  $$RETURN_i \in \mathcal{DAT}.$$

- der Funktion F_i, welche durch den Algorithmen–Teil des Operators Op_i definiert wird.

- einer Menge von Label

  $$LABEL_i := \{label_{i,1}, \ldots, label_{i,l_{LABEL_i}}\}.$$

- einer Menge von Filtern

  $$FILTER_i := \{filter_{i,1}, \ldots, filter_{i,l_{FILTER_i}}\}.$$

Sei $INPUT_i := CALLVAL_i \times CALLREF_i \times GEN \times FILVAL \times SCRN$
und $OUTPUT_i := CALLREF_i \times GEN \times RETURN_i \times FILVAL \times SCRN$.

Dann repräsentiert der Operator Op_i nach außen die Funktion

$$F_i : INPUT_i \longrightarrow OUTPUT_i$$
$$(callval_i, callref_i, gen, filval, scrn) \longmapsto (callref_i', gen', return_i, filval', scrn'),$$

wobei $GEN = \mathcal{I}_{0,max_{\mathbb{Z}}}$ der globale Generationenzähler ist, der auch in anderen Operatoren wiederverwendet werden kann.

Zur Berechnung der repräsentierten Funktion F_i arbeitet der Operator Op_i auf der Menge

$$STATE_i := VAR_i \times STATVAR_i \times CALLVAL_i \times CALLREF_i \times GEN.$$

Die inneren Tupel werden im folgenden nicht beachtet, so daß $state_i \in STATE_i$ ein Tupel der Länge $l_{STATE_i} := ql_{VAR_i} + l_{STATVAR_i} + l_{CALLVAL_i} + l_{CALLREF_i} + 1$ ist,

mit $STATE_i = \{\mathcal{DAT}\}^{l_{STATE_i}}$.

Aus einer Anfangsbelegung von $STATE_i$ wird durch den Operator Op_i eine Endbelegung berechnet, welche dann den Rückgabewert ergibt.

Die einzelnen Funktionen von LEA und die definierten Operatoren bilden dann ein Element aus $STATE_i$ auf ein Element aus $STATE_i$ ab.

7.7.2 Funktionen von LEA

In diesem Abschnitt werden die einzelnen Befehle von LEA formal definiert.

Die einzelnen Funktionen werden dabei nur auf dem sie betreffenden Teilbereich von $STATE_i$ definiert. Später werden sie dann auf dem gesamten Arbeitsbereich $STATE_i$ des Operators definiert.

Es wird unterschieden zwischen Befehlen, die nur zur Berechnung eines Ausdrucks dienen, und Befehlen die ein Element von $STATE_i$ in ein neues Element von $STATE_i$ überführen.

Im weiteren wird nicht explizit auf Elemente der Datenstruktur ARRAY $\mathcal{AR}$ eingegangen, da es keine Funktion gibt, die explizit auf einem Element von $\mathcal{AR}$ arbeitet, sondern auf die Elemente des ARRAYs wie auf die anderen Basisdatentypen auch zugegriffen wird.

Alle Funktion liefern nur dann die hier beschriebenen Werte als Ergebnis, falls der Eingabewert von $valid = NORMAL$ ist. Der Benutzer kann diesen Wert jederzeit während der Berechnung ändern. Wird $valid = HALT$ gesetzt, wird die Berechnung gestoppt, sobald der nächste Label erreicht wird, für den gilt:

$$LABSTAT(label) \neq Inaktiv.$$

Falls $valid = ABBRUCH$ gesetzt wird, wird die Berechnung unverzüglich abgebrochen.

7.7.2.1 Befehle in Ausdrücken

Die folgenden Befehle ändern den Zustand $state_i \in STATE_i$ des Operators nicht. Sie dienen lediglich zur Berechnung eines Ausdrucks z.B. in Zuweisungen oder als Bedingung in den Ablaufkontrollkonstrukten.

Die Funktionen haben daher immer die Form

$$a : \mathcal{A}_1 \times \cdots \times \mathcal{A}_n \rightarrow \mathcal{A}_0$$

mit $\forall\, 0 \leq i \leq n : \mathcal{A}_i \in \mathcal{A}$. Es wird aus mehreren Werten ein neuer Wert berechnet.

Verknüpfung arithmetischer Ausdrücke
Für $r_1 \in \mathcal{R}_{m_1,n_1}$ und $r_2 \in \mathcal{R}_{m_2,n_2}$ sind die folgenden Ausdrücke definiert:

- Der Verknüpfung `r1 + r2` entspricht die Funktion

$$\begin{aligned} a_+ : \mathcal{R}_{m_1,n_1} \times \mathcal{R}_{m_2,n_2} &\longrightarrow \mathcal{R}_{min_R,max_R} \\ (r_1, r_2) &\longmapsto \begin{cases} r_1 + r_2, & \text{falls } min_R \leq r_1 + r_2 \leq max_R \\ \text{undef.}, & \text{sonst.} \end{cases} \end{aligned}$$

  Die Verknüpfungen `r1 - r2` , `r1 * r2` und `r1 / r2` entsprechen analog konstruierten Funktionen.

- Der Verknüpfung `r1 = r2` entspricht die Funktion

$$\begin{aligned} a_= : \mathcal{R}_{m_1,n_1} \times \mathcal{R}_{m_2,n_2} &\longrightarrow \mathcal{B} \\ (r_1, r_2) &\longmapsto \begin{cases} 1, & \text{falls } r_1 = r_2 \\ 0, & \text{sonst.} \end{cases} \end{aligned}$$

  Die Verknüpfungen `r1 > r2` , `r1 >= r2` , `r1 < r2` , `r1 <= r2` und `r1 <> r2` entsprechen analog konstruierten Funktionen.

Ebenso werden diese Funktionen analog für $z_1 \in \mathcal{I}_{m_1,n_1}$ und $z_2 \in \mathcal{I}_{m_2,n_2}$ definiert.

Verknüpfung boolescher Ausdrücke
Für $b_1, b_2 \in \mathcal{B}$ sind die folgenden booleschen Verknüpfungen definiert:

- Der Verknüpfung `b1 OR b2` entspricht die Funktion

$$\begin{aligned} a_{OR} : \mathcal{B} \times \mathcal{B} &\longrightarrow \mathcal{B} \\ (b_1, b_2) &\longmapsto \begin{cases} 1, & \text{falls } b_1 = 1 \text{ oder } b_2 = 1 \\ 0, & \text{sonst.} \end{cases} \end{aligned}$$

  Die Verknüpfungen `b1 AND b2` und `NOT b1` entsprechen analog konstruierten Funktionen

Typkonvertierung
Zur Typkonvertierung gibt es folgende Funktionen:

- Für $b \in \mathcal{B}$:
  - Dem Befehl `INTEGER(b)` entspricht die folgende Funktion:

$$\begin{aligned} a_{INTEGER} : \mathcal{B} &\longrightarrow \mathcal{I}_{min_Z,max_Z} \\ b &\longmapsto \begin{cases} 0, & \text{falls } b = 0 \\ 1, & \text{sonst.} \end{cases} \end{aligned}$$

  - Dem Befehl `REAL(b)` entspricht die analog konstruierte Funktion.

- Für $z \in \mathcal{I}_{m,n}$:
  - Dem Befehl `BIT(z)` entspricht die folgende Funktion:

$$\begin{aligned} a_{BIT} : \mathcal{I}_{m,n} &\longrightarrow \mathcal{B} \\ z &\longmapsto \begin{cases} 0, & \text{falls } z = 0 \\ 1, & \text{sonst.} \end{cases} \end{aligned}$$

  - Dem Befehl `REAL(z)` entspricht die folgende Funktion:

$$\begin{aligned} a_{REAL} : \mathcal{I}_{m,n} &\longrightarrow \mathcal{R}_{min_R, max_R} \\ z &\longmapsto z. \end{aligned}$$

- Für $r \in \mathcal{R}_{m,n}$:
  - Dem Befehl `BIT(r)` entspricht die folgende Funktion:

$$\begin{aligned} a_{BIT} : \mathcal{R}_{m,n} &\longrightarrow \mathcal{B} \\ r &\longmapsto \begin{cases} 0, & \text{falls } r = 0 \\ 1, & \text{sonst.} \end{cases} \end{aligned}$$

  - Dem Befehl `INTEGER(r)` entspricht die folgende Funktion:

$$\begin{aligned} a_{INTEGER} : \mathcal{R}_{m,n} &\longrightarrow \mathcal{I}_{min_{\mathbf{Z}}, max_{\mathbf{Z}}} \\ r &\longmapsto \lfloor r \rfloor. \end{aligned}$$

Informationen über die Problemstruktur
Aus historischen Gründen taucht in den Funktionen, die die Problemstruktur betreffen, die Abkürzung *rep* auf, die für Repräsentation steht.

Es liegt die Problemstruktur $SP \in \mathcal{A}^l$ zugrunde.

- Dem Befehl `replength()` entspricht die Funktion:

$$a_{replength} \equiv l.$$

- Sei $z \in \mathcal{I}_{1,l}$. Dem Befehl `numberrepbit(z)` entspricht die Funktion:

$$\begin{aligned} a_{numberrepbit} : \mathcal{I}_{1,l} &\longrightarrow \mathcal{I}_{min_{\mathbf{Z}}, max_{\mathbf{Z}}} \\ z &\longmapsto \max\left\{ z' \;\middle|\; \forall j = z, z+1, \cdots, z+z'-1 : SP_j = \mathcal{B} \right\}. \end{aligned}$$

  Den Befehlen `numberrepint(z)`, `numberrepreal(z)` und `numberrepperm(z)` werden analog konstruierte Funktionen zugewiesen.

Informationen über die Kodierungsstruktur
Es liegt die Kodierungsstruktur $SL \in \mathcal{A}^{l'''}\big|_{Kod}$ zugrunde.

- Dem Befehl `length()` entspricht die Funktion:

$$a_{length} \equiv l'''.$$

- Sei $z \in \mathcal{I}_{1,l'''}$. Dem Befehl `numberbit(z)` entspricht die Funktion:

$$\begin{aligned} a_{numberbit} : \mathcal{I}_{1,l'''} &\longrightarrow \mathcal{I}_{min_{\mathbf{Z}},max_{\mathbf{Z}}} \\ z &\longmapsto \max\left\{z' \;\middle|\; \forall j = z, z+1, \cdots, z+z'-1 : SL_j \in \mathcal{B}\right\}. \end{aligned}$$

  Den Befehlen `numberint(z)`, `numberreal(z)` und `numberperm(z)` werden analog konstruierte Funktionen zugewiesen.

Lesen eines Individuums
Sei $I \in \mathcal{IND}$ für $SL \in \mathcal{A}^{l'''}\big|_{Kod}$, $SP|_C \in \hat{\mathcal{A}}^l$.

- Sei $z \in \mathcal{I}_{1,l}$. Dem Befehl `getrepbit(I,z)` entspricht die Funktion:

$$\begin{aligned} a_{getrepbit} : \mathcal{IND} \times \mathcal{I}_{1,l} &\longrightarrow \mathcal{B} \\ (I,z) &\longmapsto \begin{cases} J_z, & \text{falls } a_{numberrepbit}(z) > 0 \\ \text{undef.}, & \text{sonst,} \end{cases} \end{aligned}$$

  mit $I = (S, r)$ und $Decode(S) = (J_1, \cdots, J_l)$

  Den Befehlen `getrepint(I,z)`, `getrepreal(I,z)` und `getrepperm(I,z)` werden analog konstruierte Funktionen zugeordnet.

- Sei $z \in \mathcal{I}_{1,l'''}$. Dem Befehl `getbit(I,z)` entspricht die Funktion:

$$\begin{aligned} a_{getbit} : \mathcal{IND} \times \mathcal{I}_{1,l'''} &\longrightarrow \mathcal{B} \\ (I,z) &\longmapsto \begin{cases} J_z, & \text{falls } a_{numberbit(z)} > 0 \\ \text{undef.}, & \text{sonst,} \end{cases} \end{aligned}$$

  mit $I = (S, r)$ und $S = (J_1, \cdots, J_{l'''})$.

  Den Befehlen `getint(I,z)`, `getreal(I,z)` und `getperm(I,z)` werden analog konstruierte Funktionen zugeordnet.

- Dem Befehl `fitness(I)` entspricht die Funktion:

$$\begin{aligned} a_{fitness} : \mathcal{IND} &\longrightarrow \mathcal{R}_{min_R,max_R} \\ I &\longmapsto r, \end{aligned}$$

  mit $I = (S, r)$.

Lesen einer Population
Sei $P = (I_1, \cdots, I_n) \in \mathcal{POP}$ mit $I_i \in \mathcal{IND}$.

- Dem Befehl `sizeofpop(P)` entspricht die Funktion:

$$\begin{array}{rcl} a_{sizeofpop} : \mathcal{POP} & \longrightarrow & \mathcal{I}_{min_{\mathbf{Z}}, max_{\mathbf{Z}}} \\ P = (I_1, \cdots, I_n) & \longmapsto & n. \end{array}$$

- Dem Befehl `getavgfitness(P)` entspricht die Funktion:

$$\begin{array}{rcl} a_{getavgfitness} : \mathcal{POP} & \longrightarrow & \mathcal{R}_{min_R, max_R} \\ P = (I_1, \cdots, I_n) & \longmapsto & \dfrac{1}{n} \displaystyle\sum_{1 \le i \le n} a_{fitness}(I_i). \end{array}$$

- Dem Befehl `getbestfitness(P)` entspricht die Funktion:

$$\begin{array}{rcl} a_{getbestfitness} : \mathcal{POP} & \longrightarrow & \mathcal{R}_{min_R, max_R} \\ P = (I_1, \cdots, I_n) & \longmapsto & \max \left\{ a_{fitness}(I_i) \;\middle|\; 1 \le i \le n \right\}. \end{array}$$

- Dem Befehl `getworstfitness(P)` entspricht die Funktion:

$$\begin{array}{rcl} a_{getworstfitness} : \mathcal{POP} & \longrightarrow & \mathcal{R}_{min_R, max_R} \\ P = (I_1, \cdots, I_n) & \longmapsto & \min \left\{ a_{fitness}(I_i) \;\middle|\; 1 \le i \le n \right\}. \end{array}$$

- Sei $z \in \mathcal{I}_{1,n}$. Dem Befehl `getind(P,z)` entspricht die Funktion:

$$\begin{array}{rcl} a_{getind} : \mathcal{POP} \times \mathcal{I}_{1,n} & \longrightarrow & \mathcal{IND} \\ (P, z) & \longmapsto & I_k. \end{array}$$

- Dem Befehl `getbest(P)` entspricht die Funktion:

$$\begin{array}{rcl} a_{getbest} : \mathcal{POP} & \longrightarrow & \mathcal{IND} \\ P = (I_1, \cdots, I_n) & \longmapsto & I_i, \end{array}$$

  mit $i = \min \left\{ j \;\middle|\; a_{fitness}(I_j) = getbestfitness(P) \right\}$.

- Dem Befehl `getworst(P)` entspricht die Funktion:

$$\begin{array}{rcl} a_{getworst} : \mathcal{POP} & \longrightarrow & \mathcal{IND} \\ P = (I_1, \cdots, I_n) & \longmapsto & I_i, \end{array}$$

  mit $i = \min \left\{ j \;\middle|\; a_{fitness}(I_j) = getworstfitness(P) \right\}$.

Lesen von Permutationen
Sei $\pi \in \mathcal{P}_n$ eine Permutation und $z \in \mathcal{I}_{1,n}$. Dem Befehl `getpermvalue(p,z)` wird die folgende Funktion zugeordnet:

$$\begin{aligned} a_{getpermvalue} : \mathcal{P}_n \times \mathcal{I}_{1,n} &\longrightarrow \mathcal{I}_{1,n} \\ (\pi, z) &\longmapsto \pi(z). \end{aligned}$$

Lesen des Generationenzählers
Auf den globalen Generationenzähler kann mit dem Befehl `gen()` zugegriffen werden. Dieser Befehl entspricht dann der Funktion:

$$\begin{aligned} a_{gen} : GEN &\longrightarrow GEN \\ gen &\longmapsto gen. \end{aligned}$$

Diese Definition scheint auf den ersten Blick unsinnig zu sein. Da allerdings nachher der Definitionsbereich auf $STATE_i$ erweitert wird, handelt es sich bei dieser Funktion um eine Projektion auf den Wert des Generationenzählers, um ihn in Ausdrücken benutzen zu können.

Zufallszahlen
Sei (Ω, Σ, P) ein Wahrscheinlichkeitsraum.

- Dem Befehl `getrandomreal()` entspricht eine gleichverteilte Zufallsvariable:

$$\begin{aligned} X_{getrandomreal} : \Omega &\longrightarrow \mathcal{R}_{0,1} \\ \omega &\longmapsto p. \end{aligned}$$

- Dem Befehl `getrandomint(i,j)` entspricht die Zufallsvariable

$$\begin{aligned} X_{getrandomint_{i,j}} : \Omega &\longrightarrow \mathcal{I}_{i,j} \\ \omega &\longmapsto p, \end{aligned}$$

$$\text{mit} \quad p := a_{INTEGER}\left(getrandomreal(\omega) * (j - i) + i\right).$$

7.7.2.2 Befehle als Anweisungen

Anweisungen entsprechen Funktionen, die ein Element aus der Arbeitsmenge $STATE_i$ des Operators auf ein neues Element abbilden. Hier wird zunächst jede Funktion als Abbildung von der Menge der betroffenen Variablen in diese definiert.

Wertezuweisungen
Sei `V` eine beliebige Variable mit dem Datentyp $D \in \mathcal{DAT}$.

Falls $D = \mathcal{R}_{m,n}$, entspricht der Zuweisung `V := U` die folgende Funktion:

$$f_{:=} : \mathcal{R}_{m,n} \times \mathcal{DAT} \longrightarrow \mathcal{R}_{m,n} \times \mathcal{DAT}$$
$$(v,u) \longmapsto \begin{cases} (u,u), & \text{falls } u \in \mathcal{R}_{m,n} \\ \text{undef.}, & \text{sonst.} \end{cases}$$

Analoge Funktionen werden für $D = \mathcal{I}_{m,n},\ \mathcal{B},\ \mathcal{P}_n,\ \mathcal{IND},\ \mathcal{POP}$ definiert.

Veränderung der Individuen
Sei $I \in \mathcal{IND}$ für $SL \in \mathcal{A}^{l'''}\big|_{Kod}$, $SP|_C \in \hat{\mathcal{A}}^l$.

- Sei $z \in \mathcal{I}_{1,l}$, $b \in \mathcal{B}$. Der Befehl `setrepbit(I,z,b)` entspricht der folgenden Funktion:

$$f_{setrepbit} : \mathcal{IND} \times \mathcal{I}_{1,l} \times \mathcal{B} \longrightarrow \mathcal{IND} \times \mathcal{I}_{1,l} \times \mathcal{B}$$
$$(I,z,b) \longmapsto \begin{cases} (I',z,b), & \text{falls } a_{numrepbit}(I,z) > 0 \\ \text{undef.}, & \text{sonst,} \end{cases}$$

$$\text{mit} \quad I' = (S',r), I = (S,r) \text{ und } Decode(S) = (J_1,\ldots,J_l),$$
$$\text{wobei} \quad Decode(S') = (J'_1,\ldots,J'_l) \text{ und } J'_i := \begin{cases} b, & \text{falls } i = z \\ J_i, & \text{sonst.} \end{cases}$$

  Für $r \in \mathcal{R}_{m,n}$, $z' \in \mathcal{I}_{m,n}$ und $\pi \in \mathcal{P}_n$ werden analog die Funktionen für die Befehle `setrepreal(I,z,r)`, `setrepint(I,z,z')` und `setrepperm(I,z,p)` definiert.

- Sei $z \in \mathcal{I}_{1,l'''}$, $b \in \mathcal{B}$. Der Befehl `setbit(I,z,b)` entspricht der folgenden Funktion:

$$f_{setbit} : \mathcal{IND} \times \mathcal{I}_{1,l'''} \times \mathcal{B} \longrightarrow \mathcal{IND} \times \mathcal{I}_{1,l'''} \times \mathcal{B}$$
$$(I,z,b) \longmapsto \begin{cases} (I',z,b), & \text{falls } a_{numbit}(I,z) > 0 \\ \text{undef.}, & \text{sonst.} \end{cases}$$

$$\text{mit} \quad I' = (S',r), I = (S,r) \text{ und } S = (J_1,\ldots,J_{l'''}),$$
$$\text{wobei} \quad S' = (J'_1,\ldots,J'_{l'''}) \text{ und } J'_i := \begin{cases} b, & \text{falls } i = z \\ J_i, & \text{sonst.} \end{cases}$$

  Für $r \in \mathcal{R}_{m,n}$, $z' \in \mathcal{I}_{m,n}$ und $\pi \in \mathcal{P}_n$ werden analog die Funktionen für die Befehle `setreal(I,z,r)`, `setint(I,z,z')` und `setperm(I,z,p)` definiert.

