Universität Potsdam
Naturwissenschaftliche Fakultät
Informatik

Projekt

Bildverarbeitung mit Hilfe von genetischen Algorithmen

Betti Österholz

oesterholz@fib-development.org

2. Mai 2004

Copyright (C) 2004 Betti Österholz

This document is free; you can redistribute it and / or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2, June 1991, of the License, or (at your option) any later version.

This document is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this document; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Copyright (C) 2004 Betti Österholz

Dieses Dokument ist frei. Sie können es unter den Bedingungen der GNU General Public License, wie von der Free Software Foundation veröffentlicht, weitergeben und/oder modifizieren, entweder gemäß Version 2, June 1991, der Lizenz oder (nach Ihrer Option) jeder späteren Version.

Die Veröffentlichung dieses Dokument erfolgt in der Hoffnung, daß es Ihnen von Nutzen sein wird, aber OHNE IRGENDEINE GARANTIE, sogar ohne die implizite Garantie der MARKTREIFE oder der VERWENDBARKEIT FÜR EINEN BESTIMMTEN ZWECK. Details finden Sie in der GNU General Public License.

Sie sollten eine Kopie der GNU General Public License zusammen mit diesem Dokument erhalten haben. Falls nicht, schreiben Sie an die Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.

Inhaltsverzeichnis

I Aufgabenstellung **1**

1 Aufgabenstellung: Bildverarbeitung mit genetischen Algorithmen **2**
1.1 Die Idee 2
1.2 Problem 2
1.3 Lösungsidee 2
1.4 Hilfestellung zur Einteilung des Projektes 3
1.4.1 Einführung in die Grundlagen 3
1.4.2 Entwurf der Bildbeschreibungssprache und der genetischen Operationen 3
1.4.3 Implementierung und Testen der Idee 4
1.4.4 Beleuchtung der Hintergründe und Auswertung des Ergebnisses 4

II Einführung in die Grundlagen **5**

2 Einleitung **6**

3 Grundlagen von genetischen Algorithmen **8**
3.1 Geschichte 8
3.2 Evolution 9
3.3 Grundlagen evolutionäre Algorithmen (EA) 12
3.4 Unterarten 12
3.4.1 Genetische Algorithmen (GA) 14
3.4.2 Klassische genetische Algorithmen (GA) 14
3.5 Anmerkungen zu EA 15

4 Bilder **17**
4.1 Einleitung 17
4.2 Bildformat 17
4.3 Eigenschaften von Bildern 18
4.4 Codierungen in den Darstellungsarten Vektorgrafik oder Bitmapgrafik 19
4.4.1 Vektorgrafik 19

4.4.2 Bitmapgrafik 20
4.5 Farbmodell/Farbschema 21
4.5.1 S/W (schwarz/weiß) 21
4.5.2 Grayscale (Halbton) 21
4.5.3 RGB 21
4.5.4 Induzierte Farben 21
4.6 Komprimierung 22
4.6.1 Verlustfreie Komprimierung 22
4.6.2 Verlustbehaftete Komprimierung 22

5 Anmerkungen zur Aufgabenstellung 24

III Entwurf der Bildbeschreibungssprache und der genetischen Operationen 25

6 Anforderungen 26

7 Warum sich genetische Algorithmen zur Bildcodierung anbieten 27

8 Die Bildsprache 29
8.1 Elemente 29
8.1.1 Vektoren 29
8.1.2 Punkte 30
8.1.3 conc 30
8.1.4 Bereichsobjekt 30
8.1.5 Funktionen 31
8.1.6 Sinusfunktionen (geplant) 32
8.1.7 Hintergrundfarbe (herausgenommen) 32
8.2 Definitionen für fib 32
8.2.1 Definition korrektes/vollständiges fib-Objekt 32
8.2.2 Definition von ”unterhalb“ und ”oberhalb“ in einem fib-Objekt 33
8.2.3 Definition Teilobjekt 33
8.2.4 Definition fib-Bild 34
8.2.5 Definition korrektes/vollständiges fib-Bild 34
8.3 Theoretische Aussagen zu fib 34
8.3.1 Warum können mit dieser Sprache alle möglichen Bilder dargestellt werden? 35
8.3.2 Mächtigkeit von fib 37
8.3.3 Warum kann jedes mit den fib-Elementen gebildete Objekt als Bild dargestellt werden? 38

9 Die genetischen Operationen auf fib **40**
9.1 Die Fitness eines Individuums 40
9.2 Selektion 41
9.2.1 Ein Programm Löschen 41
9.3 Vermehrung 41
9.3.1 Erzeugung eines Individuums 41
9.4 Ein Teilobjekt einfügen (Genimport) 41
9.5 Mutation 42
9.5.1 Löschung eines Teilobjektes 42
9.5.2 Ändern eines Wertes 42
9.5.3 Änderung einer Variablen 42
9.5.4 Einfügen einer Variablen 42
9.5.5 Löschung eines Terms 43
9.5.6 Hinzufügen eines Vektors zu einer Liste 43
9.5.7 Löschen eines Vektors aus einer Liste 43
9.5.8 Verschieben eines Elements 44
9.5.9 Vertauschen der Objekte in conc Objekten 44

10 Komplexitätsabschätzung **45**
10.1 Abschätzung 45
10.1.1 Annahmen 45
10.1.2 Die Sammelmaschine 47
10.1.3 Würfel Annahme 47
10.2 Rechnung 48
10.3 Ergebnis 50

11 Weitere Annahmen zu fib **51**
11.1 Komprimierungsmöglichkeiten vom fib 51
11.2 Überdeckung von fib-Objekten und Dichte der fib-Objekte im Hypothesenraum 51
11.3 Annahme über verdeckte Objekte 52

12 Parallelen zur natürlichen Evolution **54**

IV Implementierung der Idee **55**

13 Einleitung für die Implementierung **56**

14 Entwurf des Programmsystems **57**
14.1 Entwurf der Benutzerschnittstelle 57

15 Realisierung des Programmsystems **58**
15.1 Namenskonventionen 58
15.2 Realisierung der Bildbeschreibungssprache fib 59
15.2.1 Ordnungen 59
Ordnung der Objektpunkte O_O 60
Ordnung der Verschiebepunkte 61
Ordnung der Vertauschpunkte für conc 62
Ordnung der Teilobjektpunkte 63
Ordnung der Punktteilobjektpunkte 63
Ordnung der einzelnen Werte 63
Ordnung der einzelnen Variablen 64
Ordnung der Listenvektoren 64
Ordnung der Funktionsobjekte und Bereichsobjekte 64
15.2.2 Benötigte Klassen 64
15.2.3 Die Klassen im einzelnen 65
Basisklasse GraphicObject 65
Die Vektorklassen Vector, Color, Position, UnderFunction und UnderArea 65
Die Klassen Point, Function, Area, Conc, ListObject, PicturObject 65
Vererbungsgraph der fib-Klassen 66
15.3 Benötigte Methoden der Bildbeschreibungssprache fib 66
15.3.1 Gemeinsame Methoden aller Klassen 67
Konstruktoren 67
Methoden zur Bestimmung des Wertebereichs der Ordnungen 67
Andere Abfragemethoden 68
Verändernde Methoden 70
Allgemeine Methoden 74
15.3.2 Zusätzliche Methoden der Klassen Point, Function, Area, Conc, ListObject und PictureObject 79
Methoden zur Bestimmung des Wertebereichs der Ordnungen 79
Andere Abfragemethoden 81
Verändernde Methoden 89
Allgemeine Methoden 104
15.3.3 Zusätzliche Methoden der Vector Klasse 108
Abfragemethoden 108
Verändernde Methoden 108
15.3.4 Zusätzliche Methoden der Color Vektorklassen 109
Allgemeine Methoden 109
15.3.5 Zusätzliche Methoden der Klasse Point 109
Abfragemethoden 109
15.3.6 Zusätzliche Methoden der Klasse ListObject 109

Abfragemethoden . . . 109
15.3.7 Realisierung der genetischen Operatoren mit der Klasse Environment . . . 110
Konstruktoren . . . 110
Methoden . . . 110
15.3.8 Realisierung der Klasse Individual für einzelnen Individuen 115
Konstruktoren . . . 115
15.3.9 Realisierung der Klasse Parameter für die Parameter des genetischen Algorithmus . . . 115
Konstruktoren . . . 116
Bedeutung von Parametern in den Arrays . . . 116
15.3.10 Realisierung der Klasse PicturMatrix für die Bildmatrix . 118
Konstruktoren . . . 118
Methoden . . . 118

16 Anmerkung zur Realisierung 123

17 Testen des Programmsystems 124
17.1 Parameter . . . 125
17.2 Die PicturMatrix Klasse . . . 125
17.3 Klassen für die fib-Objekte . . . 125
17.3.1 Die Vector Klassen . . . 125
17.3.2 Die Klasse Point . . . 126
17.3.3 Die Klasse Conc . . . 126
17.3.4 Die ListObjekt Klassen . . . 126
17.4 Die Klasse Environment . . . 126
17.5 Die Klasse Individual . . . 127
17.6 Testen der `moveElement()` Methode . . . 127
17.7 Bewertung des Tests des Programmsystems . . . 127

18 Bewertung der Realisierung 128

19 Aussicht 129

V Auswertung 131

20 Aufwand zur Bildcodierung 132
20.1 Aufwand der Bildcodierung hängt von der Komplexität des Bildes ab 132

21 Komprimierung 136

22 Verständlichkeit 137

23 Operatoren 139

24 Parameter **140**

25 Bezugnahme auf die Aufgabenstellung **141**

26 Bezugnahme auf die Abschätzung aus Kapitel 10 **142**

27 Weitere Ergebnisse **144**
27.1 Kombination von Bereichs- und Funktionsobjekten 144
27.2 Init mit korrekten Bildern 145
27.3 Armageddon - Neustart zur Widerbelebung 145
27.4 Unsterbliche Individuen 146
27.5 Wachsende Individuen 146

28 Zusammenfassung **148**

29 Ausblick/Bewertung **149**

30 Mithelfer **150**

31 Eidesstattliche Erklärung **151**

Teil I

Aufgabenstellung

Kapitel 1

Aufgabenstellung: Bildverarbeitung mit genetischen Algorithmen

1.1 Die Idee

Ein (natürliches) Bild besteht im allgemeinen nicht aus einer Aneinanderreihung zusammenhangloser Pixel (Punkte mit einer bestimmten Farbe), sondern zeigt z.B. Objekte die gegeneinander abgegrenzt sind und eine Textur haben. Solche Objekte (z.B. Kreise, Linien aber auch Komplexere) wiederholen sich oftmals auf Bildern, sei es als selbstähnliche Kopie oder transformierte Kopie.