- Der Befehl `evaluate(I)` entspricht der folgenden Funktion:

$$\begin{aligned} f_{evaluate} : \mathcal{IND} &\longrightarrow \mathcal{IND} \\ I &\longmapsto F_{fitness}(I). \end{aligned}$$

$F_{fitness}$ ist dabei eine vom Benutzer mittels des Operators $OP_{fitness} = OP_0$ definierte Funktion für die Fitneßfunktion.

Veränderung von Populationen
Seien $P_1, P_2 \in \mathcal{POP}$ mit $P_1 = (I_1, \ldots, I_{n_1}), P_2 = (I'_1, \ldots, I'_{n_2})$.

- Der Befehl `clearpop(P1)` entspricht der folgenden Funktion:

$$\begin{aligned} f_{clearpop} : \mathcal{POP} &\longrightarrow \mathcal{POP} \\ P_1 &\longmapsto P = \emptyset \end{aligned}$$

- Der Befehl `mergepop(P1,P2)` entspricht der folgenden Funktion:

$$\begin{aligned} f_{mergepop} : \mathcal{POP} \times \mathcal{POP} &\longrightarrow \mathcal{POP} \times \mathcal{POP} \\ (P_1, P_2) &\longmapsto (P, P_2), \end{aligned}$$

mit $P := (I_1, \ldots, I_{n_1}, I'_1, \ldots, I'_{n_2})$.

- Sei $z \in \mathcal{I}_{1,n_1+1}, I \in \mathcal{IND}$. Der Befehl `insertind(P1,I,z)` entspricht dann der folgenden Funktion:

$$\begin{aligned} f_{insert} : \mathcal{POP} \times \mathcal{IND} \times \mathcal{I}_{1,n_1+1} &\longrightarrow \mathcal{POP} \times \mathcal{IND} \times \mathcal{I}_{1,n_1+1} \\ (P_1, I, z) &\longmapsto (P, I, z), \end{aligned}$$

$$\text{wobei} \quad P = (I''_1, \ldots, I''_{n_1+1}) \text{ mit } I''_i := \begin{cases} I, & \text{für } i = z \\ I_i, & \text{für } i < z \\ I_{i-1}, & \text{für } i > z. \end{cases}$$

- Sei $z \in \mathcal{I}_{1,n_1}$. Dann entspricht der Befehl `killinpop(P1,z)` der folgenden Funktion:

$$\begin{aligned} f_{killinpop} : \mathcal{POP} \times \mathcal{I}_{1,n_1} &\longrightarrow \mathcal{POP} \times \mathcal{I}_{1,n_1} \\ (P_1, z) &\longmapsto (P, z), \end{aligned}$$

wobei $P = (I''_1, \ldots, I''_{n_1-1})$ mit $I''_i := \begin{cases} I_i, & \text{für } i < z \\ I_{i+1}, & \text{für } i \geq z. \end{cases}$

- Der Befehl `evaluate(P1)` entspricht der folgenden Funktion:

$$\begin{aligned} f_{evaluate} : \mathcal{POP} &\longrightarrow \mathcal{POP} \\ P_1 &\longmapsto P = (I''_1, \ldots, I''_{n_1}), \end{aligned}$$

mit $I''_i := f_{evaluate}(I_i)$.

Veränderung von Permutationen
Sei $\pi = (\pi(1), \ldots, \pi(n)) \in \mathcal{P}_n$, $z_1, z_2 \in \mathcal{I}_{1,n}$.

- Der Befehl `setpermvalue(P,z1,z2)` entspricht der folgenden Funktion:

$$\begin{aligned} f_{setpermvalue} : \mathcal{P}_n \times \mathcal{I}_{1,n} \times \mathcal{I}_{1,n} &\longrightarrow \mathcal{P}_n \times \mathcal{I}_{1,n} \times \mathcal{I}_{1,n} \\ (\pi, z_1, z_2) &\longmapsto (\pi', z_1, z_2), \end{aligned}$$

$$\text{wobei} \quad \pi' = (\pi'(1), \ldots, \pi'(n))$$

$$\text{mit} \quad \pi'(i) := \begin{cases} \pi(z_2), & \text{für } i = z_1 \\ \pi(i), & \text{falls } i < z_2 < z_1 \\ & \text{oder } i < z_1 < z_2 \\ & \text{oder } z_1 < z_2 < i \\ & \text{oder } z_2 < z_1 < i \\ & \text{oder } i \neq z_1 = z_2 \\ \pi(i+1), & \text{falls } z_2 \leq i < z_1 \\ \pi(i-1), & \text{falls } z_1 < i \leq z_2. \end{cases}$$

- Der Befehl `xchangeperm(P,z1,z2)` entspricht der folgenden Funktion:

$$\begin{aligned} f_{xchangeperm} : \mathcal{P}_n \times \mathcal{I}_{1,n} \times \mathcal{I}_{1,n} &\longrightarrow \mathcal{P}_n \times \mathcal{I}_{1,n} \times \mathcal{I}_{1,n} \\ (\pi, z_1, z_2) &\longmapsto (\pi', z_1, z_2), \end{aligned}$$

$$\text{wobei} \quad \pi' = (\pi'(1), \ldots, \pi'(n))$$

$$\text{mit} \quad \pi'(i) := \begin{cases} \pi(i), & \text{falls } i \neq z_1 \\ & \text{und } i \neq z_2 \\ \pi(z_1), & \text{falls } i = z_2 \\ \pi(z_2), & \text{falls } i = z_1. \end{cases}$$

- Der Befehl `reverseperm(P,z1,z2)` entspricht der folgenden Funktion:

$$\begin{aligned} f_{reverseperm} : \mathcal{P}_n \times \mathcal{I}_{1,n} \times \mathcal{I}_{1,n} &\longrightarrow \mathcal{P}_n \times \mathcal{I}_{1,n} \times \mathcal{I}_{1,n} \\ (\pi, z_1, z_2) &\longmapsto (\pi', z_1, z_2), \end{aligned}$$

$$\text{wobei} \quad \pi' = (\pi'(1), \ldots, \pi'(n))$$

$$\text{mit} \quad \pi'(i) := \begin{cases} \pi(i), & \text{falls } i < z_1 < z_2, \\ & \text{oder } i < z_2 < z_1, \\ & \text{oder } z_1 < z_2 < i, \\ & \text{oder } z_2 < z_1 < i, \\ & \text{oder } z_1 = z_2 \\ \pi(z_1 + z_2 - i), & \text{sonst.} \end{cases}$$

Veränderung des Generationenzählers
Der globale Generationenzähler kann mit dem Befehl `incgen()` erhöht werden. Dies entspricht dann der folgenden Funktion:

$$\begin{aligned} f_{incgen} : GEN &\longrightarrow GEN \\ gen &\longmapsto \begin{cases} gen+1, & \text{falls } gen < max_Z \\ \text{undef.}, & \text{sonst.} \end{cases} \end{aligned}$$

7.7.2.3 Befehle, welche die berechnete Funktion nicht beeinflussen

Filter
Entspreche der Befehl `FILTER V` dem Filter $filter_{i,j} \in FILTER_i$ im Operator Op_i, mit

$$\texttt{V} = V_n \in \mathcal{DAT},\ 1 \leq n \leq l_{STATE_i} \text{ und } STATE_i = (V_1, \ldots, V_{l_{STATE_i}}).$$

So repräsentiert der Befehl `FILTER V` die folgende Funktion:

- Falls $FSTREAM_i(filter_{i,j}) \neq NOSTREAM$, gilt:
$$\begin{aligned} f_{filter_{i,j}} : FILVAL_k \times (V_1, \ldots, V_{l_{STATE_i}}) &\longrightarrow FILVAL_k \\ \Big(filval_k, (v_1, \ldots, v_n, \ldots, v_{l_{STATE_i}})\Big) &\longmapsto filval'_k, \end{aligned}$$
mit $FILVAL_k = FSTREAM_i(filter_{i,j})$ und $filval'_k = filval_k \circ v_n$.
- Anderenfalls gilt:
$$f_{filter_{i,j}} \equiv id.$$

Bildschirmausgabe
Einem Befehl der Form `WRITE V` bzw. `WRITE "..."`, mit

$$\texttt{V} = V_n \in \mathcal{DAT},\ 1 \leq n \leq l_{STATE_i} \text{ und } STATE_i = (V_1, \ldots, V_{l_{STATE_i}}),$$

entsprechen folgende Funktionen:

- $$\begin{aligned} f_{write} : SCRN \times (V_1, \ldots, V_{l_{STATE_i}}) &\longrightarrow SCRN \\ (e, (v_1, \ldots, v_n, \ldots, v_{l_{STATE_i}})) &\longmapsto e \circ Alph(v_n), \end{aligned}$$
wobei $Alph : V \longrightarrow \Sigma^*$ den Wert von V im Alphabet ausdrücke.
- $$\begin{aligned} f_{write} : SCRN \times \Sigma^* &\longrightarrow SCRN \\ (e, d) &\longmapsto e \circ d. \end{aligned}$$

7.7.2.4 Label

Label haben ebenfalls keinen direkten Einfluß auf $STATE_i$, nur kann mit Hilfe der Label die Berechnung des Ergebnisses abgebrochen werden.

- Im Zustand $LABSTAT_i(label_{i,j}) = Halt$

  wird die Berechnung geordnet beim Erreichen des Label gestoppt. Geordnet heißt dabei, daß die Berechnung von diesem Punkt aus direkt wieder aufgenommen werden kann.

  $$f_{label_{i,j}} = \begin{cases} id, & \text{falls } valid = NORMAL \\ STOP, & \text{sonst.} \end{cases}$$

- Der Zustand $LABSTAT_i(label_{i,j}) = Abbruch$

  sollte vom Benutzer nur für solche Label verwendet werden, die von einer geordneten Berechnung nicht berührt werden sollen, um Ausnahmefälle auffangen zu können. Beim Erreichen eines so gesetzten Label bricht die Berechnung ab.

  $$f_{label_{i,j}} = STOP.$$

- Falls $LABSTAT_i(label_{i,j}) = Inaktiv$

  ist der Filter $filter_{i,j}$, wie der Name schon sagt, nicht aktiv und es gilt:

  $$f_{label_{i,j}} = id.$$

7.7.3 Ablaufkontrollkonstrukte

Es stehen in LEA die folgenden Ablaufkontrollkonstrukte zur Verfügung.

IF–THEN–ELSE–Verzweigung
Es sei die folgende Anweisung $s :=$ `IF a THEN s1 ELSE s2 END` in LEA gegeben. Dabei repräsentieren `a`, `s1` und `s2` folgende Funktionen

$$a : STATE_i \longrightarrow \mathcal{B},$$

$$f_{s_1}, f_{s_2} : STATE_i \times FILVAL \times SCRN \longrightarrow STATE_i \times FILVAL \times SCRN.$$

Dann entspricht `s` der folgenden Funktion:

$$\begin{aligned} f_s : STATE_i \times FILVAL \times SCRN &\longrightarrow STATE_i \times FILVAL \times SCRN \\ (state_i, filval, scrn) &\longmapsto (state', filval', scrn'), \end{aligned}$$

$$\text{mit} \quad (state', filval', scrn') := \begin{cases} f_{s_1}(state_i, f, s), & \text{falls } a(state_i) = 1 \\ f_{s_2}(state_i, f, s), & \text{sonst.} \end{cases}$$

Ist $s :=$ `IF a THEN s1 END`, wird in der obigen Definition $f_{s_2} \equiv id$ gesetzt.

WHILE–Schleife
Es sei die folgende Anweisung s :=`WHILE a DO s1 END` in LEA gegeben. Dabei repräsentieren `a` und `s1` folgende Funktionen:

$$a : STATE_i \longrightarrow \mathcal{B},$$

$$f_{s_1} : STATE_i \times FILVAL \times SCRN \longrightarrow STATE_i \times FILVAL \times SCRN.$$

Dann ist s Lösung der folgenden Fixpunktgleichung:
$R =$ `IF a THEN s1; R END`.

Zu einer solchen Funktion läßt sich gemäß dem Fixpunkttheorem (Kleene) (mindestens) eine Lösung

$$f_s : STATE_i \times FILVAL \times SCRN \longrightarrow STATE_i \times FILVAL \times SCRN.$$

finden.

FOR–Schleife
Es sei die folgende Anweisung $s :=$ `FOR V := i1 TO i2 DO s1 END` in LEA gegeben. Sei $i_1, i_2 \in \mathcal{I}_{min_{\mathbf{Z}}, max_{\mathbf{Z}}}$ und $V := V_j \in \mathcal{I}$ eine Variable wobei $STATE_i = (V_1, \ldots, V_{l_{STATE_i}})$, $1 \leq j \leq l_{STATE_i}$. `s1` repräsentiere eine Funktion

$$f_{s_1} : STATE_i \times FILVAL \times SCRN \longrightarrow STATE_i \times FILVAL \times SCRN.$$

Dann läßt sich s umformen zu

$s =$
```
v := i1;
WHILE v <= i2 DO
  s1; V := V + 1
END;
```

FOREACH–Schleife
Es sei die folgende Anweisung $s :=$ `FOREACH Vj IN Vk DO s1 END` in LEA gegeben. Seien $V_j \in \mathcal{IND}$ und $V_k \in \mathcal{POP}$ Variablen, wobei $STATE_i = (V_1, \ldots, V_{l_{STATE_i}})$, $1 \leq j, k \leq l_{STATE_i}$. `s1` repräsentiere eine Funktion

$$f_{s_1} : STATE_i \times FILVAL \times SCRN \longrightarrow STATE_i \times FILVAL \times SCRN.$$

Dann läßt sich s umformen zu

```
s =  Vx := sizeof(Vk);
     FOR Vy := 1 TO Vx DO
       Vj := getind(Vk,Vy); s1
     END;
```

wobei jetzt die Menge der Variablen $VAR'_i = (V_1, \ldots, V_{n+2})$ sei, mit $V_{n+1} = V_x$ und $V_{n+2} = V_y$.

7.7.4 Repräsentierte Funktion des Operators

Hier soll nun die Funktion F_i angegeben werden, die durch den gesamten Operator Op_i beschrieben wird. Hierfür werden die in Abschnitt 7.7.2 eingeführten Funktionen auf dem gesamten Wertebereich $STATE_i$ des Operators Op_i fortgesetzt.

Der Operator Op_i habe den Operatortext

$$OP_i = s_1; s_2; \ldots; s_m \in \Sigma^*,$$

wobei die letzte Anweisung $s_m =$ `RETURN V` sei, für ein `V` $= V_j \in \mathcal{DAT}$, mit $1 \leq j \leq l_{STATE_i}$, und

$$STATE_i = (V_1, \ldots, V_{l_{STATE_i}}) \quad \text{und} \quad RETURN_i = V_j.$$

Werde jetzt der Operator Op_i mit $(callval_i, callref_i, gen, f, s) \in INPUT_i$ aufgerufen.

$$INPUT_i := CALLVAL_i \times CALLREF_i \times GEN \times FILVAL \times SCRN.$$

Dann ergibt sich der Anfangszustand $state_i$ des Operators Op_i für die Berechnung

$$state_i := (var_i, statvar_i, callval_i, callref_i, gen) \in STATE_i,$$

wobei vor dem ersten Operatoraufruf $statvar_i := statvar_i^*$ die Default–Belegung der statischen Variablen ist. Die Anfangsbelegung der Variablen var_i ist beim Aufruf des Operators Op_i stets die Defaultbelegung $var_i := var_i^*$.

Die Funktion des Operators Op_i wird durch Nacheinanderausführen der verschiedenen durch die Statements s_i repräsentierten Funktionen f_i, $1 \leq i \leq m$, definiert:

$$\begin{aligned} F_{intern,i} : STATE_i \times FILVAL \times SCRN &\longrightarrow STATE_i \times FILVAL \times SCRN \\ (state_i, filval, scrn) &\longmapsto (state'_i, filval', scrn'), \end{aligned}$$

$$\text{mit} \quad (state_i', filval', scrn') := f_m \circ f_{m-1} \circ \ldots \circ f_1(state_i, filval, scrn)$$

$$\text{und} \quad state_i' := (var_i', statvar_i', callval_i', calref_i', gen').$$

Die nach außen repräsentierte Funktion ergibt sich dann durch die folgende Funktion:

$$\begin{aligned} F_i : INPUT_i &\longrightarrow OUTPUT_i \\ (callval_i, callref_i, gen, filval, scrn) &\longmapsto (callval_i', gen', filval', scrn'), \end{aligned}$$

wobei $OUTPUT_i := CALLREF_i \times GEN \times FILVAL \times SCRN$.

7.8 Experiment

Nach der Eingabe der Laufinitialisierung wird das Ergebnis des Experiments berechnet.

Bei der Laufinitialisierung wird neben den Eingaben, die die einzelnen Operatoren initialisieren, noch die Eingabe der Anfangspopulation $P_0 \in \mathcal{POP}$ erwartet.

Folgende Ausgaben werden dann berechnet:

- ein bestes Individuum $bestind \in \mathcal{IND}$
- eine Endpopulation $P_{end} \in \mathcal{POP}$
- die gefilterten Werte

$$\begin{aligned} &filval \in FILVAL := (FILVAL_1, \ldots, FILVAL_{l_{STREAMS}}) \\ &\text{mit} \quad FILVAL_j := \left\{ filval_j \;\middle|\; filval_j \in \mathcal{DAT}^* \right\}, \quad 1 \leq j \leq l_{STREAMS}. \end{aligned}$$

- die Bildschirmausgabe $scrn \in SCRN := \hat{\Sigma}^*$.

Dies geschieht mit der Funktion F_1, die dem Hauptoperator OP_1 entspricht. F_1 hat die folgende Form:

$$F_1 : INPUT_1 \longrightarrow OUTPUT_1$$

mit $INPUT_1 := (\mathcal{POP}) \times GEN \times FILVAL \times SCRN$
und $OUTPUT_1 := (\mathcal{POP}) \times \mathcal{IND} \times GEN \times FILVAL \times SCRN$.

Die Anfangsbelegung bei einem Experiment sei die folgende:

$$\begin{aligned} filval &:= \mathcal{DAT}^0 \\ scrn &:= \hat{\Sigma}^0 \\ gen &:= 1 \end{aligned}$$

Dann ergibt sich folgende Funktion EA_{F_1}:

$$\begin{aligned} EA_{F_1} : \mathcal{POP} &\longrightarrow \mathcal{IND} \times \mathcal{POP} \times FILVAL \times SCRN \\ P_0 &\longmapsto (IND, P_{end}, filval', scrn') \end{aligned}$$

mit $F_1(P_0, gen, filval, scrn) = (P_{end}, IND, gen', filval', scrn')$.

Kapitel 8

Kritik an EAGLE

Dieser Abschnitt sollte sich eigentlich mit dem fertigen Software-Produkt EAGLE auseinandersetzen. Dies gestaltet sich allerdings schwierig, da es noch kein fertiges Produkt gibt. Daher kann eine Kritik ausschließlich auf den Entscheidungen bis zum Beginn der Implementation und den danach noch aufgetretenen Problemen während der Teilimplementation beruhen. Diese Kritik gliedert sich folgendermaßen: zunächst wird im ersten Teilabschnitt der derzeitige Entwurf des Produkts mit den Ansprüchen, die wir hatten, verglichen. Im zweiten Teilabschnitt werden verschiedene Eigenschaften betrachtet, die wir wohl gerne an EAGLE sehen würden, die allerdings frühzeitig aus Gründen der Machbarkeit fallengelassen wurden. Abschließend soll noch eine kurze „visionäre“ Weiterentwicklung des Konzepts hinter EAGLE betrachtet werden.

8.1 Was wurde aus unseren Ansprüchen?

Unsere Zielsetzung war zwar anfangs, ein möglichst gutes System zu schreiben, das nahezu alle positiven Eigenschaften, die uns in den Sinn kamen, in sich vereinbart. Im Entwurfsprozeß hat es sich allerdings relativ früh abgezeichnet, daß ein solches System nicht im Rahmen einer Projektgruppe geschrieben werden kann. Daher wandelte sich unsere Zielsetzung bald zugunsten eines Forschungssystems. Mit diesem sollten EA schnell entwickelt und analysiert werden können. Die Alternative wäre ein System gewesen, mit dem große, reale Probleme tatsächlich gelöst werden könnten. In diesem Abschnitt soll allerdings das entstehende Produkt mit der Zielsetzung des Forschungssystems verglichen werden.

Positiv fällt am geplanten Produkt zunächst auf, daß durch das „Baukastenprinzip“ die verschiedenen Operatoren schnell und leicht wiederverwendet werden können. Dadurch können auf einfache Art und Weise neue EA aus bekannten Operatoren zu-

sammengebaut werden. Ob das gefundene Konzept gut genug ist oder ob es einfachere Möglichkeiten gibt, dieses Prinzip umzusetzen, soll hier nicht diskutiert werden.

Durch die entwickelte Sprache LEA ist es zudem möglich, neue Crossover- und Mutationsoperatoren zu schreiben und auszuprobieren, die ganz anders aussehen als die bekannten EA-Operatoren.

Weiter können durch die gefundene Trennung von Problemstruktur und Kodierung sehr einfach verschiedene Modelle zur Lösung eines Problems miteinander verglichen werden, z. B. die Evolutionsstrategie mit dem Genetischen Algorithmus.

Eines unserer wichtigsten Ziele war allerdings Interaktion, also die Möglichkeit, jederzeit in den Ablauf eines Experiments einzugreifen und nahezu beliebige Änderungen vorzunehmen. Hier sind dann auch die ersten Abstriche zu machen. Es ist zwar möglich, während des Laufs die Parameter zu ändern, allerdings eine Population oder ein Individuum kann nicht von EAGLE aus geändert werden. Auch bestehen nahezu keine Möglichkeiten, während eines Experiments (online) detaillierte Informationen über die Entwicklungen in der Population zu bekommen. Derzeit bestehen darüberhinaus auch keine Möglichkeiten, nach einem Experiment eine genaue Analyse durchzuführen. Es ist zwar für diesen Zweck ein Analyse-Tool vorgesehen, aus Zeitgründen war es uns jedoch noch möglich zu spezifizieren, was dieses Tool alles leisten soll. Daher ist auch noch unklar, welche Daten hier von EAGLE benötigt werden und in welcher (eindeutigen) Form sie dem Analyse-Tool zur Verfügung gestellt werden. Hier werden noch Formate und Konventionen zur Speicherung der Daten benötigt. In diesem Zusammenhang wird leider auch deutlich, daß unsere bisherigen Möglichkeiten zur Filterung von Daten und ihrer Weiterverarbeitung in EAGLE etwas eingeschränkt sind. Ein neues, umfassenderes Konzept zur Filterung sollte noch entwickelt werden, damit eine verbesserte Analyse der Experimente möglich ist. Eine Erweiterung im Bereich der online-Analyse ist zwar mit dem bisherigen Konzept möglich, würde allerdings größere Änderungen in verschiedenen Teilen des Programms nach sich ziehen.

Daher dürften die Einsatzmöglichkeiten unseres Produkts im Bereich des Forschungssystems wohl etwas dürftig ausfallen. Wir denken allerdings, daß mit geringen Änderungen im Bereich der Interaktivität und einer Standard-Online-Analyse das Programm nahezu optimale Möglichkeiten bieten könnte, spielerisch die Funktionsweisen von EA kennenzulernen. Wir haben dies schließlich in unserer Namensgebung berücksichtigt (Evolutionary Algorithms GAMING and LEARNING Environment).

Zu den derzeit vorhandenen Datenstrukturen sind folgende Punkte anzumerken: Zum einen wurde zwar die Datenstruktur „Permutation“ eingeführt, jedoch wurden nur eher triviale Möglichkeiten ihrer Bearbeitung vorgesehen. Hier wären weitere Konzepte bezüglich der Kodierung und der Manipulation zu untersuchen (Erkennung von Zyklen u. ä.). Erst dann wäre wohl ein vernünftiger Einsatz der Datenstruktur möglich.
Ebenso wurde durch den Verzicht auf Records und Zeigerstrukturen bzw. Listen

in LEA ein weiterer Einsatz z. B. im Bereich der Evolutionären Programmierung unmöglich gemacht. Hier sind wohl auch kaum Erweiterungsmöglichkeiten denkbar.

Zu Änderungen am irgendwann hoffentlich fertigen Source-Code soll nur soviel gesagt werden: Änderungen, die einzelne Klassen betreffen, könnten problemlos durchgeführt werden. Ein größerer Aufwand wäre notwendig, wenn die Oberfläche geändert werden soll. Auch von größeren Änderungen an der Grammatik von LEA wird eher abgeraten, da sich hier vermutlich viele Änderungen im Simulator ergeben. Hier wäre u. U. ein Tcl-basierter Simulator wartungsfreundlicher gewesen (siehe auch Abschnitt 9.1.2.2).

8.2 Was fehlt?

In diesem Teilabschnitt sollen nochmals die getroffenen Entscheidungen dahingehend hinterfragt werden, inwieweit sie unser Produkt u. U. beschränkt haben. Da diese Entscheidungen äußerst früh gefällt wurden, sind sie hier nicht Grundlage für eine Kritik am Produkt, sondern dienen nur dazu zu überlegen, inwieweit andere Eigenschaften begrüßenswert gewesen wären.

Beim Entwurf des Systems wurden Fragen der Effizienz nicht besonders berücksichtigt. Dies wurde allerdings bewußt in Kauf genommen, als die Entscheidung für einen Interpreter zur Abarbeitung der EA gefällt wurde. Geschwindigkeit sehen wir nicht als das Hauptmerkmal eines Forschungssystems.

Der Verzicht auf den Anschluß externer Prozesse war eine der Entscheidungen, die wir im Rückblick bedauern. Da LEA die einzige Möglichkeit ist, Operatoren und damit auch Fitneßfunktionen zu formulieren, können hier weder kompliziertere Funktionen untersucht werden (aus Geschwindigkeitsgründen) noch Fitneßfunktionen, die sich aufgrund physikalischer Einflüsse nicht in LEA formulieren lassen. Der Verzicht auf die externen Prozesse schränkt zwar ein Forschungssystem nicht zwingend ein, stellt allerdings eine Anwendung in einigen Teilbereichen der Physik in Frage.

Anzumerken bleibt die fehlende Berücksichtigung paralleler Modelle (coarse-grained bzw. fine-grained) sowohl in der Grundstruktur von EAGLE als auch im Sprachentwurf von LEA. Ebenso sind keine Metaverfahren vorgesehen, die als Individuen einzelne verschiedene EA haben, welche dann auf ein Problem angesetzt werden, um ein „gutes“ Verfahren für dieses Problem zu finden. Der Verzicht auf diese parallelen Modelle ist in einem Forschungssystem ungünstig, war jedoch für EAGLE eine richtige Entscheidung, um die Arbeit des Entwurfs und der Implementierung zu erleichtern.

Ebensowenig wie Parallelität auf der Modellebene gibt es eine implizite oder explizite Verwendung von parallelen Rechnern zur Bearbeitung eines Algorithmus'. Es

wurden bisher noch keine Überlegungen angestellt, in welcher Form parallele Rechner in EAGLE verwendet werden können. Dies ist insofern nachvollziehbar, als sich von den Teilnehmern der Projektgruppe niemand mit diesem Thema auskannte und die Einarbeitung und vernünftige Anwendung im System vermutlich zuviel Zeit in Anspruch genommen hätte.

Insgesamt dürfte der Aufbau von EAGLE vermutlich etwas zu statisch sein: es gibt nur ein Problem, eine Population, alle Individuen haben dieselbe Kodierung. Hier wären wesentlich flexiblere Ansätze denkbar, mit denen dann auch leicht Parallelität in EA und ähnliches betrachtet und untersucht werden könnte.

8.3 Wie könnte eine Weiterentwicklung von EAGLE aussehen?

Da hier kein vollständiges Konzept entwickelt und vorgestellt werden kann, soll im folgenden nur knapp ein möglicher Ansatz skizziert werden. Neben den hier angerissenen Möglichkeiten bestehen naturgemäß auch viele andere Ansätze. Zwischen den verschiedenen Ideen fand keine Abwägung statt, so daß auch kein Anspruch auf eine „beste“ Weiterentwicklung erhoben wird.

In einer Weiterentwicklung von EAGLE sollten die grundsätzlichen Merkmale von EAGLE beibehalten werden. Das wären im einzelnen:

- die grundsätzliche Aufteilung in Problem, Kodierung und den eigentlichen Evolutionären Algorithmus
- das „Baukastenprinzip“ für die Operatoren
- die Syntax und Semantik von LEA

Zusätzlich sollten die folgenden Details in einer neueren Version von EAGLE berücksichtigt werden:

- verbesserte Analysemöglichkeiten, insbesondere während eines Experiments
- Möglichkeit zur Formulierung und Untersuchung paralleler Modelle
- Möglichkeit zur Formulierung von allgemeinen Metaverfahren

Analysemöglichkeiten: Während eines Experiments sollten möglichst viele verschiedene Details über die Entwicklung des EA und über die Population(en) anzeigbar sein. Zum einen gibt es verschiedene Performance-Maße, die mittels

einer Grafik visualisiert werden könnten. Außerdem sind noch die verschiedensten anderen (grafischen) Ausgaben denkbar, die vom jeweiligen Problem und Verfahren abhängen. Hier sollte es dem Benutzer möglich sein, beliebige anzeigbare Funktionen zu formulieren, die sich aus den jeweiligen Populationen und der Entwicklung des Verfahrens berechnen lassen.

Zudem ist es auch notwendig, daß von EAGLE aus nach und während eines Experiments die Populationen und Individuen genauer untersucht werden können. Zusätzlich sollte auf die gefilterten Daten zugegriffen werden können. Damit eine vernünftige Verarbeitung der Daten möglich ist, sollte das Konzept, wie Daten gefiltert und gespeichert werden, nochmals überarbeitet werden.

Parallele Modelle: Wenn parallele Modelle bearbeitet werden sollen, reicht nicht mehr nur eine Population aus, sondern es sollen beliebige Mengen von Populationen (u. U. auch mit Nachbarschaftsbeziehungen) bearbeitet werden können. Dabei sind z. B. die folgenden Fälle denkbar: verschiedenen Populationen sind verschiedene Fitneßfunktionen zugeordnet, oder die Individuen einer Population haben eine andere Kodierung wie die einer anderen (z. B. wegen eines anderen EA oder einer anderen Fitneßfunktion). Hier ergeben sich natürlich sofort verschiedene Probleme, z. B. wie ein Individuenaustausch bei verschiedenen Kodierung aussehen kann.

Daher bietet es sich an, statt der bisherigen, eher statischen Definition von Experimenten einen „objektorientierten Ansatz“ zu verfolgen. Die Idee ist nun, am Anfang nicht nur *eine* Anfangspopulation zu haben, sondern einen ganzen Pool von mehreren Populationen.

Bisher wurde die Anfangspopulation beschrieben, indem die Fitneßfunktion, die Kodierung und der Hauptoperator zur Veränderung der Population angegeben wurde. Eine Möglichkeit wäre, daß Populationen zunächst abstrakt beschrieben werden, indem ihnen eine zu optimierende Fitneßfunktion, eine Kodierung und die auf dieser Population und ihren Individuen möglichen Operatoren zugeordnet werden. Nun ließe sich ein Verfahren dadurch formulieren, daß in einem Hauptoperator die verschiedenen Populationen miteinander verknüpft werden.

Der Hauptoperator gibt dabei an, wie das Verfahren ablaufen soll, d. h. in welcher Reihenfolge welche Operatoren bei welchen Populationen aufgerufen werden sollen. Hier wird auch das Abbruchkriterium programmiert. Hierzu wären wohl kleinen Änderungen in LEA notwendig.

Es könnten folgende Operatorentypen unterschieden werden:

- Operatoren, die auf einer gesamten Population arbeiten, also z. B. ein Operator, der die Population um eine Generation altert.
- Operatoren, die auf Individuen in der Population arbeiten
- Operatoren, die Individuen von einer Population in eine andere schicken. Dabei ist insbesondere zu beachten, daß der Zielpopulation eine andere

Kodierung zugeordnet sein kann. Hierzu sind geringfügige Änderungen in LEA notwendig.

- „new“-Operator, der bei jeder Erzeugung eines neuen Individuums in der Population aufgerufen wird.

Metaverfahren: Bei einem Metaverfahren wird versucht, aus einer Klasse von verschiedenen Evolutionären Algorithmen denjenigen herauszufinden, der eine oder mehrere Fitneßfunktionen besonders gut optimieren kann. Dabei bietet es sich an, diese Suche mit einem anderen Evolutionären Verfahren zu realisieren. Davon ausgehend sind natürlich auch noch andere Verfahren denkbar, bei denen verschiedene Verfahren miteinander arbeiten.

Dies läßt sich zum Beispiel so realisieren, indem in einer in LEA formulierten Fitneßfunktion eine neue Population erzeugt wird, mit der dann versucht wird, die Güte festzustellen, mit der ein EA aus dieser Anfangspopulation eine Zielfunktion optimieren kann. Daraus könnte dann die Fitneß des zu bewertenden EA berechnet werden.

Die Einzelheiten von „EAGLEobject“ müssen noch ausgearbeitet werden. Vermutlich ergeben sich dabei weitere Probleme, die zu lösen sind. Daraus ergeben sich vermutlich verschiedene Änderungen insbesondere der Oberfläche und des Interpreters. Hier sollte nur kurz demonstriert werden, daß sich das bisherige Konzept verhältnismäßig leicht zu einem wesentlich mächtigeren Produkt erweitern läßt.

Kapitel 9

Rückblick

Es ist von Vorteil, die Fehler,
aus denen man lernen kann,
recht frühzeitig zu machen.
(Winston Churchill)

9.1 Chronik der Projektgruppe

Wozu eine Chronik? Wir, die erste Projektgruppe an der Fakultät Informatik der Universität Stuttgart, wollen an dieser Stelle im Rückblick unsere Arbeit analysieren. Erstens haben wir selbst aus unseren Fehlern eine ganze Menge gelernt (wir werden sie in Zukunft zu vermeiden suchen). Zweitens werden Ergebnisse der Projektgruppe leichter verständlich, wenn bekannt ist, wie sie zustandegekommen sind. Drittens hat ein solcher Rückblick für die Beteiligt-Gewesenen einen gewissen Unterhaltungswert. Und viertens hoffen wir, den uns nachfolgenden Projektgruppen damit einen Wegweiser geben zu können, der ihnen manche Sackgasse ersparen kann, in die wir geraten sind. Allerdings sind wir uns durchaus bewußt, daß zu Beginn einer Projektgruppe natürlich alles gut aussieht, so daß gutgemeinter Rat angenommen, aber noch lange nicht befolgt wird. Und so wird die nächste Projektgruppe vielleicht eine Chronik erstellen, die sich von dieser inhaltlich nicht sehr unterscheidet. Womit sie wie wir das Ziel einer Projektgruppe erreicht hätte, denn das ist, „Fehler zu machen“, vor allem, Fehler einmal *selber* machen zu dürfen.

9.1.1 Arbeitsorganisation

Die Überschrift dieses Abschnitts verleitet zu der falschen Annahme, daß von vornherein eine klare Vorstellung vorhanden gewesen wäre, wie die Arbeit der Projektgruppe organisiert werden sollte. Diese Annahme ist falsch. Zu Beginn bestand die

gesamte „Organisation“ für die Gruppentreffen aus den Terminen dreimal wöchentlich, dem Verfassen eines Protokolls zu jedem dieser Treffen und einem Gerüst für eine Tagesordnung. All dies war von Herrn Prof. Claus so vorgeschlagen worden.

Weiteres entwickelte sich erst im Laufe der Zeit, und dies oft nicht, bevor es mehrmals unorganisiert, um nicht zu sagen chaotisch, abgelaufen war. Als Beispiele hierfür werden im folgenden die Entscheidungsfindung innerhalb der Gruppe und die Arbeit von Untergruppen geschildert.

9.1.1.1 Entscheidungsfindung

Da es außer dem vagen Ziel, irgendein System, das mit Evolutionären Algorithmen arbeitet, zu erstellen, kaum Vorgaben gab, mußten die meisten Entscheidungen von der Projektgruppe selbst getroffen werden. Natürlich möchte niemand, daß über seinen Kopf hinweg entschieden wird, so daß anfangs Entscheidungen basisdemokratisch von einer Mehrheit der bei den regelmäßigen Treffen Anwesenden getroffen wurden, und das möglichst im allgemeinen Konsens.

Obwohl deshalb sehr viel Zeit mit Diskussionen über nebensächliche Kleinigkeiten verging, wurde diese Ein-Ebenen-Hierarchie bis zum Ende der Implementierungsphase beibehalten. Dabei ist „Entscheidung“ nicht gleichbedeutend mit „Entscheidung für immer“. Abstimmungen über ein und dasselbe Thema konnten beliebig oft wiederholt werden, und Mehrheiten wechselten oft innerhalb einer Sitzung, wenn über einen gerade gefaßten Entschluß nur lange genug diskutiert worden war. Um während dieser Diskussionen wenigstens jeden gleichberechtigt zu Wort kommen zu lassen, wurde das Amt des von Mal zu Mal wechselnden Sitzungsleiters mit der Befugnis versehen, das Wort zu erteilen und zu entziehen.

Erst zur Erstellung dieses Endberichts wurde das Verfahren — im Konsens! — grundlegend geändert. Eine Leitung zur Erstellung des Endberichts wurde eingerichtet. Zwar gab es auch zuvor schon eine „Projektleitung“, der aber von den nicht daran Beteiligten keinerlei Entscheidungsbefugnis zugebilligt wurde. Die Leitung zur Erstellung des Endberichts, bestehend aus einem Teilnehmer und der Betreuerin Nicole Weicker, erstellte einen straffen organisatorischen Rahmen inklusive eines genauen Zeitplans und setzte dessen Einhaltung bei den Autoren durch.

9.1.1.2 Untergruppen

Beinahe von Anfang an wurden zur ausführlicheren Diskussion einzelner Themen und zur Bearbeitung von Teilaspekten des Projekts Untergruppen aus zwei bis fünf Teilnehmern eingerichtet. Solche Untergruppen sollten vorbereitend zu einer effizienteren Zeitausnutzung bei den Treffen der Gesamtgruppe beitragen und das Vorankommen

des Projekts beschleunigen. Anfangs wurden solche Untergruppen durch bloße Angabe einer Bezeichnung ohne Angabe einer genauen Aufgabenstellung ins Leben gerufen. Dies hatte zur Folge, daß Ergebnisse der Untergruppe, soweit es überhaupt welche gab, oft nur in den Köpfen der Untergruppenteilnehmer vorhanden waren und den anderen nicht vermittelt werden konnten. Auch lagen die Ergebnisse häufig zu spät vor oder hielten nicht, was man sich bei der Einrichtung der Untergruppe versprochen hatte — soweit das bei allen dasselbe war. Als diese Arbeitsweise nach einiger Zeit als wenig effektiv erkannt wurde, wurde ein „Formular für die Einrichtung einer Untergruppe“ entworfen, in das unter anderem die Namen der Untergruppenteilnehmer, eine Beschreibung der Aufgabenstellung, ein Termin für die Vorstellung der Ergebnisse und die Art und Weise, wie diese Vorstellung zu erfolgen hatte, eingetragen wurden. So führte oft ein Mehr an Bürokratie zu einem Mehr an Effektivität.

9.1.2 Ablauf

Die beiden Semester, über die sich die Arbeit der Projektgruppe erstreckte, sind nach den zu erwerbenden Scheinen in

- eine Seminarphase,
- eine Fachpraktikumsphase und
- eine Studienarbeitsphase

unterteilt. Die einzelnen Phasen werden in den folgenden Abschnitten näher beschrieben. Die Fachpraktikums- und Studienarbeitsphase sind — stärker als die Seminarphase — durch die Ergebnisse charakterisiert. Bei der Fachpraktikumsphase ist das ein Rechnerprogramm, bei der Studienarbeitsphase der vorliegende Abschlußbericht. Die Arbeit einer Projektgruppe läßt sich aber inhaltlich nicht eindeutig in „Arbeit im Rahmen des Fachpraktikums“ und „Arbeit im Rahmen der Studienarbeit“ trennen. Deshalb werden diese beiden Phasen in einem gemeinsamen Abschnitt beschrieben.

9.1.2.1 Seminarphase

Die Seminarphase begann bereits in der vorlesungsfreien Zeit vor Beginn des Sommersemesters, in der die Seminarvorträge vorbereitet wurden, und endete mit dem letzten Vortrag Ende Mai. Dabei wurden folgende Vorträge gehalten (der in Klammern jeweils angegebene Name bezeichnet den Vortragenden):

- Naturanaloge Optimierungsverfahren (Nicole Weicker)
- Die Evolutionsstrategie (Bernd Kawetzki)

- Genetische Algorithmen, Schematheorem (Wilfried Kuhn)
- Beschreibung konkreter Implementierungen (Kurt Jaeger)
- Einfluß und Wahl der Parameter, der Kodierung und der Operatoren (Karsten Jung)
- Neuronale Netze, Hopfield-Netze und Boltzmann-Maschinen (Markus Schaal)
- Anwendung genetischer Algorithmen auf neuronale Netze (Frank Amos)
- Anwendung genetischer Algorithmen in der Produktionsplanung (Ralf Reißing)
- TSP — Lösungsansätze (Oliver Pertler)

Eine Zusammenfassung jedes dieser Vorträge kann dem Zwischenbericht der Projektgruppe [AJJ+94] entnommen werden.