1.2 Problem

Wie kann ich solche Informationen aus einem Bild extrahieren, das in Form einer Matrix von Pixeln gegeben ist? Diese Information kann dann z.B. zur Komprimierung benutzt werden, da die Informationen, die benötigt werden um ein optimales Objekt zu kodieren, maximal seiner Pixelmatrix entspricht (diese Codierung ist ja gegeben). Man kann mit dieser Information aber auch Objekte identifizieren und auf anderen (Teil-) Bildern wiedererkennen. Oder die Informationen können genutzt werden, um das Bild zu manipulieren, z.B. Objekte als Ganzes zu entfernen, kopieren oder einzufügen.

1.3 Lösungsidee

Es wird eine ”Bildsprache“ definiert, in der Objekte, Transformationen und Zusammenhänge zwischen diesen definieren werden können. Mit Hilfe dieser entwirft eine KI dann Programme (in der Bildsprache), testet und verändert diese, bis sie ein Bild darstellen, das dem zu kodierendem Bild ähnlich genug ist (was

dies ist, sollte spezifiziert werden). Dabei können an dieses Programm noch weitere Einschränkungen gestellt werden, z.B. Kürze und Abarbeitungsgeschwindigkeit des Programms. Mit diesen Kriterien kann dann feststellen werden, wie "gut" das Programm ist.

Unter anderem bieten sich dafür genetische Algorithmen/Programmierung an. Da die einzelnen Parameter und Konstrukte der Sprache durch Mutation erzeugt und angepasst werden können und die "besten" Programme weiterentwickelt werden. Mit "crossing over", Austausch von Programmcode zwischen Programmen oder Programmstellen, kann der Umstand berücksichtigt werden, dass die Objekte an anderen Stellen eventuell nur verändert (z.B. vergrößert) wiederverwendet werden.

Warum genau sich genetische Algorithmen/Programmierung anbieten, soll noch im Einzelnen erläutert werden. Es soll auch eine Verbindung zur "biologischen" Evolution/Genetik hergestellt werden. Ein weiterer Bestandteil der Idee ist es, dass nicht nur einem "Bildprogramm/-objekt" als Ganzes eine Fitness zugeordnet wird, sondern auch seinen Teilobjekten, indem bestimmt wird, welcher Teil von diesem überdeckt wird und wie gut dieser Teil überdeckt wird. Die Objekte mit der höchsten Fitness erhalten höhere Chancen beim "Überleben" (werden seltener gelöscht) und "Vermehren" (werden häufiger für neue Objekte verwendet).

Es ist weiterhin möglich bei der Codierung neuer Bilder Objekte zu verwenden, die schon in anderen Bildern eine hohe Fitness erreichten. (als Vorwissen)

1.4 Hilfestellung zur Einteilung des Projektes

1.4.1 Einführung in die Grundlagen

Einführung in die Grundlagen genetischer Algorithmen und der Bilddarstellung.

1.4.2 Entwurf der Bildbeschreibungssprache und der genetischen Operationen

- Entwurf der Bildbeschreibungssprache:
  - Spezifizieren der einzelnen Bauteile und deren mögliche Beziehungen zueinander
  - Begründung warum mit dieser alle möglichen Bilder realisiert werden können
  - den Entwurf im Hinblick auf die Aufgabenstellung beleuchten
  - Zu berücksichtigende Fragestellung dabei sind:
    - Was ist zu beachten?
    - Wie sind die Anforderungen an die Bildbeschreibungssprache?
    - Welche Erweiterungen und Parameter sind möglich?

- Die genetischen Operationen spezifizieren:
  - die genetischen Operationen, die auf dieser Bildbeschreibungssprache im genetischen Algorithmus angewendet werden können, beschreiben
  - mögliche Parameter dieser beschreiben/festlegen
- Abschätzung der Komplexität der Codierung von Bildern:
  - Um die Realisierbarkeit und die Komplexität der Bilder, die in vernünftiger Zeit codiert werden können, abzuschätzen, wird eine einfache Abschätzung gemacht.
  - Eine Begründung, warum die Abschätzung realistisch ist, sollte angegeben werden.

1.4.3 Implementierung und Testen der Idee

Implementierung der Idee: Für die spezifizierte Sprache und Operationen ist ein Programmsystem zu entwickeln, dass diese realisiert. Mögliche Parameter und Einstellungen der Operatoren sind zu berücksichtigen. Für die statistische Auswertung des Ergebnisses eines Durchlaufes, sind relevante Daten dieses Durchlaufes persistent zu halten. Statistische Funktionen sind zu realisieren, z.B. Anzahl der Mutationen, Zeit/"beste Fitness“ Verlauf darstellen.

Testen der Idee: Es sind verschieden Durchläufe mit verschiedenen Einstellungen durchzuführen und auszuwerten. In wie weit werden Vermutungen bestätigt?

1.4.4 Beleuchtung der Hintergründe und Auswertung des Ergebnisses

Die Hintergründe der Idee sind Theoretisch zu beleuchten. Abschätzungen der Komplexität und des Zeitaufwands sind zu erörtern. Was sind Vorteile und Nachteile der Idee. Untersuchung der gewonnenen Bildprogramme auf Strukturen. Kann man Objekte des Bildes im Bildprogramm wiederfinden? (Mögliche) Parallelen zur natürlichen Evolution aufzeigen. Da alle Aspekte auszuwerten zu umfangreich wären, soll nur auf ausgewählte Aspekte eingegangen werden.

Es wäre eventuell besser die Abschätzung des Aufwands zur Bildkodierung vor der Implementierung der Idee anzufangen (oder Beide von verschiedenen Personen realisieren zu lassen), aus zeittechnischen Gründen halte ich es für besser, dies aber parallel zu tun.

Teil II

Einführung in die Grundlagen

Kapitel 2

Einleitung

Die nachfolgende Arbeit entstand aus meinem Wunsch, eine eigene wissenschaftliche Abhandlung zu verfassen.

Die Idee, etwas über evolutionäre Algorithmen (EA) zu erarbeiten, hat seinen Ursprung darin, dass ich mich für Verfahren interessiere, die zur automatischen Problemlösung dienen, und darin, dass auf dem Gebiet der evolutionären Algorithmen, gerade im Bereich der komplexeren Anwendungen, anscheinend noch wenig Forschung betrieben wurde. Ich suchte also ein Thema, das sich von mir mit evolutionären Algorithmen bearbeiten ließ, das hieß vor allem, dass genug ”Lernmaterial“ für den Algorithmus zugänglich ist, und eventuell noch praktischen Nutzen bringen konnte. So verfiel ich auf die Bildbearbeitung mit evolutionären Algorithmen. Die Bildbearbeitung ist ein großer Sektor in der EDV und der Grundstoff - Bilder - ist leicht verfügbar.

Die Idee ist: Aus speicheraufwändigen Bitmap Bildern mit evolutionären Algorithmen speichersparende Vectorbilder zu generieren.

In meinen Recherchen habe ich über solche Verfahren nichts gefunden, es ist also gut möglich das ich ”Neuland betrete“.

Die nachfolgende Dokumentation ist dazu da, die Ideen hinter dem Projekt darzustellen und anderen die Möglichkeit zu geben, mein Projekt für weitere Forschungen auf diesem Gebiet nutzen zu können. Da die Aufgabenstellung ohne genügend Vorwissen entstanden ist, behalte ich mir vor, von dieser, wenn es vorteilhaft erscheint oder nicht anders möglich ist, abzuweichen. Dies betrifft vor allem die Aufteilung der Themen zu den einzelnen Punkten, die, der besseren Übersichtlichkeit wegen, eventuell nicht eingehalten wird.

Außerdem können Ergebnisse nicht im Voraus bekannt sein. Deshalb können auch nur Ergebnisse, die ich wirklich erhalte, ausgewertet werden und eventuell Vermutungen angestellt werden, warum andere, in der Aufgabenstellung erwähnte Dinge, nicht eingetroffen sind.

Die Einteilung des Projektes in Teile erfolgt nach folgendem Schema:

Teil II soll die Grundlagen für das Projekt bereitstellen

Teil III dient zur theoretischen Ausarbeitung der Bildbeschreibungssprache und des Algorithmus

Teil IV widmet sich der Implementierung des Projektes

Teil V dient der Zusammenfassung der Ergebnisse

Evolution ist eines der grundlegensten Naturgesetze unseres Universums. Seit rund 3 Milliarden Jahren gibt es auf der Erde eine genetische Evolution, die, wie wohl nur wenige bestreiten werden, ganz erstaunliche Ergebnisse und Problemlösungen hervorgebracht hat. Deshalb ist es nicht verwunderlich, dass versucht wird, die Methoden und/oder die Wirkungsweise der genetischen Evolution auch in der Informationstechnik einzusetzen. Vielfach werden schon große Fortschritte in der Wissenschaft und Forschung erzielt, indem man sich Prinzipien der Ergebnisse (z.B. Flügelauftrieb) der natürlichen Evolution von der Natur "abgeschaut" hat. Aber auch viele Interpretationen der Evolution wurden schon, teilweise mit Erfolg, versucht anzuwenden.

Die nachfolgende Arbeit ist einer dieser Versuche.

Kapitel 3

Grundlagen von genetischen Algorithmen

Genetische Evolution gibt es auf diesem Planeten wahrscheinlich schon mindestens drei Milliarden Jahren.

Wie funktioniert Evolution aber oder was ist sie überhaupt genau?

Ob ich diese Fragen zufriedenstellend beantworten kann, weiss ich nicht. Ich will es aber versuchen und vielleicht noch ein paar Denkanstöße geben. Die nachfolgende Sicht von Evolution und genetischer Evolution ist grundlegend für die ganze nachfolgende Arbeit. Sie nimmt aber keinesfalls in Anspruch vollständig oder "richtig" zu sein. Um genetische Evolution verstehen zu können, muss wahrscheinlich zuerst erklären werden was Evolution ist.