Durch die Seminarvorträge konnte jeder Teilnehmer der Projektgruppe innerhalb weniger Wochen sowohl ein Verständnis für die Funktionsweise evolutionärer Verfahren als auch einen Einblick in ihre Umsetzung in Rechnerprogramme und Möglichkeiten ihrer Anwendung gewinnen. Dadurch entwickelten sich bereits während der Seminarphase beim einzelnen Vorstellungen über das Projekt, das die Gruppe während der darauffolgenden Fachpraktikumsphase in Angriff nehmen sollte.

9.1.2.2 Fachpraktikums- und Studienarbeitsphase

Während die Vorträge der Seminarphase dazu dienten, Kenntnisse über das zu bearbeitende Fachgebiet der Evolutionären Algorithmen zu sammeln, stellten die zu Beginn der Fachpraktikums-/Studienarbeitsphase eingeladenen Gäste Beispiele für bereits erfolgte Implementierungen Evolutionärer Algorithmen vor und berichteten über ihre Erfahrungen. Eine Ausnahme hiervon ist der Vortrag von Kurt Schneider, der die Durchführung eines Projekts unter dem Gesichtspunkt des Software Engineering erläuterte. Die Gastvorträge waren im einzelnen:

- Modelle und Simulation von Software-Projekten (Kurt Schneider)
- Quantitative hochauflösende Elektronenmikroskopie (Rainer Bierwolf)
- Geometrieoptimierung von Clustern mit Genetischen Algorithmen (Dr. Bernd Hartke)
- Genetische Algorithmen zur Lösung von Ablaufplanungsproblemen (Ralph Bruns)
- Evolutionäre Algorithmen in Theorie und Praxis (Günther Rudolph)

Zusammenfassungen zu den Gastvorträgen können ebenfalls dem Zwischenbericht [AJJ+94] entnommen werden. Ergänzend hierzu wurden während der Gruppensitzungen von einigen Projektgruppenteilnehmern sowie von Herrn Prof. Claus Kurzvorträge zu speziellen Themengebieten wie „Konsensfindung in Gruppen“ oder „Software Engineering“ gehalten.

Der weitere Ablauf der kombinierten Fachpraktikums- und Studienarbeitsphase war von Herrn Prof. Claus wie folgt vorgesehen:

- Planungsphase (Mitte Mai 1994 bis Anfang Juni 1994)
- Entwurfsphase (Anfang Juni 1994 bis Anfang Juli 1994)
- Pufferphase (Anfang Juli 1994 bis Mitte Juli 1994)
- Implementierungsphase (ab Mitte Juli 1994)

Für den Endbericht sollten während der einzelnen Phasen stets die entsprechenden Teile der Dokumentation angefertigt werden.

Ein diese Planung verfeinernder und von der „Projektleitung“ erstellter Meilensteinplan wurde, als die ersten Meilensteine Anfang Juli bereits längst überschritten waren, von der Gruppe kurzerhand verworfen. Somit ergibt sich folgende tatsächliche Zeitaufteilung:

- Planungs-/Entwurfsphase (Anfang Juni 1994 bis Mitte Oktober 1994)
- Implementierungsphase (Mitte Oktober 1994 bis Ende 1994)

Ein Vergleich des vorgesehenen mit dem tatsächlichen Zeitaufwand zeigt, daß der ursprüngliche Zeitplan von der Projektgruppe insbesondere hinsichtlich der Planungs- und Entwurfsphase nicht ganz eingehalten wurde. Außerdem ist es nicht möglich, zwischen Planungs- und Entwurfsphase eine scharfe Trennlinie zu ziehen, weil sich die Projektgruppe zeitweise in einem Stadium der Entwurfsphase befand, dann feststellte, daß einige der in der Planungsphase getroffenen Entscheidungen so nicht umsetzbar waren, und deshalb wieder in die Planungsphase eintrat.

Beispielsweise war bereits Ende Juni ein detaillierter Systementwurf mit genauen Beschreibungen der Systemteile Problem, Verfahren, Lauf und Auswertung erarbeitet worden. Die Beschreibungen dieser Teile waren aber von jeweils einer Untergruppe entwickelt worden. Beim Versuch, die einzelnen Systemteile zu einer Gesamtheit zusammenzufügen, traten deutliche Unterschiede in den Vorstellungen der Teilnehmer hinsichtlich der vom System zu erbringenden Leistung hervor. Dies führte dazu, daß alle bis dahin erstellten Dokumente beiseite gelegt oder zumindest in Frage gestellt

wurden und in einem „Neuanfang“ jeder Teilnehmer seine Vorstellung in einem Szenario formulierte. Sowohl der erste Entwurf der Systemteile als auch die Szenarien sind im Zwischenbericht [AJJ+94] enthalten.

Um anschließend die einzelnen Vorstellungen miteinander in Einklang zu bringen, wurde eine Liste der mit den Szenarien identifizierten Systemteile erstellt. Beispiele für Punkte auf der so erstellten Liste sind Debug-Möglichkeiten bei der Ausführung Evolutionärer Algorithmen, Bereitstellung einer Operatorbibliothek mit der Möglichkeit, benutzerdefinierte Operatoren hinzuzufügen und eine grafische Benutzungsoberfläche zur Bedienung des Programms. Eine Realisierung aller auf der Liste aufgeführten Systemteile würde somit eine Art *Maximalsystem* bilden, das alle von den einzelnen Teilnehmern geforderten, zum Teil widersprüchlichen Möglichkeiten zur Erforschung Evolutionärer Algorithmen bietet. Anschließend wurde jeder Punkt der Liste gemeinschaftlich in eine der Kategorien *muß*, *soll möglich sein* und *kann* eingeteilt, was bedeutet:

muß: Ein in diese Kategorie eingeordneter Systemteil wird innerhalb des Gesamtsystems unbedingt implementiert.

soll möglich sein: Unter diese Kategorie fallende Systemteile werden nur dann implementiert, wenn die Implementierung mit geringem Aufwand verbunden ist. Ansonsten sind unbedingt Schnittstellen vorzusehen, die eine nachträgliche Implementierung ermöglichen.

kann: Die Systemteile dieser Kategorie sind nicht Bestandteil des Gesamtsystems. Schnittstellen werden nur dann vorgesehen, wenn der dafür erforderliche Aufwand gering ist.

Mit dieser Einordnung der Systemteile lag nun eine Art Pflichtenheft für den Entwurf vor. Allerdings ist festzustellen, daß die Anforderungen an das zu implementierende Gesamtsystem mit fortschreitender Zeit immer geringer wurden, so daß sich der Entwurf immer mehr auf den „muß“-Teil konzentrierte.

Die Wahl von C++ als Programmiersprache überdauerte den Neuanfang. Die einzige Alternative, die ernsthaft in Betracht gezogen wurde, war Smalltalk, wurde aber nach Einholung von Erfahrungsberichten bei Mitarbeitern des Informatik-Instituts, die von der Verwendung von Smalltalk für eine Gruppenarbeit abrieten, fallengelassen.

Da der Neuanfang am Ende des Sommersemesters stattfand, verzögerte sich die kombinierte Planungs-/Entwurfsphase in die vorlesungsfreie Zeit hinein. Wegen Prüfungen und wohlverdienten Urlaubs erschienen in dieser Zeit zu praktisch keiner Sitzung mehr alle Teilnehmer. Das muß aber nicht als Nachteil gesehen werden, weil sich dadurch natürlich die Anzahl unterschiedlicher Meinungen verringerte. Andererseits führte der Fortgang oder Stillstand des Projekts bei der Rückkehr manchmal zu Überraschungen.

In dieser Zeit wurden außer der schließlich in die Implementierung übernommenen Lösung noch zwei weitere Ansätze verfolgt. Der eine wird hier als „Compiler-Modell“, der andere als „Tcl-Entwurf“ bezeichnet. Da diese beiden Ansätze gleichwertig zu den dann gewählten Entwürfen sind, werden sie an dieser Stelle kurz vorgestellt.

Vergleich von Compiler-Modell und Tcl-Entwurf Das Compiler-Modell entstand als Alternative zu dem bis dahin favorisierten Entwurf, der die Erweiterung eines vorhandenen Interpreters unter UNIX, des sogenannten Tcl-Interpreters, um Datenstrukturen und Befehle zur Ausführung Evolutionärer Algorithmen vorsah. Da dieser Interpreter zwar eine Schnittstelle zu C besitzt, aber keine objektorientierte Schnittstelle zu C++ , entwickelte eine Untergruppe einen Entwurf, der die Bereitstellung einiger grundlegender Datenstrukturen und Operatoren zur Ausführung Evolutionärer Algorithmen in C++ vorsah.

Um bei diesem Entwurf ein ausführbares Programm zu erhalten, ist dabei der zu verwendende Algorithmus sowie die Problemparameter vom Benutzer zu spezifizieren. Diese werden dann in ein in C++ formuliertes „main“-Modul übersetzt. Anschließend wird dieses „main“-Modul vom normalen C++ -Compiler übersetzt, mit den bereitgestellten Basisdatenstrukturen und der Operatorbibliothek zusammengebunden und zur Ausführung gebracht.

Für dieses Modell und damit gegen das Tcl-Interpreter-Modell spricht, daß Evolutionäre Algorithmen in einer speziell dafür entwickelten Sprache formuliert werden können und die Probleme, die eine Schnittstelle zum Interpreter mit sich bringt, nicht auftreten. Zu den Argumenten, die gegen das Compiler-Modell vorgebracht wurden, zählte die gegenüber dem Interpreter-Modell längere Zeitspanne bis zum Beginn der Ausführung des Evolutionären Algorithmus' nach seiner Eingabe und der Einwand, daß die Projektgruppe am Versuch, einen Übersetzer zu schreiben, scheitern würde.

Daraufhin wurde der Ansatz mit Tcl-Interpreter unter dem neuen Projektnamen „EAGLE“ bis Ende September weiterverfolgt, als einer der Hauptbefürworter der Verwendung des Tcl-Interpreters mit Hinweis auf die durchaus uneffiziente Arbeitsweise der Projektgruppe seinen Austritt erklärte. Kurz darauf wurde beschlossen, den Tcl-Interpreter nicht zu verwenden, sondern den Interpreter selbst zu schreiben. Damit spart man die Programmierung der Schnittstelle, weil der selbst geschriebene Interpreter auf die grundlegenden Datenstrukturen und Operatoren direkt zugreift.

Umsetzung des Entwurfs Da es uns aus Zeitmangel nicht mehr möglich war, eine vollständige Design-Phase zu durchschreiten, erfolgte ein ad hoc Entwurf in Form von Header-Files. In der relativ kurzen Zeit, die uns dazu zur Verfügung stand, stützten wir uns auf Erfahrungen eines relativ vollständigen Entwurfs, der in der Zeit entstand, als die PGA beschloß, einen Neuanfang zu starten. Aus diesem Grund wurde dieser Entwurf nicht zu Ende geführt und vor allem nicht in akzeptabler Form schriftlich

festgehalten. Dennoch bot er uns genügend Anhaltspunkte, um eine Schnittstellenspezifikation zu erstellen.

Anhand der Header-Files wurde in einer knapp bemessenen Kodierungsphase versucht, Code zu erzeugen. Um dies in der zur Verfügung stehenden Zeit zu bewerkstelligen, wurden die Anforderungen für eine erste Version des Codes nochmal reduziert. So wurde auf die Realisierung der Oberfläche für das erste verzichtet. Allein die Kernanforderungen (Problemeingabe, Kodierungseingabe, Verfahrenseingabe und Verarbeitung des in LEA spezifizierten EA-Codes) sollten in dieser ersten Version verfügbar sein. Da zu diesem Zeitpunkt aber eine Einbindung des LEA-Interpreters unrealistisch war, wurde nur mit in C++ hart codierten Operatoren gearbeitet. Zum Abschluß der Codierungsphase war Code soweit vorhanden um folgenden Test zu erfüllen. Es wurde als Problem ein nur aus einem Real bestehende Problemstruktur im Code fest implementiert. Im Kodierungsteil eine als REAL_CODING kodierte Kodierungsstruktur fest implementiert und als Ablauf ein fest implementierter Mutationsoperator. Mit diesen Anforderungen wurde die Integration angegangen.

Integration Da unser Zeitbudget zu diesem Zeitpunkt eigentlich schon erschöpft war, konnte in der Integrationsphase nicht mehr die Kommunikationsdichte aufrechterhalten werden, die nötig gewesen wäre, um diese Phase erfolgreich abzuschließen. Von der Entdeckung eines Fehlers (unabhängig von der Art des Fehlers) bis zu seiner Weiterleitung an die richtige Adresse und seiner Behebung verging zu viel Zeit, um diese Phase noch abschließen zu können. Dennoch gab uns diese Integrationsphase und auch die vorausgehende Codierungsphase, die von der Qualität eigentlich als Prototyping zu bezeichnen ist, wertvolle Informationen und Hinweise im Hinblick auf Pflichtenheft, Spezifikation der Anforderungen und den Grobentwurf.

9.1.2.3 Ausblick

Nachdem die Projektgruppe mit der Vorlage dieses Endberichts offiziell zu einem Abschluß gekommen ist, stellt sich die Frage danach, wie es nun weitergeht.

Obwohl der Projektgruppe zunächst eine Umsetzung ihrer Vorstellungen nicht möglich war, halten wir sie grundsätzlich für realisierbar. Für eine Realisierung wird es sicher nötig sein, die ersten Phasen der Software-Erstellung in einer Art Schnelldurchlauf erneut zu durchschreiten, beginnend mit einem Pflichtenheft, einer ausformulierten Spezifikation bis zu einem klaren und verständlichen Entwurf. Diese Punkte sind im Rahmen der Projektgruppe schon inhaltlich bearbeitet worden. Die zu diesen Phasen des Software-Entwicklungsprozesses gehörenden Dokumente wurden für diesen Endbericht erstellt und sollten für ein kleineres EAGLE-System eventuell noch einmal überarbeitet werden.

9.2 Fazit

Die Projektgruppe soll in erster Linie Lehrveranstaltung sein. Entscheidend für ihren Erfolg ist nicht die Güte des entwickelten Produktes, sondern das von den Teilnehmern erworbene Wissen. Die bereits in Kapitel 1 aufgelisteten Ausbildungsziele — Arbeiten im Team, selbständige Erarbeitung von Lösungsvorschlägen und deren Vorstellung und Verteidigung in einer Gruppe, Übernahme von Verantwortung, Persönlichkeitsbildung, Durchlaufen eines vollständigen Software-Lifecycles usw. — sind z. T. nicht abprüfbar. Dennoch ist ihre Bedeutung unbestreitbar wichtig.

„Sichtbare“, d.h. nachprüfbare Ergebnisse der PGA sind:

- Umfangreiches Fachwissen im Bereich Evolutionäre Algorithmen
- Vorbereitung und Vortrag eigener und fremder Ideen
- Grundideen des Projektmanagements
- Programmierkenntnisse in C++ und Arbeiten unter einer Entwicklungsumgebung
- Kenntnisse im Umgang mit Tools (Emacs, E-mail, LaTeX, FAQs, ...)
- Erfahrung mit Dokumentationsarbeit

Zudem viel weiteres Fachwissen aus den zahlreichen Vorträgen und der recherierten Literatur. Dabei zeigte sich ein weiterer Vorteil der Arbeit in der Gruppe: Studenten können auch von anderen Studenten etwas lernen. Mehr als in anderen Lehrveranstaltungen war hier Gelegenheit, jemanden etwas zu fragen oder sich etwas zeigen zu lassen. Da jeder von uns sich in bestimmte Aufgabengebiete einarbeitete, konnte er sein damit oder schon vorher erworbenes Fachwissen innerhalb der Gruppe weitergeben.

Außer diesen nachprüfbaren Fakten gibt es noch Ergebnisse, die sich erst dann zeigen werden, wenn ein Projektgruppenmitglied wieder in einem Softwareprojekt oder einer Gruppe arbeitet. Selbstverständlich ist kein Projektgruppenteilnehmer nun ein projekterfahrener Mitarbeiter, doch sind oft die Einsichten, wie etwas nicht geht, die nachhaltigsten. Die Abwicklung von größeren Softwareprojekten erfordert praktische Erfahrung, die in der Lehre andernorts kaum vermittelt werden kann. Die Arbeit in der Projektgruppe zeigt, welche Punkte bei derartigen Vorhaben besonders *wichtig* sind:

- Umfassende Planung
- Beachtung der Prinzipien des Software-Engineering

- Disziplin bei der Einhaltung von Terminen (Meilensteinen),
- Erfahrung in Projektabwicklung und Management
- Kommunikationsfluß innerhalb der Gruppe
- Saubere Ausarbeitungen verbindlicher Dokumente für jede Phase der Entwicklung
- angemessene Dokumentation der gesamten Projektarbeit
- Zuhören und Verständnis für andere Vorschläge

Probleme und Fehler

Hauptproblem war wohl der Frust, daß das System im geplanten Umfang nicht mehr von uns fertiggestellt werden konnte. Dazu gibt es eine Menge von Gründen:

- Keiner von uns hatte Erfahrung bei der Planung eines Projekts dieser Größe, von der Betreuung kam hier zu wenig Unterstützung.
- Da wir zu acht die Anforderungsspezifikation erstellen wollten, kam es zu Kommunikationsproblemen und das anvisierte System wurde zu umfangreich.
- Mehrmals wurden schon festgelegte Meilensteine verschoben, ohne dies in eine weitergehende Planung einzubinden.
- Entscheidende Dokumente (Anforderungsspezifikation, Grobdesign, ...) wurden zu spät oder zu unsauber ausgearbeitet oder nicht als verbindliche Grundlage einbezogen. So wurde in Diskussionen oft aneinander vorbei geredet.

Eine „stärkere“ Betreuung oder engere Vorgaben hätten die Fertigstellung des Systems im geplanten Umfang vielleicht sichern können. Jedoch stellt sich die Frage, ob uns so auch die Fallgruben und Fehlerquellen in solchem Maße bewußt geworden wären. Nach unserer allgemeinen Ansicht hätte etwas steuerndes Eingreifen uns hier dennoch erfreut, um Frust zu vermeiden. Die Fertigstellung des Systems war für alle ein wichtiger Motivationsgrund gewesen, deutlich mehr als die geplanten 16 Semesterwochenstunden zu investieren.

Vorschläge zur Durchführung weiterer Projektgruppen

Zur Organisation der Projektgruppe Die von Anfang an bestehende und sich über die gesamte Projektzeit hinziehende Ungewißheit hinsichtlich der Anerkennung

des Abschlußberichts als Prüfungsleistung vom Niveau einer Studienarbeit war sicherlich aus studentischer Sicht alles andere als erfreulich. Für nachfolgende Projektgruppen sollte diese Frage jetzt zwar geklärt sein, dennoch wäre eine Verankerung der Projektgruppe in der Prüfungsordnung hier sicher hilfreich. Wir empfehlen, dabei vom Prinzip der individuell bewertbaren Leistung abzurücken, da dieses Prinzip in einer Projektgruppe nicht sinnvoll umgesetzt werden kann. (Man denke dabei auch an den Prüfer, der diese individuelle Bewertung vornehmen soll.)

Eine Vorbereitung durch eine Vorlesung ist auch in Erwägung zu ziehen. Dann wäre das Themengebiet vor Beginn der Projektgruppe bekannt und die Zeit zur Einarbeitung könnte anderweitig verwendet werden.

Zur Betreuung Hilfestellung seitens eines Betreuers mit Erfahrung im Projektmanagement ist wohl notwendig. Diese sollte als Beratung auf Anfrage der Projektgruppe gestellt werden, nicht durch strenge Vorgaben zu Beginn. Damit sie nicht öfter als notwendig genutzt wird, könnte sie zeitlich oder durch Kosten limitiert werden.

Ratschlag für weitere Projektgruppen Schaut Euch unsere Fehler an. Leicht überschätzt man die Schwierigkeiten, die ein Projekt in sich birgt. So hätten auch wir uns wohl Berichte von Projektgruppen anderer Universitäten durchlesen sollen. Alles

in allem war die Teilnahme an der Projektgruppe für uns interessant und lehrreich, und die inoffiziellen Aktivitäten wie Ausflüge und Essen waren eine willkommene Auflockerung unseres tristen Studentendaseins. Es wäre aus unserer Sicht durchaus wünschenswert, diese Lehrveranstaltungsform an der Fakultät fest einzuführen.

Anhang A

Glossar und Abkürzungen

A.1 Glossar

Dieser Abschnitt ist als Begriffsklärung der wichtigsten in dieser Dokumentation verwendeten Begriffe gedacht. Wir bitten um Verständnis, daß sich dabei nicht in allen Fällen eine Konsistenz mit anderen Autoren herstellen ließ. Alle Begriffe, die einen eigenen Eintrag besitzen, sind in den Erklärungen *kursiv* geschrieben.

Aktionselement: *Aktionselemente* sind funktionale Einheiten, die *Datenelemente* erzeugen, manipulieren und löschen können. Im Zusammenspiel von Funktionen und Operanden entsprechen Sie den Funktionen.

Algorithmus: Ein *Algorithmus* ist eine Rechenvorschrift.

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

Bitstring: Ein *Bitstring* ist eine geordnete Liste von Boolean-Atomen.

coarse-grained-model: siehe *parallele Genetische Algorithmen.*

Constraint: Nebenbedingung, die den *Suchraum* einschränkt.

Crossover: Unter *Crossover* wird die Kreuzung zweier (oder mehrerer) *Individuen* einer *Population* in Analogie zur Kreuzung in der Natur verstanden. Teile der ausgewählten *Individuen* werden zu einem (oder mehreren) neuen *Individuen* zusammengesetzt. Setzt sich das neue *Individuum* aus je einer Hälfte der beiden Elternteile zusammen, spricht man von One–Point–Crossover. Bei Multi–Point–Crossover werden die (beiden) Eltern an mehreren analogen Stellen auseinandergebrochen, bei Uniform–Crossover werden entsprechende Werte in den Tupeln der Eltern zufällig vertauscht.

Datenelement: *Datenelemente* sind Datenstrukturen zum Aufbewahren von Daten. Im Zusammenspiel von Funktionen und Operanden entsprechen Sie den Operanden.

dekodiertes Individuum: Ein *dekodiertes Individuum* ist ein *Individuum* aus Sicht des *Problems*. Durch die *Dekodierungsfunktion* wird den einzelnen *Atomen* der *Problemstruktur* ein Wert zugewiesen. Es handelt sich dabei um ein Element des *Suchraums* bzw. um ein Wertebelegung der *Problemstruktur*.

Dekodierungsfunktion: Die *Dekodierungsfunktion* ist die Umkehrfunktion der *Kodierungsfunktion* und dient der Umwandlung eines *Individuums* (als Instanz der *Kodierungsstruktur*) in ein Element des *Suchraums* (als Instanz der *Problemstruktur*). Dies ist erforderlich, da die *Fitneßfunktion* nur auf Elemente des *Suchraums* angewendet werden kann.

Evolutionärer Algorithmus: Im Zusammenhang mit EAGLE der Oberbegriff für folgende stochastische Verfahren:

- *Evolutionsstrategien*
- *Genetische Algorithmen*
- *Threshold–Algorithmen*
- *Simulated Annealing*
- weitere stochastische Verfahren unter Verwendung von *Populationen*

Obwohl *Simulated Annealing* eigentlich auf *Individuen* angewendet wird, kann man sich vorstellen, gleich eine ganze *Population* von *Individuen* zu betrachten, um die Chancen einer erfolgreichen Suche zu erhöhen.

Evolutionsstrategie: *Evolutionstrategien* (ES) sind von I. Rechenberg [Rec73] entwickelte, naturanaloge *Verfahren*, die erst zur Optimierung konkreter technischer *Probleme* (z.B. Minimierung des Reibungswiderstands bei Düsen) eingesetzt wurden. Wesentliche Charakteristika: *Population* von möglichen Lösungen (Individuen) werden nach dem Vorbild der biologischen Evolution mit Methoden wie *Rekombination* und *Mutation* verändert und nach dem Prinzip „survival of the fittest“ ausgewählt. Bei den ursprünglichen technischen Problemen wurde die Fitneß nicht berechnet, sondern durch eine konkrete Simulation bestimmt. Dabei werden spezielle Parameter, die den weiteren Verlauf der Suche mitbestimmen, bei diesem *Verfahren* im *kodierten Individuum* gehalten und dadurch mitoptimiert, (siehe auch Bäck/Schwefel [BS93]).

Experiment: Ein *Experiment* beinhaltet zusätzlich zu den Bestandteilen einer Experimentdefinition eine Belegung der *Parameter* und eine log_datei.

Experimentdefinition: Eine *Experimentdefinition* setzt sich aus einem *Problem*, einer *Kodierung* und einem *Verfahren* zusammen. Dabei müssen die drei Teile zueinander passen in dem Sinne, daß die Kodierungsstruktur mit der Problemstruktur verträglich ist (siehe Kodierungsstruktur) und daß die Operatoren des Verfahrens auf der Kodierungsstruktur arbeiten kann.

Filter: Ein *Filter* selektiert während des Laufs anfallende Daten und schreibt sie in die mit ihm verbundene Datei. Im *LEA*–Code werden die *Filter* genauso wie *Label* gesetzt. Sie können dann in der *Laufinitialisierung* an ein File gebunden und aktiviert bzw. deaktiviert werden.

fine-grained-model: siehe *parallele Genetische Algorithmen.*

Fitneßfunktion: Die *Fitneßfunktion* erhält ein Element des *Suchraums* (*dekodiertes Individuum*) und gibt einen reellen Fitneßwert zurück.

Generation: Eine *Generation* ist eine *Population* aus einer Folge von *Populationen.* Diese Folge wird durch die Iterationen eines Verfahrens bestimmt. Die *Population* $P(t), t \in \mathbb{N}$, die durch die t. Iteration des Verfahrens erzeugt wird, ist die t. Generation.

Genetische Algorithmen: *Genetische Algorithmen* (GA) gehören zur Klasse der stochastischen Optimierverfahren, die bewußt Prinzipien der biologischen Evolution nachahmen. Bei *Genetischen Algorithmen* ist die *Kodierungsstruktur* immer ein Binärstring . Instanzen dieses Binärstrings bilden als *Individuen* eine *Population.* Diese *Population* wird durch genetische *Operatoren* wie z.B. *Mutation* und *Crossover* manipuliert. Dabei hofft man, durch Kreuzung die guten Eigenschaften in der *Population* zu vervielfältigen. Die *Individuen* einer *Population* werden nach einer vom Anwender vorgegebenen *Fitneßfunktion* bewertet und sodann mit Hilfe dieser Bewertung selektiert. Dieser Vorgang wird iteriert, bis ein vorgegebenes Abbruchkriterium erfüllt ist (z.B. Anzahl der iterierten *Generationen*, Güte der besten Lösung, ...). GA wurden von Holland schon in den 70ern entwickelt und durch das Schematheorem theoretisch von ihm begründet [Hol75], (siehe auch Goldberg [Gol89]).

Grunddatentyp: Ein *Grunddatentyp* ist einer der folgenden Datentypen:

- Permutation mit Längenangabe
- Integer mit Wertebereich
- Real mit Wertebereich
- Boolean

Hill–Climbing: *Hill–Climbing* ist der Oberbegriff zu einer Reihe von lokalen Optimierungsverfahren, die von einem beliebigen Startpunkt innerhalb des *Suchraums* aus zu einem lokalen Optimum gelangt, indem sie der Richtung der (größten) Verbesserung der *Fitneßfunktion* folgt, (siehe auch Schwefel [Sch81]).

Hauptoperator: Unter einem *Hauptoperator* wird in Zusammenhang mit *LEA* ein *Operator* verstanden, der als *Verfahren* in einer Experimentdefinition eingesetzt werden kann. *Hauptoperatoren* zeichnen sich in *LEA* dadurch aus, daß als Aufrufparameter eine *Population* übergeben wird und als Rückgabewert ein *Individuum* zurückgegeben wird. Dieses Individuum ist in der Regel das beste *Individuum*, das das *Verfahren* ermittelt hat.

Individuum: Ein *Individuum* ist die konkrete Ausprägung der *Kodierungsstruktur*. Alle *Atome* der *Kodierungsstruktur* sind mit Werten belegt. Es handelt sich also aus objektorientierter Sicht um eine Instanz der *Kodierungsstruktur* bzw. aus der Sicht herkömmlicher Programmiersprachen um ein Variable vom Typ *Kodierungsstruktur*.

Inselmodell: siehe *parallele Genetische Algorithmen*.

Kodierung: Die *Kodierung* besteht aus *Kodierungsstruktur* und *Kodierungsfunktion*.

Kodierungsfunktion: Die *Kodierungsfunktion* beschreibt die injektive Abbildung von der *Problemstruktur* auf die *Kodierungsstruktur*. Die *Kodierungsfunktion*

- bildet jedes *Atom* der *Problemstruktur* auf ein oder mehrere *Atome* der Kodierungstruktur ab. Dabei gibt es folgende Möglichkeiten:
  - Permutation wird auf Permutation abgebildet (Identität).
  - Boolean wird auf Boolean abgebildet (Identität).
  - Integer wird auf Integer (Identität) oder einen Bitstring (gray–kodiert oder standard) abgebildet (dabei wird die Länge des Bitstrings durch den Bereich von Integer und eine anzugebende Schrittweite festgelegt).
  - Real wird auf Real (Identität) oder einen Bitstring (gray–kodiert oder standard) abgebildet (dabei wird die Länge des Bitstrings durch den Bereich und eine anzugebende Genauigkeit von Real festgelegt).
- legt die Anordnung der *Atome* der *Kodierungsstruktur* in einer Liste fest. Dabei wird ein *Bitstring* wie ein einzelnes Atom behandelt.

Dadurch entsteht eine *Kodierungsstruktur*, die automatisch mit der *Problemstruktur* verträglich ist. Die Umkehrabbildung der *Kodierungsfunktion* wird *Dekodierungsfunktion* genannt und dient der Rückabbildung der Individuen als Instanzen der *Kodierung* auf Elemente des *Suchraums* als Instanzen der *Problemstruktur*.

Kodierungsstruktur: Die *Kodierungstruktur* besteht aus einer Liste von Atomen. Eine *Kodierungstruktur* ist genau dann mit einem *Problem* verträglich, wenn eine injektive, strukturerhaltende Abbildung der *Problemstruktur* auf die *Kodierungsstruktur* existiert. Diese Abbildung heißt *Kodierungsfunktion*.

Label: Ein *Label* ist eine Marke im *Verfahren*. Vom Benutzer interaktiv eingegebene Steuerungsbefehle (z.B „Halt“) werden bei Erreichen des nächsten passenden *Labels* ausgeführt. *Label* werden im LEA–Code gesetzt. Sie können in der *Laufinitialisierung* als Halte- oder Abbruchlabel definiert werden. Außerdem können sie aktiviert oder deaktiviert werden. Ein vom Benutzer eingegebenes „Halt“ bewirkt ein Anhalten beim Erreichen des nächsten Haltelabels. Hingegen wird beim Erreichen eines aktiven Abbruchlabels die Abarbeitung des *Laufs* automatisch beendet.

Lauf: Ein *Lauf* ist die Abarbeitung des LEA-Codes und der Verarbeitung der interaktiven Einflußnahme des Benutzers, sowie alle auf dem Bildschirm ausgegebenen und alle in der log_datei protokollierten Informationen.

Laufinitialisierung: Die *Laufinitialisierung* ist die vor und auch während des *Laufs* (wobei dann eine Unterbrechung des *Lauf* nötig ist) jederzeit durchführbare Modifikation der Parameterbelegung des *Verfahrens*, der Filter- und der Labelbelegung.

LEA (Language for Evolutionary Algorithms): Sprache zur Eingabe von *Evolutionären Algorithmen* in EAGLE. Zur genauen Definition von *LEA* siehe auch Abschnitt 6.

log_datei: Eine *log_datei* enthält alle Informationen, die zu einer vollständigen Rekonstruktion eines ununterbrochenen Laufs notwendig sind. Das schließt z.B. die Startpopulation und die Laufinitialisierung ein. Ein *Experiment* kann entweder ohne die Informationen aus der *log_datei* neu gestartet werden oder auf diese Informationen aufsetzen.

Meta–Verfahren Unter einem *Meta–Verfahren* versteht man ein *Verfahren*, das selbst wieder ein oder mehrere *Verfahren* zum Problemgegenstand hat. Oft wird ein solches Meta–Verfahren dazu verwendet, die Parameterbelegungen von EAs zu optimieren. Dabei besteht die *Problemstruktur* aus den *Parametern* des EA. Die *Fitneßfunktion* bewertet dann die Effizienz des EA mit diesen *Parametern*, indem sie ihn z.B. mehrmals ablaufen läßt. Das *Verfahren*, nach dem die *Parameter* optimiert werden, ist damit nicht festgelegt, (siehe auch Greffenstette [Gre86]).

Mutation: *Mutation* bezeichnet im Zusammenhang mit EA die zufällige Änderung eines *Individuums* in Analogie zur *Mutation* in der Natur. Eine Möglichkeit ist, einzelne *Atome* der *Kodierungsstruktur*, die das *Individuum* darstellt, zufällig auszuwählen. Bei Binärstrings wird der entsprechende Wert dann invertiert, bei reellen Werten innerhalb vorgegebener Grenzen verändert. Der Vorgang findet am *Individuum* statt und wird erst durch die Dekodierung auf das *dekodierte Individuum* abgebildet.

Nachbarschaftsmodell: siehe *parallele Genetische Algorithmen.*

naturanaloges Verfahren: Ein *naturanaloges Verfahren* ist ein *Verfahren*, das auf Prinzipien aufbaut, die aus der Natur abgeleitet worden sind. In der Regel sind *naturanaloge Verfahren* auch *stochastische Verfahren.*

Neuronale Netze: *Neuronale Netze* (NN) sind ein Berechnungsmodell, bei dem sehr viele einfache Einheiten (Neuronen) miteinander verbunden sind und parallel zueinander arbeiten. Dieses Prinzip wurde von der Arbeitsweise menschlicher Nervenzellen übernommen. Daher besitzen *Neuronale Netze* Vorteile gegenüber herkömmlichen Berechnungsmodellen: Sie sind robust gegen den Ausfall einzelner Neuronen, und man kann mit ihnen Lern- und Abstraktionsfähigkeiten modellieren. (Zum Training *Neuronaler Netze* siehe auch Schaffer [SWE93])

Operator: Ein *Operator* ist ein in der Programmiersprache *LEA* beschriebener *Algorithmus*, der von EAGLE verarbeitet werden kann, oder ein in C++ in EAGLE implementierter *Algorithmus*. Bestandteile eines *Operators* sind *Parameter*, Variablen, *Label*, Operatoraufrufe, *Filter* und Kontrollkonstrukte. Die *Operatoren* können anhand der Eingabe- und Ausgabedatentypen klassifiziert werden.

parallele Genetische Algorithmen: Der klassische GA läßt sich aufgrund der globalen *Selektion* nur schwer auf mehrere Prozessoren verteilen. Im Gegensatz dazu sind folgende zwei Ansätze schon von der Idee her parallel:

- *Inselmodell* (engl: *coarse-grained-model*):
  tauscht periodisch *Individuen* zwischen Subpopulationen aus, die sich ansonsten unabhängig voneinander entwickeln. Dieses Modell wurde vorwiegend für MIMD-Rechner entwickelt.
- *Nachbarschaftsmodell* (engl: *fine-grained-model*):
  definiert ein toroidales Netz mit Nachbarschaftsbeziehungen zwischen den *Individuen*. *Crossover*, *Rekombination* und *Selektion* finden nur unter Nachbarn statt. Diese Variante ist massiv parallel und arbeitet typischerweise mit sehr großen *Populationen*. Dieses Modell paßt zur Architektur der SIMD-Rechner.

Parameter: Ein *Parameter* ist ein Variable zur Ablaufsteuerung von *Verfahren*, deren Wert bei der *Laufinitialisierung* gesetzt wird und innerhalb des *Verfahrens* normalerweise nicht geändert wird. Durch die Wahl der Parameterwerte wird die Arbeitsweise des *Verfahrens* bestimmt.
In der Sprache *LEA* zur Eingabe der Operatoren für EAGLE können die *Parameter* in *Operatoren* explizit deklariert werden. Dabei muß jeweils eine Defaultbelegung und optional ein Wertebereich angegeben werden. Wird ein *Lauf* gestoppt, können in der *Laufinitialisierung* die Werte der *Parameter* innerhalb des Wertebereiches geändert und das *Verfahren* dann fortgesetzt werden. Ein Beispiel ist die Deklaration der Mutationsrate im *LEA*–Code als *Parameter.*

Dieser *Parameter* kann während des *Laufs* modifiziert werden, indem der *Lauf* angehalten wird und in der *Laufinitialisierung* der entsprechende *Parameter* mit dem gewünschten Wert belegt wird.

Population: Eine *Population* ist eine Multimenge von *Individuen*.

Problem: Wir verstehen unter einem *Problem* die mathematisch formulierte Version eine Optimierungsproblems, wie z.B. das TSP oder Stundenplanerstellung, das durch einen EA angegangen werden soll. Ein *Problem* besteht aus *Problemstruktur* und *Fitneßfunktion*.

Problemstruktur: Die *Problemstruktur* bzw. die Struktur eines Problems besteht einer Liste von *Grunddatentypen* (*Atomen*).

Produktionsplanung: Die *Produktionsplanung* umfaßt alle Planungsbereiche in einem Unternehmen, die sich mit der Vorbereitung und Durchführung der Fertigung befassen. Darunter fällt z.B. die Maschinenbelegungsplanung (engl.: scheduling; auch als Ablauf- oder Reihenfolgeplanung bezeichnet), deren Aufgabe es ist, einen optimalen Terminplan für die zu fertigenden Aufträge in einem Produktionsbetrieb zu finden, (siehe auch Blohm [BBSS88]).

Rekombination: Unter einer *Rekombination* wird die Mischung der Inhalte verschiedener *Individuen* verstanden. Eine spezielle *Rekombination* ist der Crossover–Operator der *Genetischen Algorithmen*.

Selektion: Die *Selektion* ist ein fundamentales evolutionäres Prinzip. Bei der *Selektion* werden die *Individuen* ausgewählt, mit denen das *Verfahren* weitergeführt wird. Dies entspricht in der Evolutionstheorie dem Prinzip „survival of the fittest“. In *stochastischen Verfahren* wird eine Fülle unterschiedlicher Selektionsstrategien simuliert (z.B. elitistselection, roulette–wheelselection, usw.).

Simulated Annealing: *Simulated Annealing* (SA, dtsch. simulierte Abkühlung) ist ein Optimierverfahren, das auf der Idee einer thermodynamischen Optimierung aufbaut. Das *Verfahren* läßt sich wie folgt beschreiben:

1. In der lokalen Umgebung eines Punktes des *Suchraums* wird ein Kandidat als Nachfolgepunkt bestimmt.
2. Wenn der Kandidat eine bessere Fitneß hat als der ursprüngliche Punkt, so wird er als Nachfolger übernommen.
   Wenn der Kandidat eine schlechtere Fitneß hat als der ursprüngliche Punkt, so wird er mit einer gewissen Wahrscheinlichkeit trotzdem übernommen, um so lokalen Optima entkommen zu können. Diese Wahrscheinlichkeit hängt dabei vom Grad der Verschlechterung und einem Temperaturparameter T exponentiell ab ($P([alt := neu]) = exp(\frac{\Delta Fitneß}{T})$). Der Temperaturparameter wird dabei langsam gesenkt.

3. Wenn die Abbruchbedingung nicht erfüllt ist, gehe zu 1.

Wenn die Temperatur Null ist, entspricht *Simulated Annealing* einem *Hill–Climbing*, (siehe auch de Groot [dGWH90]).

stochastisches Verfahren: Ein *stochastisches Verfahren* ist ein *Verfahren*, in dem Entscheidungen aufgrund von Zufallszahlen getroffen werden.

Suchraum: Der *Suchraum* ist die Menge aller Wertebelegungen, die es zu einer *Problemstruktur* gibt.

Tcl–Interpreter: Interpreter für die Sprache Tcl (Tool Command Language). Diese ist eine Rapid–Prototyping–Sprache, in der schnell lauffähige Programme geschrieben werden können. Diese Programme werden dann von einem Interpreter abgearbeitet, (siehe dazu auch [Ous94]). Da zunächst geplant war, für den Interpreter von EAGLE den Tcl–Interpreter zu benutzen, wird im Zusammenhang mit EAGLE unter einem Tcl–Interpreter auch der Tcl–basierte *LEA*–Interpreter verstanden.

Travelling Salesman Problem: Das *Travelling Salesman Problem* besteht darin, zu einer Menge von Städten und gegebenen Distanzen zwischen je zwei Städten eine kürzeste Rundreise zu finden, bei der jede Stadt genau ein mal besucht wird. Dieses *Problem* ist NP-vollständig und u.a. deshalb eines der Standardprobleme für stochastische *Verfahren*, (siehe auch Grötschel/Holland [GH91]).

Threshold–Algorithmus: Der Threshold–Algorithmus (TA) ist eine Vereinfachung der Idee des *Simulated Annealing*. Hier wird allerdings der ebenfalls zu verringernde Temperaturparameter T direkt als Schwellwert eingesetzt und so der Rechenaufwand erheblich verringert. Das *Verfahren* läßt sich wie folgt beschreiben:

1. In der lokalen Umgebung eines Punktes des *Suchraums* wird ein Kandidat als Nachfolgepunkt bestimmt.
2. Wenn der Kandidat eine bessere Fitneß hat als der Ausgangspunkt, oder die Verschlechterung nicht größer ist als T, so wird der Kandidat übernommen. Der Temperaturparameter wird dabei langsam gesenkt.
3. Wenn die Abbruchbedingung nicht erfüllt ist, gehe zu 1.