3.1 Geschichte

Der erste Forscher der auf die Evolution aufmerksam wurde und darüber eine Buch verfasste war Charles Darwin (1809 - 1882), der 1859 das Buch "Die Entstehung der Arten" veröffentlichte. Dabei ging es um die Evolution in der Natur und wie Arten aus anderen Arten entstanden sind. Eine Idee die ihm besonders im Kreise der damals noch wesentlich mächtigeren Kirche viel Unsympathien einbrachte, da diese Idee dem Schöpfungsgedanken zuwiderlief.

Die Idee, Evolution auch zum maschinellen Lernen zu verwenden, war sehr früh in der Informatik vorhanden. Schon 1932 beschrieb W. D. Cannon natürliche Evolution als einen Lernprozess. A. M. Turing (1950) vermutete "an obvious connection between [machine learning] and evolution." (Übersetzung: "eine offensichtliche Verbindung zwischen [maschinellem Lernen] und Evolution"). Leider war zu dieser Zeit die Rechnerleistung bei weitem noch nicht hoch genug um eine Simulation der Evolution im vernünftigem Rahmen zu gewährleisten.

Erst in den 1980'er war die zur Verfügung stehende Rechenleistung hoch genug, um Evolution zum maschinellen Lernen sinnvoll einsetzen zu können. Danach wurde das öffentliche Interesse an Evolution in der Informatik größer. Evolution

wurde schon in vielen Bereichen mehr oder weniger erfolgreich eingesetzt. Angefangen vom Finden globaler/lokaler Optima von Funktionen, bis hin zum automatischem Schaltungsentwurf oder Roboterkonstruktion.

3.2 Evolution

Evolution ist ein allgemeines Prinzip/Naturgesetz das man kurz mit: "Das Bessere setzt sich durch." erklären kann. Bei genauerer Betrachtung wird der Spruch etwas undeutlich, denn was ist "besser" (für wen) und was ist "sich" oder "durchsetzen".

"Sich" kann auf alles bezogen werden. Ob es nun ein Lebewesen, ein Programm, ein Prinzip oder Sonstiges ist. Für "sich" sind aber die Informationen die es enthält (z.B. der Aufbau der Moleküle oder Art des Gemeinschaftslebens) und nicht seine Materie oder Energie (wenn vorhanden) für die Evolution ausschlaggebend. Ein Hund ist nicht so erfolgreich, weil er 25 kg Materie enthält, sondern weil diese Materie (und seine Energie, wenn diese von Materie getrennt wird) auf ganz spezielle Weise "verteilt" ist. Auch darf "sich" nicht als ein abgeschlossenes Einzelnes gesehen werden, sondern muss die Umwelt mit einbeziehen. Ameisen sind z.B. so erfolgreich, weil sie in Staaten leben. "Sich" kann auch weitere "sich's" enthalten oder in einem "sich" enthalten sein. So haben viele Säugetiere Darmbakterien, ohne die sie nicht mehr die gleichen Tiere wären, viele Arten bestehen nur aufgrund dieser Darmbakterien, z.B. Kühe.

Die Frage "für wen besser? " muss mit: "für sich besser" beantwortet werden. "Für sich besser" ist, wenn "sich" häufiger oder wahrscheinlicher wird.

Damit ist auch gleich die Frage beantwortet, was "durchsetzen" ist: Etwas setzt "sich" durch, wenn es wahrscheinlicher/häufiger in einer Gruppe/Menge von Dingen/anderen "sich's" wird. Die in Australien eingeführten Kaninchen, haben sich in den letzten Jahrhunderten gegenüber andern Tierarten durchgesetzt. Oder innerhalb dieser Tierartengruppe (Grasfresser) sind sie, auf Kosten der Anderen, wahrscheinlicher/häufiger geworden. Damit etwas wahrscheinlicher/häufiger werden kann, ist natürlich Zeit (oder Ähnliches) von Nöten. Das heißt, Evolution ist ein zeitlicher Prozess.

Was bei der Erklärung: "Das Bessere setzt sich durch." nicht direkt über die Evolution gesagt wird, ist, dass das "Sich" in mehreren ähnlichen Kopien vorhanden sein muss, damit es "sich" gegen andere "sich's" durchsetzen kann (damit sich ein Grasfresser gegenüber anderen durchsetzen kann, muss es davon mehrere geben). Daraus folgt unter anderem; um an einer Evolution teilnehmen zu können, müssen von "sich's" ähnliche Kopien erstellt werden (z.B. durch Vermehrung) und auch mehrere dieser Kopien gleichzeitig vorhanden sein (für Selektion). Nur so kann sich davon "das Bessere" "durchsetzen" und es kann immer wieder "Besseres" entstehen. Zusammengefasst kann gesagt werden, dass Evolution besagt: "Dinge", die so sind, das sie sich selbst in einer Umwelt wahrscheinlicher machen, werden wahrscheinlicher. Insofern ist Evolution trivial.

Das "sich" wird als Individuum bezeichnet.

Wie gut ein Individuum ist, wird als Fitness bezeichnet.

Selektion ist die Auswahl von Individuen, so dass die besseren Individuen wahrscheinlicher/mehr werden.

Genetische Evolution bezieht sich auf eine Evolution in der die Gene von Lebewesen (Individuen) die Informationsträger darstellen (Genotyp). Alle Gene eines Individuums zusammen stellen dann den Genotyp eines Individuums dar. Diese werden als ein Lebewesen "dekodiert" bzw. führen dann zur Ausprägung des Phänotyps des Individuums in der Umwelt (die Umwelt kann Auswirkungen auf den Phänotypen und dessen Ausbildung haben). Der Phänotyp wiederum bestimmt zusammen mit der Umwelt (ein Fisch an Land hat keine hohe Fitness im Verhältnis zu Landlebewesen) die Fitness (eigentlich nur ein theoretisches Maß) des Individuums. Der Phänotyp zusammen mit der Umwelt hat Auswirkungen darauf, wie wahrscheinlich das Individuum Nachkommen zeugen wird oder wie viele Individuen ein zumindest ähnliches Genom in Zukunft erhalten und damit, wie wahrscheinlich dieses oder ähnliche Genome in Zukunft sind, so wird der Kreis geschlossen.

Am Beispiel der Evolution in der Natur:

Man nehme z.B. den Schneehasen. Besonders gut geeignet für die Evolution, weil er sich so stark vermehrt. Den Hasen, das kleine, kuschelige, weiße, hoppelnde Ding, wird Phänotyp genannt, also das Phänomen in der Natur. Die Informationen die in seinen Genen, den spiralförmigen Dingern im Zellkern und Gene anderswo (z.B. in Mitochondrien), gespeichert sind, wird Genotyp genannt. Genotyp und Phänotyp zusammen wird als Individuum bezeichnet, also der Hase als ganzes. Mehrere Individuen, die untereinander in Beziehung stehen und sich untereinander fortpflanzen können, werden als Population bezeichnet. Die Gesamtheit der Genotypen einer Population wird als Genpool der Population charakterisiert. Weil die Evolution auf die Gene bzw. den Genpool wirkt, wird sie als genetische Evolution bezeichnet.

Der Genotyp bestimmt stark den Phänotyp, das Aussehen des Hasen, legt den Phänotyp aber nicht allein fest, denn die Umwelt kann als weiterer Faktor den Phänotyp bestimmen. Wenn ein Hase nur wenig zu fressen bekommt, wird er nicht dick werden, ganz egal was in seinen Genen steht, wo nichts ist kann auch nichts werden. Der Genpool, also die Gesamtheit der Genotypen, ist das, was die Evolution in einer Population verändert oder anpasst. Wird der Phänotyp durch die Umwelt geändert, wird das meist nicht weitergegeben. Wenn ein Hase dünn ist, weil er wenig zu fressen bekommt, können seine Kinder trotzdem dick werden.

Es ist allerdings möglich, das Mütter erworbene Immunitäten an ihre Kinder weitergeben, z.B. mit der Muttermilch. Also, etwas was eigentlich zum Phänotyp gehört und nicht zum Genotyp, da diese Immunität nicht in den Genen kodiert ist. Hier sieht man noch einen wichtigen Punkt der Natur: Die Ausnahme ist meist Regel.

Die Veränderung des Genpools geschieht allerdings über den Umweg der Phänotypen der Population. Unter vielen Komponenten, ist die Fortpflanzung dabei die

Wichtigste. Um so besser der Phänotyp dazu geeignet ist, dass das Individuum sich fortpflanzt, um so eher wird es viele Nachkommen mit seinen Genen haben und um so wahrscheinlicher werden seine Gene in der zukünftigen Population anzutreffen sein.

Ein flinker Hase wird wahrscheinlicher ins geschlechtsreife Alter kommen, da er Jägern besser entkommen kann, als ein nicht so flinker Hase. Dadurch haben die flinken Hasen eine bessere Chance sich fortzupflanzen und viele Nachkommen zu haben, als Langsamere. Flinke Hasen haben auch eher Gene die sie flink machen als Langsamere und dadurch werden Gene die flink machen eher vererbt, weshalb es ziemlich schwierig ist Hasen mit der Hand zu fangen. (weiß ich aus eigener Erfahrung) Hier wird ein weiterer Punkt der Evolution sichtbar, es geht immer um Wahrscheinlichkeiten und nicht um ”mit Sicherheit“. Hasen die flinker als andere sind, können auch einfach das Glück gehabt haben, als sie klein gewesen sind, dort gelebt zu haben, wo es immer gutes Futter gab, das sie groß und flink machte, aber nicht ”flinkmachendere Gene“ als nicht so schnelle Hasen haben. Ein Hase der noch so gute flinkmachende Gene hat, kann trotzdem noch als kleiner Hase einem Unglück oder Räuber zum Opfer fallen. Es ist nur wahrscheinlicher das ein Hase mit ”flinkmachenden Genen“ flink wird, länger lebt und mehr Nachkommen zeugt. Das ist auch ein Grund warum Evolution nur auf vielen Individuen wirkt und nicht auf ein Einzelnes alleine.

Hasen die eine bessere Chance sich fortzupflanzen und viele Nachkommen zu zeugen haben als andere, haben eine höhere Fitness als andere. Das heißt um so fitter Hasen sind, um so wahrscheinlicher werden ihre Gene in späteren Generationen aufzufinden sein.

Ein weiterer Punkt ist, dass nichts so einfach oder abgeschlossen ist, wie es zunächst scheint. Trotz der Tatsache das ein Hase ein Individuum ist, kann er wiederum andere Individuen, sogar ganze Populationen, enthalten. Hasen haben Darmbakterien, die als eigenständige Individuen in einer Umwelt (Hasendarm) angesehen werden. Der Hase ohne seine Darmbakterien ist allerdings nicht mehr das gleiche Individuum aus evolutionstechnischer Sicht, da ohne seine Darmbakterien seine Fitness anders ist. In diesem Fall ist sie wahrscheinlich schlechter, weil er seine Nahrung schlechter verdauen kann. Diese Darmbakterien haben ihre eigenen Gene, bzw. Genotyp, und ihren eigenen Phänotyp. Als Population haben sie ihre eigene genetische Evolution. Auch können Gruppen von Hasen zu Individuen zusammengefasst werden, die untereinander in evolutionärer Beziehung stehen. Bei solchen ”Gruppenindividuen“ zählt dann nicht mehr das Überleben des einzelnen Hasen aus der Gruppe, sondern nur noch das der Gruppe, bzw. dessen Genpool. So können Gene die dazu führen das sich einzelne Hasen nicht fortpflanzt (z.B. Gene für einen Beschützerinstinkt, der stärker ist als der Überlebensinstinkt) durchaus sinnvoll sein und erhalten bleiben, wenn sie die Fitness der Gruppe erhöhen.

Evolution wirkt auch bei vielen Vorgängen, in denen sie gar nicht vermutet wird, z.B. hat man bei Ideen oder Programmen meist mehrere von diesen zu einem Thema und mit der Zeit setzen sich dann die ”Bessern“ (was besser ist, bestimmt die

Umwelt) von diesen durch und andere nicht so gute verschwinden.

Die gewollte und bewusste Anwendung in der Informatik von Wissen über die Evolution, heißt evolutionäre Algorithmen, die der genetischen Evolution entsprechend genetische Algorithmen. Deren Grundlagen seien im Nachfolgendem beschrieben.

3.3 Grundlagen evolutionäre Algorithmen (EA)

Evolutionäre Algorithmen arbeiten auf einer Menge (Population) von potentiellen Lösungen (Individuen), die das gegebene Problem mehr oder weniger gut lösen, bzw. der optimalen Lösung mehr oder weniger nahe kommen. Von diesen werden dann Neue geschaffen, indem einige bis alle zufällig verändert werden. Von allen (potentiellen) Lösungen, Neue und Alte, werden dann, in einem Selektionsschritt, einige gelöscht. Dabei haben weniger gute Lösungen eine höhere Wahrscheinlichkeit gelöscht zu werden. Dies führt im Allgemeinen dazu, dass die guten Lösungen eine höhere "Überlebenschance" (Fitness), also eine höhere Wahrscheinlichkeit, nicht gelöscht zu werden, haben und damit in der Menge bessere Lösungen zunehmen. Diesen Prozess der Veränderung und Selektion wird dann solange wiederholt, bis eine Terminalkondition (Endbedingung) zutrifft. Im Allgemeinen eine bestimmte Anzahl von Durchläufen oder/und eine Lösung überschreitet eine festgelegte Grenze, in wie gut sie das Problem löst.

Listing 3.1: Allgemeiner Algorithmus

```
1  initialisieren der Menge der potentiellen Lösungen
2  loop
3     - hinzufügen neuer Individuen zur Menge, die aus alten
4     Individuen, aus der Menge, mit Zufall generiert werden
5     - Selektion = zufälliges Löschen von Individuen aus der
6     Menge, bessere haben eine höhere Wahrscheinlichkeit
7     nicht gelöscht zu werden
8  until (Terminalkondition erfüllt)
```

Im Listing 3.1 ist schematisch der allgemeine Algorithmus eines evolutionären Algorithmus in pseudo-C dargestellt. Der gleiche Algorithmus wird im Bild 3.1 grafisch dargestellt.

Bei der Selektion von Individuen findet oft die weel-selection Anwendung, dargestellt in Bild 3.2 (übernommen aus [3]). Dabei haben Individuen mit einer niedrigeren Fitness einen höheren Prozentsatz gelöscht zu werden. Diese können grafisch in einem Kreis angeordnet werden, auf diesem Kreis wird bei der Selektion zufällig ein Punkt ausgewählt und das zugehörige Individuum gelöscht.

3.4 Unterarten

Es werden zwei Arten von evolutionären Algorithmen unterschieden. Die Evolutionsstrategien und die genetische Algorithmen, wobei in diesen noch in weitere

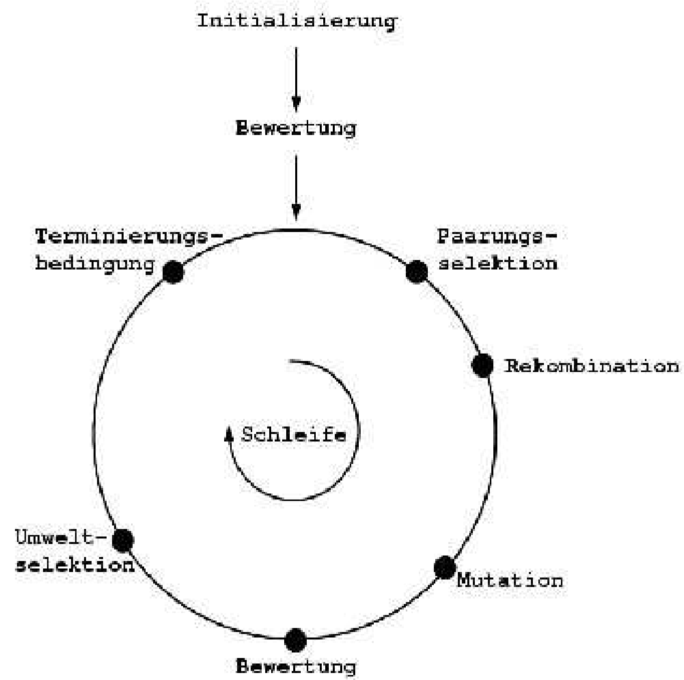

Abbildung 3.1: Allgemeiner Algorithmus

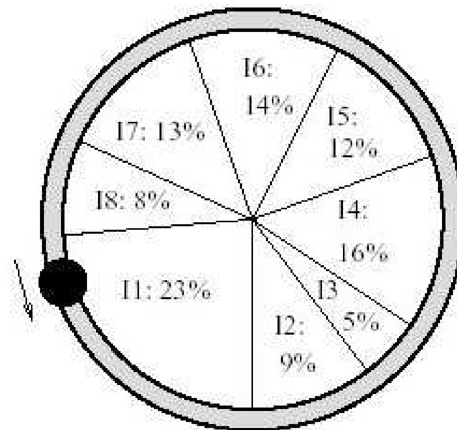

Abbildung 3.2: weel-selection

Unterarten unterschieden werden kann (z.B. sequentielle und parallele oder klassische GA und genetische Programmierung). Eine mögliche Einteilung ist im Bild 3.3 (übernommen aus [3]) zusehen, dabei sind die evolutionären Algorithmen, neben den Simulated Annealing, als eine Unterart der informellen Zufallssuche eingeordnet.

Wo denn nun genau diese Unterschiede liegen, ist oft nicht mehr so klar zu erfassen: [9] "Within the past five years, each area of evolutionary computation has borrowed and modified ideas from the others. Over time, an iterative blending has occurred such that all classes of evolutionary algorithms now appear quite similar, if not for all intents and purposes identical. ..." Womit eine genaue Einteilung von realisierten Algorithmen nicht mehr so leicht möglich ist.

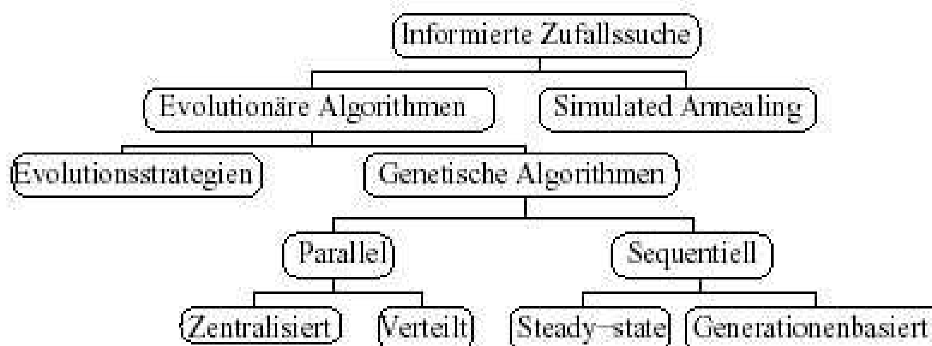

Abbildung 3.3: Einteilung von Lernverfahren

3.4.1 Genetische Algorithmen (GA)

Genetische Algorithmen können einfach beschrieben werden als: evolutionäre Algorithmen + crossing over

Vorbild: natürliche Evolution auf Genbasis

Dabei wird als crossing over der Vorgang bezeichnet, bei den zwei oder mehr Individuen ihre (Erb-)Informationen kombinieren um ein neues Individuum zu schaffen.

3.4.2 Klassische genetische Algorithmen (GA)

Die klassischen genetische Algorithmen (GA) sind Algorithmen die auf einer Liste mit Bits arbeiten, die die potentiellen Problemlösungen/Individuen repräsentieren. Mutation ist dabei das invertieren von Bits. Beim crossing over gibt es meist one oder two point crossing over (bei GA's). Beim one point crossing over wird

die Bitsequenz der beiden Individuen jeweils an einem Punkt, meist die gleiche Position, durchgeschnitten und die beiden Teilhälften eines Individuums mit den entsprechend anderen Teilhälften des anderen Individuums verbunden. Two point crossing over ist identisch nur das hier an zwei Punkten geschnitten wird. Die Arbeitsweise von one point crossing over ist in Bild 3.4 gezeigt, die von two point crossing over im Bild 3.5 (beide übernommen aus [3]).