(Siehe auch Dück/Scheuer [DS90])

Verfahren: Wir verstehen unter einem *Verfahren* einen *Evolutionären Algorithmus*, der eine *Population* übergeben bekommt und das beste gefundene *Individuum* zurückgibt. Ein *Verfahren* besteht aus einem *Hauptoperator*, der weitere *Operatoren* aufrufen kann.

A.2 Akronyme und Abkürzungen

CFS = Classifier System

EA = Evolutionärer Algorithmus

EAGLE = Evolutionary Algorithms Gaming and Learning Environment

EP = Evolutionäre Programmierung

ES = Evolutionsstrategie

GA = Genetischer Algorithmus

GDA = Great Deluge Algorithm

GP = Genetische Programmmierung

LEA = Language for Evolutionary Algorithms

NN = Neuronales Netz

PGA = Projektgruppe Genetische Algorithmen

PPS = Produktionsplanungssysteme

RRT = Record–to–Record Travel

SA = Simulated Annealing

TA = Threshold Algorithmus

TSP = Travelling Salesman Problem

Anhang B

Bestehende Systeme

Die folgende Liste bestehender nicht–kommerzieller EA-Systeme wurde aus dem FAQ zur Newsgroup comp.ai.genetics von Jörg Heitkötter[HB94] entnommen.

BUGS: BUGS (Better to Use Genetic Systems) is an interactive program for demonstrating the GENETIC ALGORITHM and is written in the spirit of Richard Dawkins' celebrated Blind Watchmaker software. The user can play god (or 'GA FITNESS function,' more accurately) and try to evolve lifelike organisms (curves). Playing with BUGS is an easy way to get an understanding of how and why the GA works. In addition to demonstrating the basic GENETIC OPERATORs (SELECTION, CROSSOVER, and MUTATION), it allows users to easily see and understand phenomena such as GENETIC DRIFT and premature convergence. BUGS is written in C and runs under Suntools and X Windows.

BUGS was written by Joshua Smith <jrs@media.mit.edu> at Williams College and is available by FTP from
santafe.edu:/pub/misc/BUGS/BUGS.tar.Z
and from
ftp.aic.nrl.navy.mil:/pub/galist/src/ga/BUGS.tar.Z
Note that it is unsupported software, copyrighted but freely distributable.
Address: Room E15-492, MIT Media Lab, 20 Ames Street, Cambridge, MA 02139. (Unverified 8/94).

DGenesis: DGenesis is a distributed implementation of a Parallel GA. It is based on Genesis 5.0. It runs on a network of UNIX workstations. It has been tested with DECstations, microVAXes, Sun Workstations and PCs running 386BSD 0.1. Each subpopulation is handled by a UNIX process and the communication between them is accomplished using Berkeley sockets. The system is programmed in C and is available free of charge by anonymous FTP from

lamport.rhon.itam.mx:/ and from
ftp.aic.nrl.navy.mil:/pub/galist/src/ga/dgenesis-1.0.tar.Z

DGenesis allows the user to set the MIGRATION interval, the migration rate and the topology between the SUB-POPULATIONs. There has not been much work investigating the effect of the topology on the PERFORMANCE of the GA, DGenesis was written specifically to encourage experimentation in this area. It still needs many refinements, but some may find it useful.

Contact Erick Cantu-Paz <ecantu@lamport.rhon.itam.mx> at the Instituto Tecnologico Autonomo de Mexico (ITAM)

Dougal: DOUGAL is a demonstration program for solving the TRAVELLING SALESMAN PROBLEM using GAs. The system guides the user through the GA, allowing them to see the results of altering parameters relating to CROSSOVER, MUTATION etc. The system demonstrates graphicaly the OPTIMIZATION of the route. The options open to the user to experiment with include percentage CROSSOVER and MUTATION, POPULATION size, steady state or generational replacement, FITNESS technique (linear normalised, is evaluation, etc).

DOUGAL requires an IBM compatible PC with a VGA monitor. The software is free, however I would appreciate feedback on what you think of the software.

Dougal is available by FTP from ENCORE (see Q15.3) in file
EC/GA/src/dougal.zip
It's pkzipped and contains executable, vga driver, source code and full documentation. It is important to place the vga driver (egavga.bgi) in the same directory as DOUGAL.
Author: Brett Parker, 7 Glencourse, East Boldon, Tyne + Wear, NE36 0LW, England. <b.s.parker@durham.ac.uk>

ESCaPaDE: ESCaPaDE is a sophisticated software environment to run experiments with Evolutionary Algorithms, such as e.g. an EVOLUTION STRATEGY. The main support for experimental work is provided by two internal tables: (1) a table of objective functions and (2) a table of so- called data monitors, which allow easy implementation of functions for monitoring all types of information inside the Evolutionary Algorithm under experiment.

ESCaPaDE 1.2 comes with the KORR implementation of the EVOLUTION STRATEGY by H.-P. Schwefel which offers simple and correlated MUTATIONs. KORR is provided as a FORTRAN 77 subroutine, and its cross-compiled C version is used internally by ESCaPaDE.

An extended version of the package was used for several investigations so far and has proven to be very reliable. The software and its documentation is fully

copyrighted although it may be freely used for scientific work; it requires 5-6 MB of disk space.

In order to obtain ESCaPaDE, please send a message to the e-mail address below. The SUBJECT line should contain 'help' or 'get ESCaPaDE'. (If the subject lines is invalid, your mail will be ignored!).
For more information contact: Frank Hoffmeister, Systems Analysis Research Group, LSXI, Department of Computer Science, University of Dortmund, D-44221 Dortmund, Germany.
Net: <hoffmeister@ls11.informatik.uni-dortmund.de>

Evolution Machine: The Evolution Machine (EM) is universally applicable to continuous (real-coded) OPTIMIZATION problems. In the EM we have coded fundamental evolutionary algorithms (GENETIC ALGORITHMs and EVOLUTION STRATEGIEs), and added some of our approaches to evolutionary search.

The EM includes extensive menu techniques with:

- Default parameter setting for unexperienced users.
- Well-defined entries for EM-control by freaks of the EM, who want to leave the standard process control.
- Data processing for repeated runs (with or without change of the strategy parameters).
- Graphical presentation of results: online presentation of the EVOLUTION progress, one-, two- and three-dimensional graphic output to analyse the FITNESS function and the evolution process.
- Integration of calling MS-DOS utilities (Turbo C).

We provide the EM-software in object code, which can be run on PC's with MS-DOS and Turbo C, v2.0, resp. Turbo C++,v1.01. The Manual to the EM is included in the distribution kit.

The EM software is available by FTP from
ftp-bionik.fb10.tu-berlin.de:/pub/software/Evolution-Machine/
This directory contains the compressed files em_tc.exe (Turbo C), em_tcp.exe (Turbo C++) and em_man.exe (the manual). There is also em-man.ps.Z, a compressed PostScript file of the manual. If you do not have FTP access, please send us either 5 1/4 or 3 1/2 MS-DOS compatible disks. We will return them with the compressed files (834 kB).

Official contact information: Hans-Michael Voigt or Joachim Born, Technical University Berlin, Bionics and EVOLUTION Techniques Laboratory, Bio- and Neuroinformatics Research Group, Ackerstrasse 71-76 (ACK1), D-13355 Berlin, Germany.
Net: <voigt,born@fb10.tu- berlin.de> (Unverified 8/94).

GAC, GAL: Bill Spears <spears@aic.nrl.navy.mil> writes: These are packages I've been using for a few years. GAC is a GA written in C. GAL is my Common Lisp version. They are similar in spirit to John Grefenstette's Genesis, but they don't have all the nice bells and whistles. Both versions currently run on Sun workstations. If you have something else, you might need to do a little modification.

Both versions are free: All I ask is that I be credited when it is appropriate. Also, I would appreciate hearing about improvements! This software is the property of the US Department of the Navy.

The code will be in a "shar" format that will be easy to install. This code is "as is", however. There is a README and some documentation in the code. There is NO user's guide, though (nor am I planning on writing one at this time). I am interested in hearing about bugs, but I may not get around to fixing them for a while. Also, I will be unable to answer many questions about the code, or about GAs in general. This is not due to a lack of interest, but due to a lack of free time! Available by FTP from
ftp.aic.nrl.navy.mil:/pub/galist/src/ga/GAC.shar.Z and GAL.shar.Z .
PostScript versions of some papers are under "/pub/spears". Feel free to browse.

GAGA: GAGA (GA for General Application) is a self-contained, re-entrant procedure which is suitable for the minimization of many "difficult" cost functions. Originally written in Pascal by Ian Poole, it was rewritten in C by Jon Crowcroft. GAGA can be obtained by request from the author: Jon Crowcroft <jon@cs.ucl.ac.uk>, Univeristy College London, Gower Street, London WCIE 6BT, UK, or by FTP from
ftp://cs.ucl.ac.uk:/darpa/gaga.shar

GAGS: GAGS 0.92 (Genetic Algorithms from Granada, Spain) is a library and companion programs written and designed to take the heat out of designing a GENETIC ALGORITHM. It features a class library for genetic algorithm programming, but, from the user point of view, is a genetic algorithm application generator. Just write the function you want to optimize, and GAGS surrounds it with enough code to have a genetic algorithm up and running, compiles it, and runs it. GAGS Is written in C++, so that it can be compiled in any platform running this GNU utility. It has been tested on various machines. Documentation is available.

GAGS includes:

- Steady-state, roulette-wheel, tournament and elitist SELECTION.
- FITNESS evaluation using training files.

- Graphics output through gnuplot.
- Uniform and 2-point CROSSOVER, and bit-flip and gene-transposition MUTATION.
- Variable length CHROMOSOMEs and related operators.

The application generator gags.pl is written in perl, so this language must also be installed before GAGS. Available by FTP from:
kal-el.ugr.es:/pub/GAGS-0.92.tar.gz
The programmer's manual is in the same directory, file gagsprogs.ps.gz. GAGS is also available from ENCORE (see Q15.3) in file EC/GA/src/gags-0.92.tar.gz with documentation in EC/GA/docs/gagsprog.ps.gz

Maintained by J.J. Merelo, Grupo Geneura, Univ. Granada <jmerelo@kal-el.ugr.es>

GALOPPS: GALOPPS (Genetic Algorithm Optimized for Portability and Parallelism) is a flexible, generic GA, based upon SGA-C. It has been extended to provide three types of island parallelism, ranging from a single PC simulating parallel subpopulations, to multiple computers on a network. It's been tested on a wide variety of DOS and UNIX machines. An 80-page User Guide is provided.

GALOPPS extends the SGA capabilities several fold:

- 5 SELECTION methods.
- Random or superuniform initialization of binary CHROMOSOMEs.
- 3 CROSSOVER routines for value-based representations, and 4 for order-based reps.
- 3 MUTATION routines.
- 4 FITNESS scaling routines.
- Various replacement strategy options, including crowding replacement and new incest-reduction option.
- Elitism is optional.
- Convergence: lost, CONVERGED, percent converged, etc.
- Various PERFORMANCE measures
- Uses "SGA philosophy" of one template file for the user to modify, but enriched with many additional user callbacks, for added flexibility, extensibility.

Ten sample applications are provided – "standard" ones are Goldberg's three examples, Holland's Royal Road "Challenge" problem, and a blind traveling salesperson problem.

For portability, the user interface in the standard distribution is non-graphical. A number of GUIs are in development.

GALOPPS Release 2.20 and manual v2.20.ps are available by FTP from isl.cps.msu.edu:/pub/GA/GALOPPS2.20/

Contact: Erik Goodman, Genetic Algorithms Research and Applications Group (GARAGe), Computer Science and Case Center for Computer-Aided Engineering and Manufacturing, 112 Engineering Building, Michigan State University, East Lansing 48824. <goodman@egr.msu.edu>

GAMusic: GAMusic 1.0 is a user-friendly interactive demonstration of a simple GA that evolves musical melodies. Here, the user is the FITNESS function. Melodies from the POPULATION can be played and then assigned a fitness. Iteration, RECOMBINATION frequency and MUTATION frequency are all controlled by the user. This program is intended to provide an introduction to GAs and may not be of interest to the experienced GA programmer.

GAMusic was programmed with Microsoft Visual Basic 3.0 for Windows 3.1x. No special sound card is required. GAMusic is distributed as shareware (cost $10) and can be obtained by FTP from
wuarchive.wustl.edu:/pub/MSDOS_UPLOADS/GenAlgs/gamusic.zip
or from
fly.bio.indiana.edu:/science/ibmpc/gamusic.zip
The program is also available from the America Online archive.

Contact: Jason H. Moore <jhm@superh.hg.med.umich.edu> or <jasonUMICH@aol.com>

GANNET: GANNET (Genetic Algorithm / Neural NETwork) is a software package written by Jason Spofford in 1990 which allows one to evolve neural networks. It offers a variety of configuration options related to rates of the GENETIC OPERATORs. GANNET evolves nets based upon three FITNESS functions: Input/Output Accuracy, Output 'Stability', and Network Size.

The evolved neural network presently has a binary input and binary output format, with neurodes that have either 2 or 4 inputs and weights ranging from -3 to +4. GANNET allows for up to 250 neurodes in a net. Research using GANNET is continuing and version 2.0 will be released in early 1995.

GANNET is available by FTP from fame.gmu.edu:/gannet/source/ There are separate directories for GANNET itself, a verifier program which verifies the best neural network generated (/gannet/verifier), and some sample datasets (/gannet/datasets). Further, Spofford's masters thesis descrribing GANNET is available in postscript format (/gannet/thesis).

Contact: Darrell Duane or Dr. Kenneth Hintz, George Mason University, Dept. of Electrical & Computer Engineering, Mail Stop 1G5, 4400 University Drive,

Fairfax, VA 22033-4444 USA.
Net: <dduane@fame.gmu.edu> or <khintz@fame.gmu.edu>

GAucsd: GAucsd is a Genesis-based GA package incorporating numerous bug fixes and user interface improvements. Major additions include a wrapper that simplifies the writing of evaluation functions, a facility to distribute experiments over networks of machines, and Dynamic Parameter Encoding, a technique that improves GA PERFORMANCE in continuous SEARCH SPACEs by adaptively refining the genomic representation of real-valued parameters.

GAucsd was written in C for Unix systems, but the central GA engine is easily ported to other platforms. The entire package can be ported to systems where implementations of the Unix utilities "make", "awk" and "sh" are available.

GAucsd is available by FTP from
cs.ucsd.edu:/pub/GAucsd/GAucsd14.sh.Z
or from
ftp.aic.nrl.navy.mil:/pub/galist/src/ga/GAucsd14.sh.Z
To be added to a mailing list for bug reports, patches and updates, send "add GAucsd" to <listserv@cs.ucsd.edu>.

Cognitive Computer Science Research Group, CSE Department, UCSD 0114, La Jolla, CA 92093-0114, USA.
Net: <GAucsd-request@cs.ucsd.edu>

GA Workbench: A mouse-driven interactive GA demonstration program aimed at people wishing to show GAs in action on simple FUNCTION OPTIMIZATIONs and to help newcomers understand how GAs operate. Features: problem functions drawn on screen using mouse, run-time plots of GA POPULATION distribution, peak and average FITNESS. Useful population STATISTICS displayed numerically, GA configuration (population size, GENERATION gap etc.) performed interactively with mouse. Requirements: MS-DOS PC, mouse, EGA/VGA display.

Available by FTP from the simtel20 archive mirrors,
e.g. wsmr- simtel20.army.mil:/pub/msdos/neurlnet/gaw110.zip or
wuarchive.wustl.edu: or oak.oakland.edu:
Produced by Mark Hughes <mrh@i2ltd.demon.co.uk>. A windows version is in preparation.

GECO: GECO (Genetic Evolution through Combination of Objects) is an extensible, object-oriented framework for prototyping GENETIC ALGORITHMs in Common Lisp. GECO makes extensive use of CLOS, the Common Lisp Object System, to implement its functionality. The abstractions provided by the classes have been chosen with the intent both of being easily understandable to anyone familiar with the paradigm of genetic algorithms, and of providing the algorithm developer with the ability to customize all aspects of its operation.

It comes with extensive documentation, in the form of a PostScript file, and some simple examples are also provided to illustrate its intended use.

GECO Version 2.0 is available by FTP. See the file
ftp.aic.nrl.navy.mil:/pub/galist/src/ga/GECO-v2.0.README
for more information.

George P. W. Williams, Jr., 1334 Columbus City Rd., Scottsboro, AL 35768.
Net: <george@hsvaic.hv.boeing.com>.

Genesis: Genesis is a generational GA system written in C by John Grefenstette. As the first widely available GA program Genesis has been very influential in stimulating the use of GAs, and several other GA packages are based on it. Genesis is available together with OOGA (see below), or by FTP from
ftp.aic.nrl.navy.mil:/pub/galist/src/ga/genesis.tar.Z
(Unverified 8/94).

GENEsYs: GENEsYs is a Genesis-based GA implementation which includes extensions and new features for experimental purposes, such as SELECTION schemes like linear ranking, Boltzmann, (mu, lambda)-selection, and general extinctive selection variants, CROSSOVER operators like n-point and uniform crossover as well as discrete and intermediate RECOMBINATION. SELF-ADAPTATION of MUTATION rates is also possible.

A set of objective functions is provided, including De Jong's functions, complicated continuous functions, a TSP-problem, binary functions, and a fractal function. There are also additional data- monitoring facilities such as recording average, variance and skew of OBJECT VARIABLES and MUTATION rates, or creating bitmap-dumps of the POPULATION.

GENEsYs 1.0 is available via FTP from
lumpi.informatik.uni-dortmund.de:/pub/GA/src/GENEsYs-1.0.tar.Z
The documentation alone is available as
/pub/GA/docs/GENEsYs-1.0-doc.tar.Z

For more information contact: Thomas Baeck, Systems Analysis Research Group, LSXI, Department of Computer Science, University of Dortmund, D-44221 Dortmund, Germany.
Net: <baeck@ls11.informatik.uni- dortmund.de> (Unverified 8/94).

GenET: GenET is a "generic" GA package. It is generic in the sense that all problem independent mechanisms have been implemented and can be used regardless of application domain. Using the package forces (or allows, however you look at it) concentration on the problem: you have to suggest the best representation, and the best operators for such space that utilize your problem-specific knowledge. You do not have to think about possible GA models or their implementation.

The package, in addition to allowing for fast implementation of applications and being a natural tool for comparing different models and strategies, is intended to become a depository of representations and operators. Currently, only floating point representation is implemented in the library with few operators.

The algorithm provides a wide selection of models and choices. For example, POPULATION models range from generational GA, through steady-state, to (n,m)-EP and (n,n+m)-EP models (for arbitrary problems, not just parameter OPTIMIZATION). (Some are not finished at the moment). Choices include automatic adaptation of operator probabilities and a dynamic ranking mechanism, etc.

Even though the implementation is far from optimal, it is quite efficient - implemented in ATT's C++ (3.0) (functional design) and also tested on gcc. Along with the package you will get two examples. They illustrate how to implement problems with heterogeneous and homogeneous structures, with explicit rep/opers and how to use the existing library (FP). Very soon I will place there another example - our GENOCOP operators for linearly constrained OPTIMIZATION. One more example soon to appear illustrates how to deal with complex structures and non-stationary problems - this is a fuzzy rule-based controller optimized using the package and some specific rep/operators.

If you start using the package, please send evaluations (especially bugs) and suggestions for future versions to the author.

GenET Version 1.00 is available by FTP from
radom.umsl.edu:/var/ftp/GenET.tar.Z
To learn more, you may get the User's Manual, available in compressed postscript in "/var/ftp/userMan.ps.Z". It also comes bundled with the complete package.

Cezary Z. Janikow, Department of Math and CS, CCB319, St. Louis, MO 63121, USA.
Net: <janikow@radom.umsl.edu>

Genie: Genie is a GA-based modeling/forecasting system that is used for long-term planning. One can construct a model of an ENVIRONMENT and then view the forecasts of how that environment will evolve into the future. It is then possible to alter the future picture of the environment so as to construct a picture of a desired future (I will not enter into arguments of who is or should be responsible for designing a desired or better future). The GA is then employed to suggest changes to the existing environment so as to cause the desired future to come about.

Genie is available free of charge via e-mail or on 3.5" disk from: Lance Chambers, Department of Transport, 136 Stirling Hwy, Nedlands, West Australia 6007. Net: <pstamp@yarrow.wt.uwa.edu.au> It is also available by FTP from

hiplab.newcastle.edu.au:/pub/Genie&Code.sea.Hqx

Genitor: "Genitor is a modular GA package containing examples for floating- point, integer, and binary representations. Its features include many sequencing operators as well as subpopulation modeling.