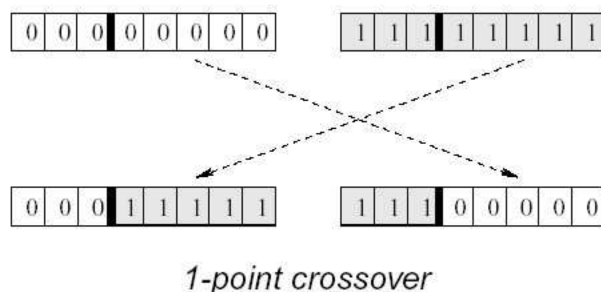

Abbildung 3.4: one point crossing over

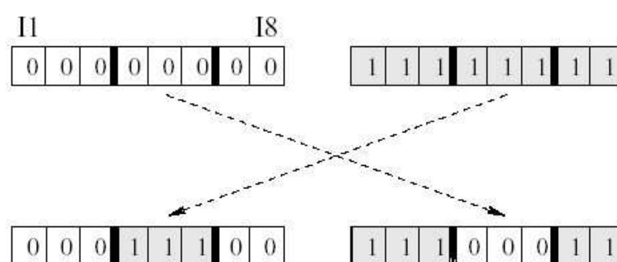

Abbildung 3.5: two point crossing over

3.5 Anmerkungen zu EA

Eine geeignete Wahl der Repräsentation und angepasste Operatoren können die Performance deutlich erhöhen. [9] Durch eine geeignete Wahl der Repräsentation wird der Hypothesenraum, den diese aufspannen, günstig beeinflusst, z.B. können ungültige oder unerwünschte Lösung ganz entfallen. Weiterhin ermöglicht eine gute Repräsentation auch einen Hypothesenraum, in denen Operatoren einfacher zu realisieren sind, die im allgemeinen schneller zur Lösung führen, z.B. ist es dann wahrscheinlicher das ähnliche Lösungen auch nahe im Hypothesenraum "beieinander liegen"(für die Operatoren). Durch angepasste Operatoren werden die "Wege" die diese hervorbringen besser, bzw. führen schneller zu besseren Lösungen, z.B.

können als Operatoren gleich schon bekannte Algorithmen zur Optimierung eingesetzt werden oder in ihnen Wissen über das Problem verwendet werden. Die große Freiheit bei der Wahl der Repräsentation der potentiellen Problemlösung ist eine der größten Vorteile der evolutionären Algorithmen. Theoretisch ist eine beliebige Repräsentation wählbar, solange eine Bewertung von potentiellen Problemlösungen in dieser möglich ist. Andere Lernverfahren ermöglichen meist nur bestimmte Repräsentationen, z.B. können Lernverfahren von Neuronalen Netzen nur Neuronale Netze als Problemlösung hervorbringen. Bei EA's allerdings können unter anderen Bitlisten, Moleküle, beliebige Programmiersprachen oder mathematische Formeln verwendet werden. Genauso variabel sind damit auch die Operatoren, mit denen die potentiellen Lösungen angepasst werden können. Damit kann viel besser Vorwissen oder auch Vermutungen in den Algorithmus eingearbeitet werden.

Weiterhin kann bei der Wahl der Repräsentation auch noch die Anschaulichkeit für den Menschen berücksichtigt werden. Das heißt die Repräsentation kann so gewählt werden, dass potentielle Problemlösungen einfacher verstanden werden können.

Der große Nachteil von evolutionären Algorithmen ist der hohe Zeitaufwand bzw. Rechenaufwand den sie im Verhältnis zu anderen Algorithmen benötigen. Wo andere Algorithmen die vorhanden Daten nutzen, um aus ihnen einmal eine Lösung zu generieren, generiert ein genetischer Algorithmus ständig neue Lösungen, testet wie gut diese sind und wählt möglichst die Guten aus. Solange die Terminalkondition nicht erfüllt ist, hält ein genetischer Algorithmus auch nicht an. Die Terminalkondition kann aber auch sehr schwer oder gar nicht erfüllbar sein. Ein gutes Beispiel ist auch hier die natürliche Evolution, sie läuft schon seit mindestens 4 Milliarden Jahren auf wenigstens einen ganzen Planeten und es ist noch kein Ende abzusehen.

Kapitel 4

Bilder

4.1 Einleitung

Wie auch in vielen anderen Bereichen, wurde mit dem Aufkommen leistungsfähiger Computer die Bildverarbeitung revolutioniert. Handelte es sich Anfang der 60er Jahre [15], als die grafische Datenverarbeitung aufkam, noch hauptsächlich um Strichzeichnungen und Diagramme, die erstellt, manipuliert und verarbeitet wurden, so sind es heutzutage ganze Filme die mit Computern erzeugt werden. Es ist schon heute so gut wie unmöglich ein durch gute Manipulationen erstelltes Bild von einem echten Bild zu unterscheiden. Heute stehen eine Unzahl von Programmen zur Erstellung, Manipulation und Verarbeitung von Bildern zur Verfügung und damit einhergehend eine Unzahl von Bildformaten. Alle diese Bildformate haben ihre Vor- und Nachteile und werden in den verschiedensten Bereichen genutzt.

Im Nachfolgenden sollen einige wichtige Eigenschaften von Bildcodierungen näher erläutert werden.

4.2 Bildformat

Ein Bildformat oder Bildcodierung ist eine Struktur einer Ansammlung von Daten, die ein Bild repräsentieren. Dabei kommt es vor allem auf die effiziente Organisation der Daten im Bezug auf das Anwendungsgebiet an. Die meisten Bildformate teilen ihre Daten in zwei Blöcke auf, einen Block mit allgemeinen Informationen zum Bild (Farbpalette, Bildgröße, etc.) und einen Block der angibt was auf dem Bild dargestellt wird (z.B. die Objekte oder die Farbwerte für die einzelnen Pixel). In Bild 4.1 ist als Beispiel der Aufbau einer Datei im Bitmapformat BMP (*.bmp) angegeben. Im Information Header befinden sich Informationen zum Bild selbst. Unter anderem die Höhe und Breite des Bildes, die horizontale und vertikale Auflösung in Pixel pro Meter, der Typ der Komprimierung und die Anzahl der benutzten Farben.

Die Farbpalette definiert jede Farbe durch ihren Anteil an Rot, Grün und Blau. Die Daten enthalten die zeilenweise Rasterinformation des Bildes. Hierbei ist der

Ausgangspunkt die linke untere Ecke des Bildes.

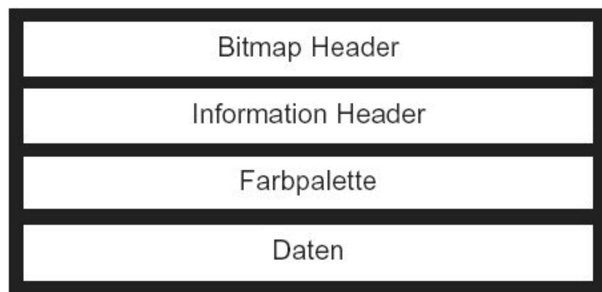

Abbildung 4.1: Dateiaufbau eines Bildes

4.3 Eigenschaften von Bildern

File Type ist das, woran im Dateisystem erkannt werden kann (oder zumindest erkannt werden sollte) um welche Art von Bildcodierung es sich handelt. Meist handelt es sich dabei um ein Dateiendungskürzel, das eine Art Abkürzung der vollen Bezeichnung der Bildcodierungsart ist, z.B. die Dateiendung .bmp für die Bitmap Codierung.

Die maximale Bildgröße gibt an, wie die maximale Größe eines Bildes sein darf, um in diesem Bildformat noch abspeicherbar zu sein. Typisch ist hier die maximale Anzahl der Pixel die ein Bild haben darf. Die maximale Bildgröße hängt vom Speicherformat ab. Da im digitalen Rechnern nicht unendlich große Zahlen abgespeichert werden können (begrenzter Speicherplatz) können auch die Zahlen für Koordinaten oder Bildgröße nur endlich sein.

Die Anzahl der Kanäle gibt an, aus wie vielen Teilbildern übereinander das Bild besteht. Üblich ist ein Kanal, aber in einigen Anwendungen ist es notwendig, mehrere Bilder übereinander zu legen, um z.B. besser Transparenzen im Bild darstellen zu können.

Das Farbmodell, ist das Modell, für die Farben die verwendet werden können. Der einfachste Fall ist z.B. S/W (nur schwarz und weiß als ”Farben“).

Farbtiefe gehört zum Farbmodell und ist die Anzahl möglichen Farben, die einem Pixel oder Objekt zugeordnet werden können. Farbtiefe in Bits, ist die Anzahl der Bits pro Farbe, die einem Pixel oder Objekt für dessen Farbe zugeordnet werden. Die beiden Begriffe hängen eng zusammen, da die Bits pro Farbe meist die Farbtiefe bestimmen, nach der Formel: $Farbtiefe = 2^{(Bits\ pro\ Farbe)}$

Der Hersteller eines Bildformates ist wichtig im Hinblick auf Lizenzen und Updates. Muss für ein Bildformat bezahlt werden, wenn es in seiner Software benutzen werden soll? Wird dieses Bildformat in Zukunft eventuell noch besser ausgebaut? usw.

Plattformabhängigkeit: Bei vielen unbekannteren Bildformaten gibt es nur für einige Plattformen Programme zum Erstellen, Bearbeiten oder Darstellen von Bildern in ihnen.

Als Komprimierung wird bezeichnet, wie viel Informationen für ein Bild benötigt werden, um es aus diesen wiederherzustellen.

Die Darstellungsarten Vektorgrafik oder Bitmapgrafik beziehen sich auf die Art, wie die Bildinformationen abgespeichert sind. Bei Bitmapbildern wird über das Bild ein Raster gelegt und die einzelnen Punkte abgespeichert. Bei Vektorgrafik werden die ”Objekte“ (z.B. Linien, Kreise) des Bildes abgespeichert.

4.4 Codierungen in den Darstellungsarten Vektorgrafik oder Bitmapgrafik

4.4.1 Vektorgrafik

Bei der Vektorgrafik werden die Daten für einzelne Objekte, Vektoren genannt, gespeichert. Der Vektorgrafiktyp bestimmt die möglichen Objekte. Oft finden dafür Punkte, Linien und Kreise Verwendung. Der Vektorgrafiktyp und damit die verwendeten Objekte orientieren sich meist stark an dem Anwendungsgebiet, für das dieser Vektorgrafiktyp geschaffen wurde, z.B. technischer Entwurf von Gebäuden oder Geräten. Die Objekte werden mathematisch durch eine Anzahl von Werten definiert, z.B. Anfangsposition, Länge oder Radius. Vektorgrafiken sind vor allem bei technischen Zeichnungen verbreitet (z.B. CAD Systeme).

Vorteile:

- Grafikobjekte können ohne Qualitätsverlust verschoben, skaliert oder mit Farbe versehen werden
- ein Vektorbild besteht aus vielen verschiedenen Einzelobjekten, die sich bei Veränderungen gegenseitig nicht beeinflussen
- im Vergleich zu Rastergrafiken wird hier meist weniger Speicherplatz benötigt, da eine geringere Anzahl von Informationen gespeichert wird (z.B. einen Kreis als Matrix aus Farbwerten zu beschreiben ist aufwendiger, als einen Kreis mit Mittelpunkt und Radius zu beschreiben)
- Vektorgrafiken sind nicht an eine bestimmte Auflösung gebunden, d.h. sie passen sich den Möglichkeiten des Ausgabegeräts an

Nachteile:

- zur Bildschirm und Druckerausgabe müssen die Vektorgrafiken gerastert werden, da diese Geräte nur Punkte darstellen können, ein Treppeneffekt ist die Folge
- eine direkte Umwandlung von Bitmapgrafik oder Vektorgrafiken eines Typs in eine Vektorgrafik anderen Typs, ist meist nur schwer und mit viel Aufwand zu realisieren (bei Vektorgrafiken geht es darum, die Objekte des Bildes darzustellen; wie findet man aber solche bei Bitmapgrafiken? einfach jeden Punkt als Objekt (z.B. als Rechteck) zu deklarieren, würde keinen der oben genannten Vorteile bringen)
- die meisten Bilder liegen nach dem Einlesen in einem Rastergrafikformat vor

4.4.2 Bitmapgrafik

Bei der Bitmapgrafik oder Rastergrafik wird über das Bild ein Raster gelegt und die einzelnen Punkte bzw. ihre Farbe (im Farbmodell) abgespeichert. Damit besteht ein Bitmapbild aus einer Menge von Pixel (Bildpunkten) die eine bestimmte Farbe haben. Bei genügend vielen Pixeln pro Fläche kann das menschliche Auge die einzelnen Pixel nicht mehr wahrnehmen und es entsteht der Effekt eines ”fotorealistischen“ Bildes.

Vorteile:

- einfaches Abspeichern von Bildern, es müssen keine Objekte bekannt sein
- da die meisten Geräte (z.B. Monitor, Scanner, Drucker) zum Darstellen oder Einlesen von Bildern mit Rastern arbeiten, ist die Umwandlung in oder von Bitmapgrafiken einfach

Nachteile:

- bei starker Vergrößerung eines Bitmapbildes kommen die einzelnen Pixel zum Vorschein und die Grafik wirkt eckig
- das jeder einzelne Punkt eines Bildes abgespeichert wird, kostet viel Speicherplatz
- Objekte des Bildes können nur noch schwer oder gar nicht automatisch identifiziert werden

4.5 Farbmodell/Farbschema

Das Farbmodell gibt an, wie die Farben im Bild dargestellt werden. Es handelt sich bei diesen Farben immer um einen Vektor mit ganzen Zahlen aus einem Wertebereich. Dieser Vektor wird dann jeweils einem Pixel oder anderem Objekt (bei Vektorgrafik) zugeordnet. Der Darstellbarkeit wegen, werden schwarz und weiß auch als Farben angesehen, ihnen ist im allgemeinen auch ein Vektor zugeordnet. Die Anzahl der Farben die ein Bild haben kann, wirkt sich sehr stark auf die Größe des nötigen Speichers für dieses aus.

Im Nachfolgenden sind einige bekannte Beispiele aufgeführt.

4.5.1 S/W (schwarz/weiß)

Das S/W (für schwarz/weiß) Farbmodell ist wohl das älteste, einfachste und am wenigsten speicheraufwändigste Farbmodell. Dabei wird pro Objekt (z.B. pro Pixel) ein Bit gespeichert, das je nach Wert (z.B. weiß = true, schwarz = false) angibt, ob das Objekt weiß oder schwarz sein soll, bzw. heller oder dunkler darzustellen ist, z.B. für Monitore die kein wirkliches Schwarz oder Weiß darstellen können.

4.5.2 Grayscale (Halbton)

Beim Grayscale werden 256 Abstufungen von Schwarz bis Weiß unterschieden. Pro Objekt wird dafür ein Byte oder 8 Bit abgespeichert, dies wird dann als eine dieser Abstufungen interpretiert.

4.5.3 RGB

Das RGB Farbmodell ist ein additives Verfahren, mit den drei Grundfarben Rot, Grün und Blau. Jeder dieser Grundfarben ist ein Wert zugeordnet, um so größer dieser ist, um so heller ist der entsprechende Anteil dieser Grundfarbe in der erzeugten Farbe, dargestellt in Bild 4.2 (übernommen aus [15]). Mit diesem Farbmodell können alle Farben mit einer Genauigkeit dargestellt werden, die mit der Wertebereichsgröße der einzelnen Farben ansteigt. Das Farbmodell hat seinen Ursprung in der Realisierung von Farben auf Monitoren (Braunsche Röhren), da auch dort nur drei verschiedenfarbige Punkte zu einem Pixel zusammenfasst werden, je stärker der Elektronenstrahl einen dieser Punkte anstrahlt, desto heller erscheint er in seiner Farbe (auch RGB). Daher eignet sich dieses Farbmodell auch besonders gut für diese Monitordarstellungen, es muss nicht mehr viel umgerechnet werden.

4.5.4 Induzierte Farben

Bei induzierten Farben wird nicht immer der gleiche ”Farbraum“ für jedes Bild verwandt, sondern je nach Bild, je nach dem welche Farben bei ihm benötigt werden, nur diese Farben oder ähnliche. Jede dieser Farben bekommt eine ”Nummer“

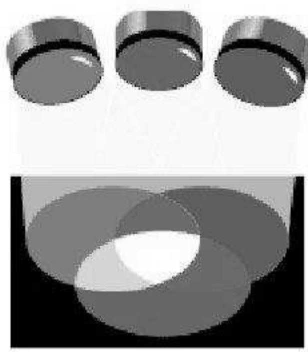

Abbildung 4.2: additives Farbschema

(Indices), die in einer Tabelle, der wirklichen Farbe (z.B. im RGB Farbmodell) zugeordnet wird. Da im Bild nun viel weniger Farben dargestellt werden müssen, wird Speicherplatz gespart.

4.6 Komprimierung

Bei der Komprimierung geht es darum, Speicherplatz für ein Bild zu sparen. Um so weniger ein Speicherformat für ein Bild Speicherplatz benötigt, um so höher wird der Kompressionsfaktor dieses Speicherformats angesehen und damit auch eines mit ihm abgespeicherten Bildes. Leider ist die Festlegung für welches Speicherformat der Kompressionsfaktor Null ist, mehr oder weniger willkürlich. Denn es können immer zu einem Speicherformat weitere Informationen über das abgespeicherte Bild hinzugefügt (z.B. Redundanzen, History) und der Kompressionsfaktor verschlechtert werden. Der Ausdruck ein Bild sei ”unkomprimiert“ ist somit nicht zutreffend. Zutreffender ist, das solche Bilder einen niedrigen Kompressionsgrad haben.

Es werden zwei große Komprimierungsarten unterschieden.

4.6.1 Verlustfreie Komprimierung

Dabei gehen keine Informationen über das Bild verloren. Das heißt, wenn ein Bild mit einem verlustfreien Verfahren komprimiert und danach wieder herstellt wird, ist das Abbild mit dem Original identisch.

4.6.2 Verlustbehaftete Komprimierung

Dabei gehen Informationen über das Bild verloren. Das heißt, wenn ein Bild mit einem verlustbehafteten Verfahren komprimiert und danach wieder hergestellt wird,

ist das Abbild mit dem Original nicht mehr identisch. Es können z.B. Pixel mit einer leicht abweichenden Farbe im Abbild vorhanden sein.

In dieser Komprimierungsart sind meist höhere Komprimierungsfaktoren möglich als mit der verlustfreie Komprimierung.

Kapitel 5

Anmerkungen zur Aufgabenstellung

Die zentrale Komponente der Idee ist, ein Pixelbild in ein weniger speicherintensives Vektorbild umzuwandeln. Im Allgemeinen gibt es dafür keinen optimalen Algorithmus.

Im ersten Schritt ist dafür eine geeignete Sprache (Bildbeschreibungssprache, Vektorformat) zur Darstellung von Problemlösungen (Vektorbilder) des genetischen Algorithmusses zu entwerfen. Dabei ist abzuwägen, ob eine einfache klassische Bitdarstellung, mit Operationen auf ihr, günstiger ist oder eine komplexere mehr auf das Problem zugeschnittene Darstellung. Bei der Wahl der klassische Bitdarstellung wird die Realisierung wahrscheinlich einfacher werden. Eventuell kann auch auf schon vorhandene, frei verfügbare genetische Algorithmen zurückgegriffen werden. Allerdings ist deshalb eine sehr geringe Performance zu erwarten, da die klassischen Algorithmen nicht für so komplexe Probleme konzipiert wurden. Wird eine komplexere, passendere Darstellung gewählt, ist damit zu rechen, dass um so angepasster und komplexer die Darstellung ist, um so aufwändiger sich dann auch deren Realisierung gestaltet. Es muss überlegt werden, wie die Informationen zum Farbmodell realisiert werden. Wie werden Farben in der Bildbeschreibungssprache realisiert? Wird sich auf ein Farbmodell (z.B. RGB) beschränkt? Werden zusätzliche Informationen zum Farbmodell, eventuell außerhalb eines Objekts der Bildbeschreibungssprache, gespeichert?

Die Darstellung (Bildbeschreibungssprache) und der Algorithmus ist dann, in einer geeigneten Sprache, zu implementieren.

Die aufgezeigte Lösung ist entsprechend zu analysieren.

Teil III

Entwurf der Bildbeschreibungssprache und der genetischen Operationen

Kapitel 6

Anforderungen

Da die klassische Bitdarstellung geringe Performance erwarten lässt, es aber gerade wegen der Komplexität des Problems eine möglichst hohe Performance von Nöten ist, wird eine mehr angepasste Problemdarstellung (Bildbeschreibungssprache) gewählt. Die Bildbeschreibungssprache sollte möglichst einfach gehalten werden, mit möglichst wenigen Alternativen, da mit dem Anstieg der zur Auswahl stehenden Alternativen, auch die Anzahl der Alternativen bei der Anwendung der genetischen Operationen steigt und damit wahrscheinlich der Rechenaufwand und der Realisierungsaufwand. Es soll mit der Bildbeschreibungssprache möglich sein, alle möglichen Bilder darzustellen. Die Erzeugung eines Bildes aus einem Programm in der Bildbeschreibungssprache soll nachvollziehbar sein. Das heißt, die Bildbeschreibungssprache soll die Unterscheidung einzelner Objekte und deren Zusammenhang unterstützen.