The Genitor Package has code for several order based CROSSOVER operators, as well as example code for doing some small TSPs to optimality.

We are planning to release a new and improved Genitor Package this summer (1993), but it will mainly be additions to the current package that will include parallel island models, cellular GAs, delta coding, perhaps CHC (depending on the legal issues) and some other things we have found useful."

Genitor is available from Colorado State University Computer Science Department by FTP from
ftp.cs.colostate.edu:/pub/GENITOR.tar

Please direct all comments and questions to <mathiask@cs.colostate.edu>. If these fail to work, contact: L. Darrell Whitley, Dept. of Computer Science, Colorado State University, Fort Collins, CO 80523, USA.
Net: <whitley@cs.colostate.edu> (Unverified 8/94).

GENlib: GENlib is a library of functions for GENETIC ALGORITHMs. Included are two applications of this library to the field of neural networks. The first one called "cosine" uses a genetic algorithm to train a simple three layer feed-Forward network to work as a cosine-function. This task is very difficult to train for a backprop algorithm while the genetic algorithm produces good results. The second one called "vartop" is developing a Neural Network to perform the XOR-function. This is done with two genetic algorithms, the first one develops the topology of the network, the second one adjusts the weights. GENlib may be obtained by FTP from
ftp.neuro.informatik.uni-kassel.de:/pub/NeuralNets/GA-and-NN/

Author: Jochen Ruhland, FG Neuronale Netzwerke / Uni Kassel, Heinrich-Plett-Str. 40, D-34132 Kassel, Germany.
Net: <jochenr@neuro.informatik.uni-kassel.de>

GENOCOP: This is a GA-based OPTIMIZATION package that has been developed by Zbigniew Michalewicz and is described in detail in his book "Genetic Algorithms + Data Structures = Evolution Programs" (Springer Verlag, 2nd ed, 1994).

GENOCOP (Genetic Algorithm for Numerical Optimization for COnstrained Problems) optimizes a function with any number of linear constraints (equalities and inequalities).

The second version of the system is available by FTP from
ftp.uncc.edu:/coe/evol/genocop2.tar.Z

Zbigniew Michalewicz, Dept. of Computer Science, University of North Carolina, Chappel-Hill, NC, USA.
Net: <zbyszek@uncc.edu>

GIGA: GIGA is designed to propogate information through a POPULATION, using CROSSOVER as its operator. A discussion of how it propogates BUILDING BLOCKs, similar to those found in Royal Road functions by John Holland, is given in the DECEPTION section of: "Genetic Invariance: A New Paradigm for Genetic Algorithm Design." University of Alberta Technical Report TR92-02, June 1992. See also: "GIGA Program Description and Operation" University of Alberta Computing Science Technical Report TR92-06, June 1992

These can be obtained, along with the program, by FTP from
ftp.cs.ualberta.ca:/pub/TechReports/ in the subdirectories TR92-02/ and TR92-06/ .

Also, the paper "Mutation-Crossover Isomorphisms and the Construction of Discriminating Functions" gives a more in-depth look at the behavior of GIGA. Its is available from
ftp.cs.ualberta.ca:/pub/joe/Preprints/xoveriso.ps.Z

Joe Culberson, Department of Computer Science, University of Alberta, CA.
Net: <joe@cs.ualberta.ca>

GPEIST: The Genetic Programming Environment in Smalltalk (GPEIST) provides a framework for the investigation of Genetic Programming within a ParcPlace VisualWorks 2.0 development system. GPEIST provides program, POPULATION, chart and report browsers and can be run on HP/Sun/PC (OS/2 and Windows) machines. It is possible to distribute the experiment across several workstations - with subpopulation exchange at intervals - in this release 4.0a. Experiments, populations and INDIVIDUAL genetic programs can be saved to disk for subsequent analysis and experimental statistical measures exchanged with spreadsheets. Postscript printing of charts, programs and animations is supported. An implementation of the Ant Trail problem is provided as an example of the use of the GPEIST environment.

GPEIST is available from ENCORE (see Q15.3) in file:
EC/GP/src/GPEIST4.tar.gz

Contact: Tony White, Bell-Northern Research Ltd., Computer Research Lab - Gateway, 320 March Road, Suite 400, Kanata, Ontario, Canada, K2K 2E3.
Tel: (613) 765-4279 <arpw@bnr.ca>

Imogene: Imogene is a Windows 3.1 shareware program which generates pretty images using GENETIC PROGRAMMING. The program displays GENERA-

TIONs of 9 images, each generated using a formula applied to each pixel. (The formulae are initially randomly computed). You can then select those images you prefer. In the next generation, the nine images are generated by combining and mutating the formulae for the most- preferred images in the previous generation. The result is a SIMULATION of natural SELECTION in which images evolve toward your aesthetic preferences.

Imogene supports different color maps, palette animation, saving images to .BMP files, changing the wallpaper to nice images, printing images, and several other features. Imogene works only in 256 color mode and requires a floating point coprocessor and a 386 or better CPU.

Imogene is based on work originally done by Karl Sims at (ex-)Thinking Machines for the CM-2 massively parallel computer - but you can use it on your PC. You can FTP Imogene from:
ftp.cc.utexas.edu:/pub/genetic-programming/code/imogenes.zip

Contact: Harley Davis, ILOG S.A., 2 Avenue Gallini, BP 85, 94253 Gentilly Cedex, France.
Tel: +33 1 46 63 66 66 <davis@ilog.fr>

LibGA: LibGA is a library of routines written in C for developing GENETIC ALGORITHMs. It is fairly simple to use, with many knobs to turn. Most GA parameters can be set or changed via a configuration file, with no need to recompile. (E.g., operators, pool size and even the data type used in the CHROMOSOME can be changed in the configuration file.) Function pointers are used for the GENETIC OPERATORs, so they can easily be manipulated on the fly. Several genetic operators are supplied and it is easy to add more. LibGA runs on many systems/architectures. These include Unix, DOS, NeXT, and Amiga.

LibGA Version 1.00 is available by FTP from
ftp.aic.nrl.navy.mil:/pub/galist/src/ga/libga100.tar.Z
or by email request to its author, Art Corcoran <corcoran@wiltel.com> (Unverified 8/94).

LICE: LICE is a parameter OPTIMIZATION program based on EVOLUTION STRATEGIEs (ES). In contrast to classic ES, LICE has a local SELECTION scheme to prevent premature stagnation. Details and results were presented at the EP'94 conference in San Diego. LICE is written in ANSI-C (more or less), and has been tested on Sparc-stations and Linux-PCs. If you want plots and graphics, you need X11 and gnuplot. If you want a nice user interface to create parameter files, you also need Tk/Tcl.

LICE-1.0 is available as source code by FTP from
lumpi.informatik.uni-dortmund.de:/pub/ES/src/LICE-1.0.tar.gz

Author: Joachim Sprave <joe@ls11.informatik.uni-dortmund.de>

Matlab-GA: The MathWorks FTP site has some Matlab GA code in the directory ftp.mathworks.com:/pub/contrib/optim/genetic It's a bunch of .m files that implement a basic GA.

mGA: mGA is an implementation of a messy GA as described in TCGA report No. 90004. Messy GAs overcome the linkage problem of simple GENETIC ALGORITHMs by combining variable-length strings, GENE expression, messy operators, and a nonhomogeneous phasing of evolutionary processing. Results on a number of difficult deceptive test functions have been encouraging with the messy GA always finding global optima in a polynomial number of function evaluations.

See TCGA reports 89003, 90005, 90006, and 91004, and IlliGAL report 91008 for more information on messy GAs (See Q14). The C language version is available by FTP from IlliGAL in the directory
gal4.ge.uiuc.edu:/pub/src/messyGA/C/

PARAGenesis: PARAGenesis is the result of a project implementing Genesis on the CM-200 in C*. It is an attempt to improve PERFORMANCE as much as possible without changing the behavior of the GENETIC ALGORITHM. Unlike the punctuated equilibria and local SELECTION models, PARAGenesis doesn't modify the genetic algorithm to be more parallelizable as these modifications can drastically alter the behavior of the algorithm. Instead each member is placed on a separate processor allowing initialization, evaluation and MUTATION to be completely parallel. The costs of global control and communication in selection and CROSSOVER are present but minimized as much as possible. In general PARAGenesis on an 8k CM-200 seems to run 10-100 times faster than Genesis on a Sparc 2 and finds equivalent solutions.

PARAGenesis includes all the features of serial Genesis plus some additions. The additions include the ability to collect timing STATISTICS, probabilistic SELECTION (as opposed to Baker's stochastic universal sampling), uniform CROSSOVER and local or neighborhood SELECTION. Anyone familiar with the serial implementation of Genesis and C* should have little problem using PARAGenesis.

PARAGenesis is available by FTP from
ftp.aic.nrl.navy.mil:/pub/galist/src/ga/paragenesis.tar.Z

DISCLAIMER: PARAGenesis is fairly untested at this point and may contain some bugs.

Michael van Lent, Advanced Technology Lab, University of Michigan, 1101 Beal Av., Ann Arbor, MI 48109, USA.
Net: <vanlent@eecs.umich.edu>.

PGA: PGA is a simple testbed for basic explorations in GENETIC ALGORITHMs. Command line arguments control a range of parameters, there are a number of

built-in problems for the GA to solve. The current set includes:

- maximize the number of bits set in a CHROMOSOME
- De Jong's functions DJ1, DJ2, DJ3, DJ5
- binary F6, used by Schaffer et al
- a crude 1-d knapsack problem; you specify a target and a set of numbers in an external file, GA tries to find a subset that sums as closely as possible to the target
- the 'royal road' function(s); a CHROMOSOME is regarded as a set of consecutive blocks of size K, and scores K for each block entirely filled with 1s, etc; a range of parameters.
- max contiguous bits, you choose the ALLELE range.
- timetabling, with various smart MUTATION options; capable of solving a good many real-world timetabling problems (has done so)

Lots of GA options: rank, roulette, tournament, marriage-tournament, spatially-structured SELECTION; one-point, two-point, uniform or no CROSSOVER; fixed or adaptive MUTATION; one child or two; etc.

Default output is curses-based, with optional output to file; can be run non-interactively too for batched series of experiments.

It's easy to add your own problems. CHROMOSOMEs are represented as character arrays, so you are not (quite) stuck with bit-string problem encodings.

PGA has been used for teaching for a couple of years now, and has been used as a starting point by a fair number of people for their own projects. So it's reasonably reliable. However, if you find bugs, or have useful contributions to make, Tell Me! It is available by FTP from
ftp.dai.ed.ac.uk:pub/pga-2.7/pga-2.7.tar.Z
(see the file pga.README in the same directory for more information)

Peter Ross, Department of AI, University of Edinburgh, 80 South Bridge, Edinburgh EH1 1HN, UK.
Net: <peter@aisb.ed.ac.uk>.

SGA-C, SGA-Cube: SGA-C is a C-language translation and extension of the original Pascal SGA code presented in Goldberg's book [GOLD89]. It has some additional features, but its operation is essentially the same as that of the Pascal version. SGA-C is described in TCGA report No. 91002.

SGA-Cube is a C-language translation of Goldberg's SGA code with modifications to allow execution on the nCUBE 2 Hypercube Parallel Computer. When run on the nCUBE 2, SGA-Cube can take advantage of the hypercube

architecture, and is scalable to any hypercube dimension. The hypercube implementation is modular, so that the algorithm for exploiting parallel processors can be easily modified.

In addition to its parallel capabilities, SGA-Cube can be compiled on various serial computers via compile-time options. In fact, when compiled on a serial computer, SGA-Cube is essentially identical to SGA-C. SGA-Cube is described in TCGA report No. 91005.

Each of these programs is distributed in the form of a Unix shar file, available via e-mail or on various formatted media by request from: Robert Elliott Smith, Department of Engineering of Mechanics, Room 210 Hardaway Hall,, The University of Alabama P.O. Box 870278, Tuscaloosa, Alabama 35487, USA.
Net: <rob@comec4.mh.ua.edu>.

SGA-C and SGA-Cube are also available in compressed tar form by FTP from ftp.aic.nrl.navy.mil:/pub/galist/src/ga/sga-c.tar.Z and sga- cube.tar.Z .

Splicer: Splicer is a GENETIC ALGORITHM tool created by the Software Technology Branch (STB) of the Information Systems Directorate at NASA/Johnson Space Center with support from the MITRE Corporation. Splicer has well-defined interfaces between a GA kernel, representation libraries, FITNESS modules, and user interface libraries.

The representation libraries contain functions for defining, creating, and decoding genetic strings, as well as multiple CROSSOVER and MUTATION operators. Libraries supporting binary strings and permutations are provided, others can be created by the user.

FITNESS modules are typically written by the user, although some sample applications are provided. The modules may contain a fitness function, initial values for various control parameters, and a function which graphically displays the best solutions.

Splicer provides event-driven graphic user interface libraries for the Macintosh and the X11 window system (using the HP widget set); a menu-driven ASCII interface is also available though not fully supported. The extensive documentation includes a reference manual and a user's manual; an architecture manual and the advanced programmer's manual are currently being written.

An electronic bulletin board (300/1200/2400 baud, 8N1) with information regarding Splicer can be reached at (713) 280-3896 or (713) 280-3892. Splicer is available free to NASA and its contractors for use on government projects by calling the STB Help Desk weekdays 9am-4pm CST at (713) 280-2233. Government contractors should have their contract monitor call the STB Help Desk; others may purchase Splicer for $221 (incl. documentation) from: COSMIC, 382 E. Broad St., Athens, GA 30602, USA. (Unverified 8/94). Last known

address <bayer@galileo.jsc.nasa.gov>. (Steve Bayer). This now bounces back with "user unknown".

TOLKIEN: TOLKIEN (TOoLKIt for gENetics-based applications) is a C++ class library, intended for those involved in GAs and CLASSIFIER SYSTEM research with a working knowledge of C++. It is designed to reduce effort in developing genetics-based applications by providing a collection of reusable objects. For portability, no compiler specific or class library specific features are used. The current version has been compiled successfully using Borland C++ Version 3.1 and GNU C++.

TOLKIEN contains a lot of useful extensions to the generic GENETIC ALGORITHM and CLASSIFIER SYSTEM architecture. Examples include: (i) CHROMOSOMEs of user-definable types; binary, character, integer and floating point; (ii) Gray code encoding and decoding; (iii) multi- point and uniform CROSSOVER; (iv) diploidy and dominance; (v) various SELECTION schemes such as tournament selection and linear ranking; (vi) linear FITNESS scaling and sigma truncation; (vii) the simplest one-taxon-one-action classifiers and the general two-taxa-one-action classifiers.

TOLKIEN is available from ENCORE (See Q15.3) in file:
GA/src/TOLKIEN.tar.gz
The documentation and two primers on how to build GA and CFS applications alone are available as:
GA/docs/tolkien-doc.tar.gz

Author: Anthony Yiu-Cheung Tang <tang028@cs.cuhk.hk>. Department of Computer Science (Rm 913), The Chinese University of Hong Kong.
Tel: 609-8403, 609-8404.

WOLF: This is a simulator for the G/SPLINES (genetic spline models) algorithm which builds spline-based functional models of experimental data, using CROSSOVER and MUTATION to evolve a POPULATION towards a better fit. It is derived from Friedman's MARS models. The original work was presented at ICGA-4, and further results including additional basis function types such as B-splines have been presented at the NIPS-91 meeting.

Available free by FTP by contacting the author; runs on SUN (and possibly any SYSV) UNIX box. Can be redistributed for noncommercial use. Simulator includes executable and C source code; a technical report (RIACS tech report 91.10) is also available.

David Rogers, MS Ellis, NASA Ames Research Center, Moffett Field, CA 94035, USA.
Net: <drogers@msi.com>.

Anhang C

Syntax von LEA

Im folgenden ist die Syntax von LEA abgedruckt. Als Notation wird die Extended Backus–Naur–Form (EBNF) verwendet, wobei bei einer Aufzählung, deren Elemente unmittelbar verständlich sind, mit ... abgekürzt wird, wie z.B. in `( "a" | ... | "z" )`. Die Wurzel (das Startsymbol) der Syntax ist `program`. In der vorliegenden Fassung sind verschiedene Kombinationen von LEA-Konstrukten noch zulässig (zum Beispiel der Ausdruck `1 AND 5`), die später in der semantischen Phase der Übersetzung ausgeschlossen werden.

```
program          = define formalparameter ":" simpletype ";" block .
define           = ( "FITNESSOP" | "CHECKOP" | "PROCEDURE"
                   | "OPERATOR" | "CREATEIND" ) .
block            = { declaration } "BEGIN" statementsequence "END" .
declaration      = ( "CONST" constdeclaration { constdeclaration }
                   | "PARAMETER" paramdeclarationlist
                   | "VAR" declarationlist
                   | "IMPORT" ident ";" ) .
number           = ( valueinteger | valuereal | valuebit ) .
valueinteger     = [ sign ] digit { digit } .
digit            = ( "0" | "1" | ... | "9" ) .
sign             = ( "+" | "-" ) .
valuereal        = digit { digit } "." { digit } [ "E" [ sign ]
                   digit { digit } ] .
valuebit         = ( "0" | "1" ) .
statementsequence = statement { ";" statement } .
basictype        =  ( "INTEGER" | "REAL" | "BIT" ) .
eatype           = ( "POPULATION" | "INDIVIDUAL" ) .
subrangetype     = basictype [ range ] .
simpletype       = ( eatype | basictype | permutationtype ) .
permutationtype  = "PERMUTATION" [ "[" expression "]" ] .
```

```
arraytype        = "ARRAY" range { range } "OF" simpletype .
constdeclaration = ident ":=" expression ";"
letter           = ( "a" | "b" | ... | "z" | "A" | ... | "Z" ) .
ident            = letter { letter | digit } .
expression       = simpleexpr [ relation simpleexpr ] .
relation         = ( "<" | "<=" | "=" | ">" | ">=" | "<>" ) .
simpleexpr       = [ sign ] term { addoperator term } .
addoperator      = ( "+" | "-" | "OR" ) .
term             = factor { muloperator factor } .
muloperator      = ( "*" | "/" | "AND" ) .
factor           = ( designator | number | "(" expression ")"
                   | "NOT" factor ) .
type             = ( simpletype | arraytype ) .
designator       = ident [ opteacall | optindex ] .
optindex         = "[" expression { "," expression } "]" .
statement        = [ assignmentorea | ifstatement | whilestatement
                   | forstatement | foreachstatement
                   | returnstatement | labelstatement
                   | filterstatement ] .
assignmentorea   = designator optassignment .
ifstatement      = "IF" expression "THEN" statementsequence
                   [ "ELSE" statementsequence ] "END" .
whilestatement   = "WHILE" expression "DO" statementsequence "END" .
forstatement     = "FOR" ident ":=" expression "TO" expression
                   "DO" statementsequence "END" .
foreachstatement = "FOREACH" ident "IN" ident "DO"
                   statementsequence "END" .
opteacall        = "(" [ oneparam { "," oneparam } ] ")" .
returnstatement  = "RETURN" expression .
formalparameter  = "(" [ paramsection { ";" paramsection } ] ")" .
paramsection     = [ "VAR" ] ident { "," [ VAR ] ident } ":"
                   simpletype .
oneparam         = ( ident [ mapping ] | string ) .
mapping          = "{" onemap { "," onemap } "}" .
onemap           = basictype "(" expression ".." expression ")" .
optassignment    = [ ":=" expression ] .
labelstatement   = "LABEL" ident .
filterstatement  = "FILTER" ident ident expression .
range            = "[" number ".." number "]" .
paramdeclarationlist = paramdecline ";" { paramdecline ";" } .
declarationlist  = decline ";" { decline ";" } .
paramdecline     = vpidentfield { "," vpidentfield } ":"
                   subrangetype .
```

```
decline          = vpidentfield { "," vpidentfield } ":" type .
vpidentfield     = [ "STATIC" ] ident [ ":=" expression ] .
```

Anhang D

Grobentwurf

D.1 Einleitung

Dieser Teil der Dokumentation soll einen groben Überblick über eine mögliche Struktur der Impelemntierung von EAGLE geben und außerdem die Grundlage für einen Feinentwurf darstellen.

Dazu werden *Datenelemente* und *Aktionselemente* definiert sowie Interaktion und Datenaustausch zwischen den *Aktionselementen* beschrieben. Außerdem werden Verwaltung und Modifikation von *Datenelementen* durch *Aktionselemente* skizziert.

Datenelemente dienen dazu, programmrelevante Daten zu speichern und für Modifikation oder Datenaustausch verfügbar zu machen.

Aktionselemente sind Programmteile, die jeweils bestimmte Teilmengen der durch Pflichtenheft und Spezifikation geforderten Aktionen und Operationen ausführen und dabei in bestimmter Weise *Datenelemente* verwenden bzw. modifizieren.