Die Bildprogramme verändernden genetischen Operationen sollten, wenn möglich, das Programm so verändern, dass im Hypothesenraum der Bildprogramme diese Operatoren einen Gradientenanstieg ermöglichen. Durch Ausführung mehrerer Operatoren sollte also die Hypothese, die das Bildprogramm darstellt, allmählich verbessert werden können.

Kapitel 7

Warum sich genetische Algorithmen zur Bildcodierung anbieten

Es gibt im allgemeinen keinen Algorithmus der Bitmapbilder direkt in speicherschonende Vektorbilder umwandeln kann, bei denen die Stärken der entsprechenden Vektorbeschreibungssprache wirklich genutzt werden.

Da ein genetischer Algorithmus das Potential hat alle möglichen Vektorbeschreibungen eines Vektorbildformats zu generieren und unter diesen natürlich auch gute Vektorbeschreibungen sind, welche die Stärken der Vektorbeschreibungssprache, im Bezug auf das Originalbild nutzen, hat ein genetische Algorithmus natürlich auch das Potential gute Vektorbeschreibungen zu generieren. Wenn im genetischen Algorithmus vorhandenes Wissen gut eingebaut wurde, kann er gute Vektorbeschreibungen wahrscheinlich auch schneller finden als eine reine Zufallssuche.

Noch ein weiterer Vorteil von genetischen Algorithmen ist die große Freiheit bei der Wahl der Problembeschreibung (Vektorbildbeschreibungssprache) und der möglichen Operatoren auf dieser. So kann völlig frei eine Vektorbildbeschreibungssprache nach eigenen Vorstellungen entworfen werden, die bestimmte Eigenschaften hat, z.B. Lesbarkeit, Einfachheit. Bei den Operatoren kann beliebig viel Wissen eingebaut werden. So ist es unter anderem auch möglich, schon bekannte gute Algorithmen zur Übersetzung von Bitmapbilder in Vektorbilder oder Teile von ihnen in Operatoren zu verwenden, so dass der genetische Algorithmus vom Ergebnis her mindestens so gut wird, wie der verwendete Algorithmus, aber wahrscheinlich noch bessere Ergebnisse erzeugen kann.

Der große Nachteil von genetischen Algorithmen, dass sie sehr viel Zeit oder Rechenaufwand benötigen, wird dadurch abgeschwächt, dass dieser anfänglich hohe Aufwand "billig" sein kann und sich später auszahlen kann. Der genetische Algorithmus kann z.B. als Hintergrundprozess mit niedriger Priorität laufen, so dass er nur überflüssige Rechnerleistung verbraucht. Später kann durch das Ergebnis,

das er geliefert hat, viel Übertragungsbandbreite eingespart werden.

Dies alles spricht für den Versuch genetische Algorithmen zur Bildcodierung zu verwenden.

Kapitel 8

Die Bildsprache

Die Bildsprache soll den Namen fib tragen (für funktionale Interpretation von Bildern oder "functional interpretation of bictures/bitmaps"). Fib ist eine Vectordarstellung von Bildern, also eine Darstellung von Bildern mit Hilfe von Objekten. Als Grundgerüst dient ein Baum. Die Blätter sind Endpunke, die für die Darstellung von Punkten dienen. In den Ästen und der Ausrichtung dieser, welche z.B. am weitesten links stehen, werden Darstellungsparameter der Blätter kodiert, z.B. wie oft es dargestellt wird.

8.1 Elemente

Im Nachfolgendem bezeichnet Obj ein fib-Objekt. Es ist nicht erlaubt, dass in einem fib-Objekt ein fib-Teilobjekt mehr als einmal vorkommt. Daraus folgt, das fib-Objekte zyklenfrei sind und keine fib-Äste irgendwann zusammenwachsen.

8.1.1 Vektoren

Eine Koordinate ist ein Vektor, der aus ganzen Zahlen oder Variablen mit dem Wertebereich von ganzen Zahlen bestehen kann, die die Position des abzubildenden Punktes angeben.

Die Farbe ist ein Vektor, der die Farbe des Punktes, mit Hilfe von ganzen Zahlen oder Variablen mit dem Wertebereich von ganzen Zahlen beschreibt (z.B. mit der RGB Darstellung). Welches Farbschema für das fib-Bild verwendet werden soll, muss extern gespeichert werden, wenn es sich nicht implizit ergibt (z.B. Farbvektor mit 3 Komponenten-> RGB Farbschema).

Wird einer Koordinate oder einer Farbe eine Variable mit einem Wert übergeben, der außerhalb des Wertebereichs seiner Komponenten liegt, wird dieser, wenn er nicht ganzzahlig ist, gerundet oder wenn er außerhalb des Wertebereichs liegt, auf den nächsten Wert, der im Wertebereich liegt, gerundet.

8.1.2 Punkte

Semantik: Die Punkte sind sozusagen die darstellenden Objekte. Sie werden an der angegebenen Koordinate mit der angegebenen Farbe in die Bildmatrix des Bildes eingefügt.

mögliche Syntax: Obj = p(Koordinate, Farbe)

Beispiel: p((27;3), (0;0;0))

8.1.3 conc

Semantik: Mit dem Objekt conc können zwei Objekte zu einem Objekt zusammengefügt werden. Dabei muss eine Ausführungsreihenfolge festgelegt werden, um im Falle einer Überdeckung der Objekte zu gewährleisten, dass immer dasselbe Objekt das andere Objekt überdeckt und dies nicht z.B. immer wechselt (Eindeutigkeit von fib-Bildern).

mögliche Syntax: Obj = conc(Obj1,Obj2)

Obj1 ;Obj2 sind die Arme/Zweige des conc Objekts.

8.1.4 Bereichsobjekt

Semantik: Das Bereichsobjekt legt für eine Variable aus dem Bereich der ganzen Zahlen die gültigen Bereiche, die sie einnimmt, fest. Das Bereichsobjekt enthält, neben der Variablen und dem Objekt in dem sie gilt, eine Liste von Bereichen, die die Variable annehmen kann. Ist die Liste leer, nimmt die Variable keinen Wert an. Die Variable gilt überall in dem enthaltenden Objekt.

mögliche Syntax: Obj = for(Variable,$[B_1, B_2, \ldots, B_n]$,Obj1)

Dabei sind die B_i $(i = 1 \ldots n)$ die Bereiche.
Ein Bereich ist ein Vektor vom Grad 2, dessen zwei ganzzahligen Komponenten einen ganzzahligen Bereich festlegen, in dem die Variable gilt. Ist eine Komponente des Vektors eine Variable, die einen nicht ganzzahligen Wert enthält, wird diese gerundet.

Beispiel: for(x,[(1;3),(10;14)],Obj) In diesem Beispiel nimmt die Variable x im Objekt Obj hintereinander die Werte 1; 2; 3; 10; 11; 12; 13; 14 an.

Anmerkung: Durch diese Definition des Bereichsobjekts sind manche kontinuierlichen Funktionen schwerer zu realisieren, da das Bereichsobjekt nur einzelne Punkte (ganze Zahlen) zulässt und keinen kontinuierlichen Übergang.

Beispiel: Die Funktion $y = x^2$ im Bereich von x = -2 bis 2 (nur zur Verdeutlichung einfach gewählt)

Wenn sie in der Form: for(x,[(-2;2)], fkt(y,[(1;x;2)], p((x,y), Farbe))) zu realisieren versucht wird, entstehen Lücken (der Übergang von (1;1) zu (2;4) ein Punkt (x;3) fehlt).
Dies kann aber durch "Stauchung des Wertebereichs" behoben werden:
for(x,[(-6;6)], fkt(sx,[(x;3;-1)], fkt(y,[(1;sx;2)], p((sx,y), Farbe))))
Nun existiert auch ein Punkt (2;3) ($x = 5 \rightarrow sx = 5/3$ aufgerundet 2; y = 2,78 aufgerundet 3)

Dies wurde zugunsten der einfacheren Realisierung und besseren Performance so gewählt. Damit entfällt leider der Vorteil von Vektorgrafiken, dass sie ohne Qualitätsverlust skaliert werden können. Bilder die in der fib-Vectorgrafik codiert sind, werden leider bei der Darstellung aus einzelnen Pixelpunkten bestehen.

8.1.5 Funktionen

Semantik: Funktionen sind Objekte, welche einer Variable einen Wert zuordnen, der mit Hilfe einer Formel (Polynom) berechnet wird. Die Formel wird in Form einer Liste aus Vektoren für Teilformeln angegeben, wobei die Ergebnisse der einzelnen Teilformeln in ihr aufsummiert werden. Ist die Liste leer, ist ihr Wert 0. Die Variable gilt überall in dem enthaltenden Objekt.

mögliche Syntax: Obj = fkt(Variable,$[T_1, T_2, \ldots, T_n]$,Obj1)
T_i $(i = 1 \ldots n)$ sind die Teilfunktionen. (Statt "fkt" kann auch "fun" verwendet werden.)

Eine Teilfunktion ist ein Vektor vom Grad 3, der aus ganzen Zahlen oder Variablen bestehen kann. Dabei wird ihr Wert folgendermaßen berechnet: Für den Vektor (a,b,c)
$Teilfunktionswert = a * b^c$

Anmerkungen: Als Werte des Vektors sind nur ganze Zahlen zugelassen, um die Mutationsoperationen möglichst einfach zu halten. Es können aber in Formeln versteckt auch rationale Zahlen einfließen, indem sie in einer oder mehreren Funktion erzeugt werden und in der Formel dann als Variablen auftreten. (Rationale Zahlen sind in der Realisierung natürlich immer nur genähert.)