D.2 Beschreibung der Aktionselemente

In diesem Abschnitt werden die Aktionselemente beschrieben. Dazu gehören die Aufgaben der Aktionselemente, aber auch die Verwaltung der von ihnen benutzten Datenelemente einschließlich ihrer konkreten Verwendung sowie die Interaktion mit anderen Aktionselementen. Nachfolgende Skizze zeigt die groben Komponenten des Systems im Überblick:

Interne Aktionselemente sind Teil des zu erstellenden Systems, externe Aktionselemente werden von der Systemumgebung eingebunden.

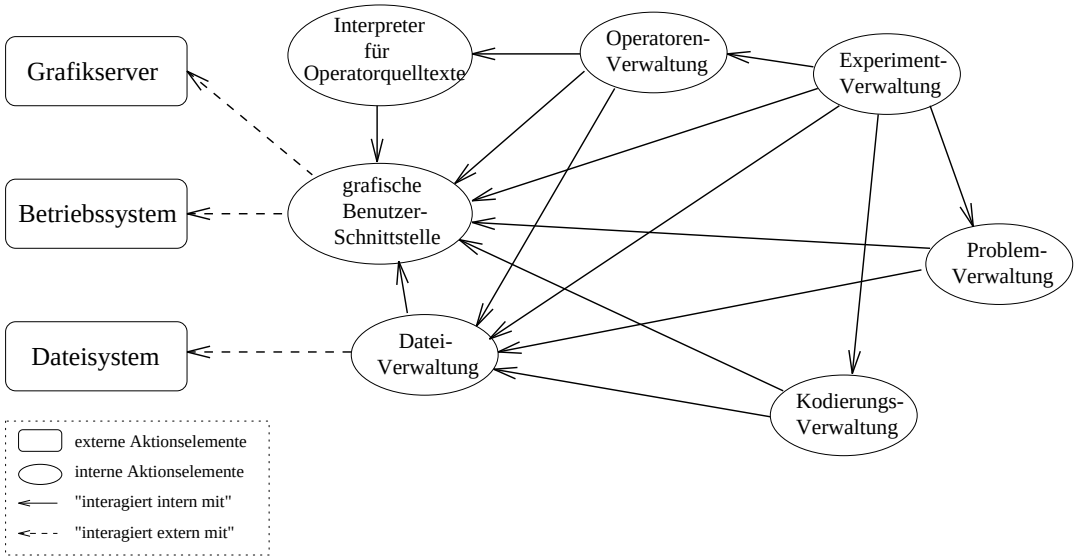


Abbildung D.1: Interaktion der Aktionselemente

grafische Benutzerschnittstelle

- Aufgaben:
  - Bereitstellung rudimentärer Grafikelemente zur Benutzerinteraktion mittels Grafikoberfläche: Anzeigen von Meldungen, Eingabe von Daten und Ablaufsteuerung.
- Interaktion:
  - extern mit Betriebssystem und Grafikserver.

Dateiverwaltung

- Aufgaben:
  - Öffnen und Schließen benötigter Dateien mit vorgegebenen Zugriffsmöglichkeiten, sowie Organisation der Lese- bzw. Schreibzugriffe darauf.
- Datenverwaltung:
  - *Dateidaten* zu jeder geöffneten Datei mit
    - einer *Zeichenkette* für den eindeutigen Dateinamen

    * einer *Zeichenkette* für den verwendeten Dateipfad
    * einer *Ganzzahl* mit Statusinformationen

- Interaktion:
  - extern mit Dateisystem.
  - mit der *grafischen Benutzerschnittstelle* zum Anzeigen von Informationen bzw. Fehlermeldungen.
  - mit der *grafischen Benutzerschnittstelle* zur Dateiauswahl.

Experimentverwaltung

- Aufgaben:
  - Koordination von Auswahl und Veränderung der aktuellen *Problemdaten*, *Kodierungsdaten* und der *Operatordaten* des aktuellen Hauptoperators. Setzen von Labels und Filtern.
  - Starten des aktuellen Hauptoperators, kontrollierter Abbruch.
- Datenverwaltung:
  - *Experimentdaten* mit
    * eindeutige Bezeichnung der aktuellen *Experimentdaten*
    * *Referenz* auf *Problemdaten* des aktuellen Problems
    * *Referenz* auf *Kodierungsdaten* der aktuellen Kodierung
    * *Referenz* auf *Operatordaten* des aktuellen Hauptoperators
  - *Referenz* auf *Dateidaten* für die *Dateiverwaltung*
- Interaktion:
  - mit der *grafischen Benutzerschnittstelle* zum Auswählen eines neuen aktuellen Experiments einschließlich Aktualisieren der *Experimentdaten*.
  - mit der *grafischen Benutzerschnittstelle* zum Auswählen eines neuen aktuellen Problems, einer neuen aktuellen Kodierung bzw. eines neuen Hauptoperators einschließlich Aktualisieren der *Referenzen* auf die damit verbundenen *Problemdaten*, *Kodierungsdaten* oder *Operatordaten*.
  - mit der *Problemverwaltung* zum Editieren der aktuellen *Problemdaten*.
  - mit der *Kodierungsverwaltung* zum Editieren der aktuellen *Kodierungsdaten*.
  - mit der *Operatorenverwaltung* zum Editieren bzw. Starten des aktuellen Hauptoperators.
  - mit der *Dateiverwaltung* zum Abspeichern der aktuellen Experimentdaten.
  - mit der *Dateiverwaltung* zum Laden abgespeicherter Experimentdaten.

Problemverwaltung

- Aufgaben:
  - Aufbau und Verwaltung eines Modells der Problemstruktur des zu bearbeitenden Optimierungsproblems. Bereitstellung dieser Struktur zur Verwendung in Operatoren und beim Aufbau der Kodierungsstruktur.
  - Speichern des aktuellen Problems.
  - Laden eines abgespeicherten Problems anhand eines vorgegebenen Bezeichners.
- Datenverwaltung:
  - *Problemdaten* mit
    - einer *Zeichenkette* zur eindeutigen Problembezeichnung
    - einer *Zeichenkette* zur Beschreibung der Fitnessfunktion
    - *Problemelementdaten* zu jedem Atom der Problemstruktur mit
      - einer *Zeichenkette* zur eindeutigen Bezeichnung
      - einer *Ganzzahl* als Kennung des repräsentierten Basisdatentyps
    - einer optionalen *Fließkommazahl* für eventuell bekanntes Optimum
  - *Referenz* auf *Dateidaten* für die *Dateiverwaltung*
- Interaktion:
  - mit der *grafischen Benutzerschnittstelle* zum interaktiven Aufbau eines Modells der Problemstruktur in der *Gruppe* von *Problemelementdaten.*
  - mit der *grafischen Benutzerschnittstelle* zum Ändern der Problembezeichnung.
  - mit der *grafischen Benutzerschnittstelle* zum Ändern des bekannten Optimums.
  - mit der *Dateiverwaltung* zum Abspeichern des aktuellen Problems so, daß eine spätere Wiederverwendung durch Wiederherstellung der *Problemdaten* gewährleistet ist.
  - mit der *Dateiverwaltung* zum Laden abgespeicherter Probleme einschließlich des Aufbaus der Problemstruktur in den *Problemdaten* und der Bereitstellung der Bezeichnung in der *Zeichenkette.*

Kodierungsverwaltung

- Aufgaben:
  - Ermöglichen des Aufbaus und der Verwaltung einer Kodierungsstruktur, analog zu einer Problemstruktur.
  - Speichern der aktuellen Kodierung.
  - Laden einer abgespeicherten Kodierung anhand eines vorgegebenen Bezeichners.
- Datenverwaltung:
  - *Kodierungsdaten* mit
    - einer *Zeichenkette* zur eindeutigen Kodierungsbezeichnung
    - *Kodierungselementdaten* für die jeweiligen *Problemelementdaten* einer analogen Problemstruktur mit
      - einer *Zeichenkette* zur eindeutigen Bezeichnung
      - einer *Ganzzahl* als Kennung des kodierten Basisdatentyps
      - einer *Ganzzahl* als Kennung der verwendeten Kodierungsart
  - *Referenz* auf *Dateidaten* für die *Dateiverwaltung*
- Interaktion:
  - mit der *grafischen Benutzerschnittstelle* zum interaktiven Aufbauen einer Kodierungsstruktur in der *Gruppe* von *Kodierungselementdaten*.
  - mit der *grafischen Benutzerschnittstelle* zum Ändern der Kodierungsbezeichnung.
  - mit der *Dateiverwaltung* zum Abspeichern der aktuellen Kodierung so, daß eine spätere Wiederverwendung durch Wiederherstellung der *Kodierungsdaten* gewährleistet ist.
  - mit der *Dateiverwaltung* zum Laden abgespeicherter Kodierungen einschließlich des Aufbaus der Kodierungsstruktur in den *Kodierungsdaten* und Bereitstellen der Bezeichnung in der *Zeichenkette*.

Operatorenverwaltung:

- Aufgaben:
  - Verwaltung und Aufbau von Operatorcharakteristiken in *Operatordaten*.
  - Speichern von Operatoren zur späteren Wiederverwendung.

  - Laden von gespeicherten Operatoren.
  - Start des Ablaufs von durch *Operatordaten* repräsentierten Operatoren.

- Datenverwaltung:
  - *Operatordaten* mit
    * einer *Zeichenkette* zur eindeutigen Operatorbezeichnung
    * einer *Ganzzahl* als Kennung für den Operatortyp
    * einer *Gruppe* von *Referenzen* auf *Operatordaten*, um eventuell direkt verwendete Operatoren zu referenzieren
    * einer optionalen *Zeichenkette*, welche den eventuell vorhandenen Operatorquelltext enthält
  - *Referenz* auf *Dateidaten* für die *Dateiverwaltung*

- Interaktion:
  - rekursiv mit der *Operatorenverwaltung* zum Starten der direkt verwendeten Operatoren mittels der *Referenzen* auf deren *Operatordaten*.
  - mit der *Dateiverwaltung* zum Speichern der *Operatordaten*.
  - mit der *Dateiverwaltung* zum Laden zuvor gespeicherter *Operatordaten*.
  - mit der *grafischen Benutzerschnittstelle* zum Editieren von Operatorquelltexten.
  - mit der *grafischen Benutzerschnittstelle* zur Ablaufkontrolle nach dem Start.
  - mit dem *Interpreter für Operatorquelltexte* zum Interpretieren von Operatorquelltexten.

Interpreter für Operatorquelltexte (LEA)

- Aufgaben:
  - Interpretieren von Operatorquelltexten.

- Datenverwaltung:
  - *Zeichenkette* mit dem Operatorquelltext
  - *Referenz* auf *Problemdaten*, um die aktuellen *Problemdaten* zum Aufbau der Repräsentation von Individuen nutzen zu können
  - *Referenz* auf *Kodierungsdaten*, um die aktuellen *Kodierungsdaten* zum Aufbauen des kodierten Teils von Individuen nutzen zu können

- Interaktion:
  - mit der *Experimentverwaltung*, um die *Referenz* auf die aktuellen *Problemdaten* zu bekommen.
  - mit der *Experimentverwaltung*, um die *Referenz* auf die aktuellen *Kodierungsdaten* zu bekommen.
  - mit der *grafischen Benutzerschnittstelle* zur Ablaufkontrolle des Interpretiervorganges.

D.3 Definition der Datenelemente

- *Basisdatentypen* sind
  - *boolsche Werte*
  - *Ganzzahlen*
  - *Fließkommazahlen*
  - *Permutationen* als *Gruppe* von *Ganzzahlen*
- *Basisdatenelemente* sind
  - *Zeichenketten*
  - *Gruppen* von gleichartigen *Basisdatentypen* oder *Basisdatenelementen*
  - *Gruppen* von *Basisdatentypen* oder *Basisdatenelementen*
  - *Referenzen* auf *Basisdatentypen* oder *Basisdatenelemente*
- *Problemelementdaten* bestehen aus
  - einer *Zeichenkette* zur Bezeichnung
  - einer *Ganzzahl* als Kennung des repräsentierten Basisdatentyps
- *Problemdaten* bestehen aus
  - einer *Zeichenkette* zur Problembezeichnung
  - einer *Gruppe* von *Problemelementdaten* für die Problemstruktur
  - einer optionalen *Fließkommazahl* für eventuell bekanntes Optimum
- *Kodierungselementdaten* bestehen aus
  - einer *Zeichenkette* zur Bezeichnung
  - einer *Ganzzahl* als Kennung des kodierten Basisdatentyps
  - einer *Ganzzahl* als Kennung der verwendeten Kodierungsart

- *Kodierungsdaten*
  - einer *Zeichenkette* zur Kodierungsbezeichnung
  - einer *Gruppe* von *Kodierungselementdaten* für die Kodierungsstruktur
- *Operatordaten* bestehen aus
  - einer *Zeichenkette* zur Operatorbezeichnung
  - einer *Ganzzahl* als Kennung für den Operatortyp
  - einer *Gruppe* von *Referenzen* auf *Operatordaten* für die verwendeten Operatoren
  - einer optionalen *Zeichenkette* für den Operatorquelltext (dieser ist bei vordefinierten Operatoren nicht erforderlich)
- *Experimentdaten* bestehen aus
  - einer *Zeichenkette* zur Bezeichnung des Experiments
  - einer *Referenz* auf *Problemdaten* für das aktuelle Problem
  - einer *Referenz* auf *Kodierungsdaten* für die aktuelle Kodierung
  - einer *Referenz* auf *Operatordaten* für den aktuellen Hauptoperator
- *Dateidaten* bestehen aus
  - einer *Zeichenkette* für einen Dateinamen
  - einer *Zeichenkette* für einen Dateipfad
  - einer *Ganzzahl* mit Statusinformationen

Anhang E

Wer war dabei?

Die Projektgruppe Genetische Algorithmen (PGA) bestand anfangs aus acht Studenten, von denen einer Ende September 1994 aus der PGA ausschied (vgl. hierzu Kapitel 9.1).

Die bis zum Abschluß der PGA tätigen Mitglieder waren:

- cand. inform. Frank Amos,
- cand. inform. Karsten Jung,
- cand. inform. Bernd Kawetzki,
- cand. inform. Wilfried Kuhn,
- cand. inform. Oliver Pertler,
- cand. inform. Ralf Reißing und
- cand. inform. Markus Schaal.

Betreut wurde die PGA von Herrn Prof. Dr. Volker Claus und Frau Dipl.–Math. Nicole Weicker.

Diese Dokumentation wurde in Zusammenarbeit aller Mitglieder verfaßt und überarbeitet. Dabei wurde die Planung und Koordination der Arbeiten an der Dokumentation sowie die Redaktion von Ralf Reißing und Nicole Weicker durchgeführt. Volker Claus unterzog die Dokumentation einer abschließenden Durchsicht.

Im folgenden wird aufgeführt, wer für welche Teile der Dokumentation hauptverantwortlich ist. Dies bedeutet, daß derjenige die erste Version erstellte und die Änderungen durchführte. Da eine Gleichbelastung aller Mitglieder angestrebt wurde, wurden

weniger belastete Mitglieder vornehmlich zur Korrektur und Verbesserung der Texte anderer Autoren herangezogen.

Frank Amos ist verantwortlich für:

- Kapitel 1: Einleitung
- Kapitel 9.2: Fazit
- Literaturverzeichnis

Karsten Jung ist verantwortlich für:

- Kapitel 6: LEA — Language for Evolutionary Algorithms
- Kapitel 7: Ein formaler Ansatz
- Kapitel 8: Kritik an EAGLE
- Anhang C: Syntax von LEA

Bernd Kawetzki ist verantwortlich für:

- Kapitel 9.1: Chronik der Projektgruppe

Wilfried Kuhn ist verantwortlich für:

- Anhang D: Grobentwurf
- Graphiken allgemein

Oliver Pertler ist verantwortlich für:

- Kapitel 5: Spezifikation des Kernsystems von EAGLE

Ralf Reißing ist verantwortlich für:

- Kapitel 2: Evolutionäre Algorithmen
- Anhang E: Wer war dabei?

Markus Schaal ist verantwortlich für:

- Kapitel 3: Einführendes zu EAGLE
- Kapitel 4: Pflichtenheft für EAGLE
- Anhang A: Glossar und Abkürzungen
- Anhang B: Bestehende Systeme

Literaturverzeichnis

In dieser Literaturliste sind nur die Stellen vermerkt, die im Text direkt zitiert werden. Eine ausführliche Literaturliste kann im Bericht der Vorbereitungsgruppe ([GJM+94]) nachgeschlagen werden.

[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.

[BBSS88] H. Blohm, T. Beer, U. Seidenberg, and H. Silber. *Produktionswirtschaft.* Neue Wirtschaftsbriefe, Herne/Berlin, 1988.

[BS93] Thomas Bäck and Hans-Paul Schwefel. An overview of evolutionary algorithms for parameter optimization. In *Evolutionary Computation*, pages 1–23. The Massachusetts Institute of Technology, 1993.

[Dar59] C. Darwin. *On the Origin of Species by Means of Natural Selection.* John Murray, London, 1859.

[Dav91] L. Davis. *Handbook of Genetic Algorithms.* Van Nostrand Reinold, New York, 1991.

[dGWH90] Claas de Groot, Diethelm Würtz, and Karl Heinz Hoffmann. Optimizing complex problems by nature's algorithms: Simulated Annealing and Evolution Strategy - a comparative study. In *Parallel Problem Solving from Nature, 1st Workshop, PPSN I*, pages 445–454. Springer–Verlag, 1990.

[DJ75] K. De Jong. *An Analysis of the Behaviour of a Class of Genetic Adaptive Systems.* Doctoral thesis, University of Michigan, Ann Arbor, 1975.

[DS90] Gunter Dueck and Tobias Scheuer. Threshold acceptance: A general purpose optimization algorithm appearing superior to simulated annealing. *Journal of Computational Physics*, (90):161–175, 1990.

[Due93] Gunter Dueck. New optimization heuristics for the Great Deluge Algorithm and the Record–to–Record Travel. In *Journal of Computational Physics*, volume 104, pages 86–92, 1993.

[Fog92] D. B. Fogel. An analysis of evolutionary programming. In D.B. Fogel and J. W. Atmar, editors, *Proceedings of the first annual conference on evolutionary programming*, La Jolla, 1992. Evolutionary Programming Society.

[FOW66] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution.* Wiley, New York, 1966.

[GH91] M. Grötschel and O. Holland. Solution of large-scale symmetric travelling salesman problems. *Mathematical Programming*, 51:141–202, 1991.

[GJM+94] Elke Göckler, Karsten Jung, Jochen Meßner, Juan Roldan Güpner, Heike Weiss, and Jörg Zedelmayr. Dokumentation zur Vorbereitung der Projektgruppe Genetische Algorithmen. Technical report, Universität Stuttgart, Fakultät Informatik, Institut für Informatik, Abteilung Formale Konzepte, 1994.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison Wesley, Reading, 1989.

[Gre86] John J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions On Systems, Man, and Cybernetics*, SMC-16(1):122–128, 1986.

[HB90] Frank Hoffmeister and Thomas Bäck. Genetic Algorithms and Evolution Strategies: Similarities and differences. In *Parallel Problem Solving from Nature, 1st Workshop, PPSN I*, pages 455–469. Springer–Verlag, 1990.

[HB94] J. Heitkötter and D. Beasley. The hitchhiker's guide to evolutionary computation: A list of frequently asked questions (faq). USENET: comp.ai.genetics, erhältlich über anonymous FTP mit: rtfm.mit.edu:/pub/usenet/news.answers/ai-faq/genetic, 1994.

[Hol75] J. H. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, 1975.

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

[Koz92] John R. Koza. *Genetic Programming.* The MIT Press, 1992.

[Ott94] T. Otto. Reiselust. *c't*, (1):188–193, 1994.

[Ous94] John K. Ousterhout. *Tcl and the Toolkit.* Addison-Wesley professional computing series. Addison-Wesley, Reading, Mass. [u.a.], 1994.

[Rec73] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Frommann-Holzbog, Stuttgart, 1973.

[SB92] Hans-Paul Schwefel and Thomas Bäck. Künstliche Evolution — eine intelligente Problemlösungsstrategie? In *KI–Zeitschrift*, pages 1–20, 1992.

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

[Sed91] Robert Sedgewick. *Algorithmen.* Addison Wesley, Bonn, 1991.

[SHF94] E. Schöneburg, F. Heinzmann, and S. Feddersen. *Genetische Algorithmen und Evolutionsstrategien.* Addison Wesley, Bonn, 1994.

[SWE93] J. D. Schaffer, D. Whitley, and L. J. Eshelman. Combinations of genetic algoritms and neural networks: a survey of the state of art. In *COGANN-92, International Workshop on Combinations of Genetic Algorithms and Neural Networks*, 1993.

[Sys89] Gilbert Syswerda. Uniform crossover in genetic algorithms. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, San Matteo, 1989. ICGA89.

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