Beispiel: Die Funktion $x = 2c^{1/2}$ lässt sich nicht mit einem Funktionsobjekt darstellen, da $\frac{1}{2}$ keine ganze Zahl ist.
Durch fkt(h,[(1;2;-1)],Obj) steht h für $\frac{1}{2}$.
Eingefügt könnte also die obige Funktion für x wie folgt realisiert werden:
fkt(h,[(1;2;-1)],fkt(x,[(2;c;h)],Obj))

8.1.6 Sinusfunktionen (geplant)

Semantik: Funktionen sind Objekte, welche einer Variable einen Wert zuordnen, der mit Hilfe einer Formel (Sinusformel) berechnet wird. Die Formel wird in Form einer Liste aus Vektoren für Teilformeln angegeben, wobei die Ergebnisse der einzelnen Teilformeln (Sinusfunktionen) in ihr aufsummiert werden. Ist die Liste leer, ist ihr Wert 0. Die Variable gilt überall in dem enthaltendem Objekt.

mögliche Syntax: Obj = sin(Variable,$[T_1, T_2, \ldots, T_n]$,Obj1)
T_i $(i = 1 \ldots n)$ sind die Teilfunktionen.

Eine Teilfunktion ist ein Vektor vom Grad 5, der aus ganzen Zahlen oder Variablen bestehen kann.
Dabei wird ihr Wert folgendermaßen berechnet: Für den Vektor (a,b,c,d,e)
$Teilfunktionswert = \sin{(a/b + c)} * d + e$

Anmerkungen: Diese Funktion kann einfach aus den allgemeinen Funktionen hergestellt werden und ist somit leicht zu realisieren. Da die Sinusfunktionen, bzw. Kosinusfunktion, in Verbindung mit der Fouriertransformation häufig in der Bildverarbeitung vorkommt, dürfte die Sinusfunktion die fib-Bildbeschreibungssprache bereichern. Die Kosinusfunktion kann dabei leicht aus der Sinusfunktion gewonnen werden (durch Addition von $90°$). Der Grund, warum die Teilfunktionen so viele Parameter haben (wo doch schon Einer ausreicht $[\sin x]$), ist, dass sie so schon mit einem vorgeschalteten einfachen Bereichsobjekt zu einem sinnvollen Ergebnis führen kann und ihre Eingabe- und Ausgabewerte nicht noch durch andere Funktionsobjekte skaliert werden müssen.

8.1.7 Hintergrundfarbe (herausgenommen)

Ein solches Objekt macht die Berechnung nur komplexer, ohne die Beschreibungsmöglichkeiten der Sprache wesentlich zu erhöhen. Es können auch anders Flächen gefüllt und eine Hintergrundfarbe definiert werden.

Semantik: Die Hintergrundfarbe ist ein Objekt, das genau einmal existiert und unter der ersten conc Verzweigung stehen muss. Sie dient nur dazu ein Objekt zu haben mit dessen Hilfe die Bildmatrix mit Farbwerten gefüllt wird.

mögliche Syntax: Obj = hcolor(Farbe,Obj)

Beispiel: hcolor((1;0;0),Obj)

8.2 Definitionen für fib

8.2.1 Definition korrektes/vollständiges fib-Objekt

Ein fib-Objekt ist korrekt bzw. vollständig, wenn es der oben aufgeführten fib-Syntax entspricht, alle Variablen, die es enthält, über ihm definiert sind und jedes

enthaltende fib-Objekt in ihm ein korrektes bzw. vollständiges fib-Objekt ist. Korrekte bzw. vollständige fib-Objekte werden auch kurz nur als fib-Objekte bezeichnet. Ein fib-Element entspricht der oben vorgestellten fib-Syntax, außer das keine fib-Objekte in ihm enthalten sind.

Beispiele:
fib-Objekte:
Obj = p((1;5), (3)) //Vektor (3) für die Farbe
Obj = conc(for(y,[(4;2)],p((7;y), (6;8))), fkt(x,[(4;3;-2),(2 ;1 ;1)],p((3;x), (x;7))))

fib-Element (elm):
elm = p((3;2;5),(12;7;9)) //3-dimensional
elm = for(x,[(3;8)],obj0) //obj0 ist kein Objekt (bei der Implementierung Null-Pointer)
elm = p((2;5),(12;x;9))

weder fib-Objekt noch fib-Element (woe):
woe = conc(for(x,[(3;7)],obj0),obj0)
woe = p(,(3;7))
woe = fkt(x,[(4;6;3)],for(y,[(6;7)],obj0))

8.2.2 Definition von ”unterhalb“ und ”oberhalb“ in einem fib-Objekt

Man stelle sich ein fib-Objekt als Baum vor, in dem die conc Objekte die Verzweigungen darstellen. Da in der Informatik im allgemeinen die Wurzel oben dargestellt wird, sind die Objekte die ein Objekt enthält unten und die Objekte die ein Objekt enthalten für dieses oben.

Unterhalb eines Objekts (Nummer 1) bedeutet also, dass die Objekte, die das Objekt (1) direkt oder indirekt enthält, gemeint sind. Oberhalb eines Objekte, die Objekte, die das Objekt (1) direkt oder indirekt, enthalten.

Dieser Sachverhalt ist in Bild 8.1 dargestellt. Das conc Objekt in der Mitte, das mit 1 gekennzeichnet ist, ist das Objekt bezüglich dem oberhalb und unterhalb bestimmt wird.

8.2.3 Definition Teilobjekt

Ein Teilobjekt ist ein Unterobjekt des Bildobjektes, das über eines der beiden Objekte, die ein conc Element enthält, definiert ist.

Jedes Objekt, das in einem conc Arm enthalten ist, ist Teil eines Teilobjekts. Weiterhin gehört zum Teilobjekt alle Funktions- und Bereichsobjekte oberhalb des conc Objekts die Variablen definieren, die in dem entsprechenden Objekt unterhalb des conc Objekts, verwendet werden. Die Vereinigung zweier Teilobjekte ist wieder ein Teilobjekt. Das ganze fib-Objekt selbst ist auch ein Teilobjekt.

Ein echtes Teilobjekt, ist ein Teilobjekt das nicht das fib-Objekt selbst ist.

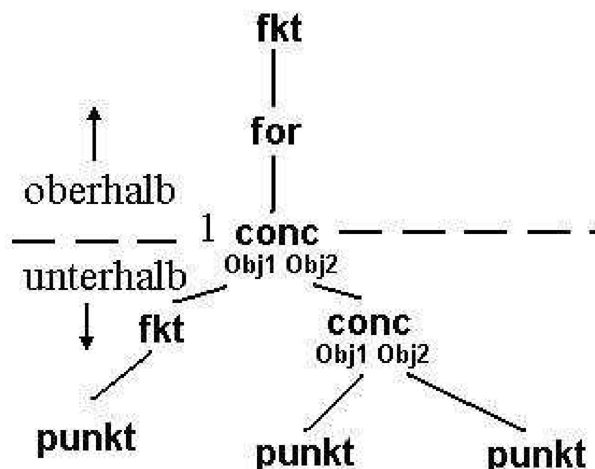

Abbildung 8.1: oberhalb und unterhalb in einem fib-Objekt

Ein einfaches Teilobjekt ist ein echtes Teilobjekt, das nur ein Punktobjekt enthält.

Ein zusammenhängendes Teilobjekt, ist ein echtes Teilobjekt, welches das Objekt eines Armes genau eines conc Objekts enthält und die benötigten Elemente über dem conc Objekt.

Im Bild 8.2 ist diese Definition an einem Beispiel dargestellt.

8.2.4 Definition fib-Bild

Wird der Ausdruck fib-Bild benutzt, ist damit ein fib-Objekt mit Schwerpunkt auf das Bild, das es repräsentiert, gemeint.

8.2.5 Definition korrektes/vollständiges fib-Bild

Ein korrektes fib-Bild, ist ein fib-Objekt, welches das Originalbild, das es repräsentieren soll, vollständig wiedergibt. Wenn also das Bild, welches das fib-Objekt repräsentiert, und das Originalbild verglichen wird, kann kein Unterschied zwischen ihnen feststellt werden.

8.3 Theoretische Aussagen zu fib

Es folgen einige theoretische Aussagen zur fib-Bildbeschreibungssprache, für die aber aus Zeitgründen meist nicht ein vollständiger Beweis angeführt wird.

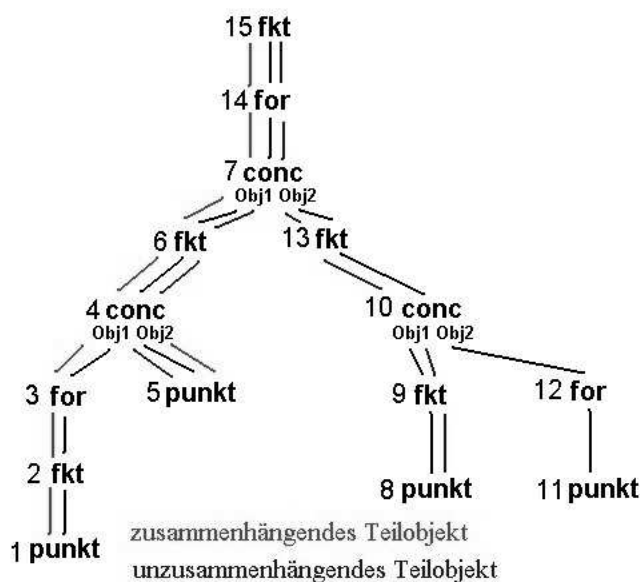

Abbildung 8.2: zusammenhängendes und unzusammenhängendes Teilobjekt

8.3.1 Warum können mit dieser Sprache alle möglichen Bilder dargestellt werden?

Allein mit dem Punkt Objekt und dem conc Objekt lassen sich schon alle möglichen (Bitmap-) Bilder darstellen.

Beweis für zweidimensionale Bilder: Ein Bild (euklidisch, zweidimensional, gequantelt) kann als Matrix dargestellt werden, in der die Spalte die x-Koordinate und die Zeile die y-Koordinate des Punktes angibt und die Werte die Farben der Punkte angeben. Eine Abbildung in die Sprache kann nun mit dem im Listing 8.1 dargestellten Algorithmus geschehen (Pseudo-C).

Dabei beginnt die Induzierung der Matrix mit (0,0).
Der Farbwert der Koordinate (x,y) kann mit matrix[x,y] bestimmt werden.
Die Syntax der fib-Objekte entspricht der oben angesprochenen möglichen Syntax.

Listing 8.1: Algorithmus zur Erzeugung eines korrekten fib-Objekts aus einer Bildmatrik

```
1 void translate(matrix)
2 {
3 int xmax = Anzahl der Spalten der Matrix;
4 int ymax = Anzahl der Zeilen der Matrix;
5 fib_Objekt_Pointer obj;
```

```
6  for(int x = 0;x == xmax;x = x + 1)
7  {
8  for(int y = 0;y == xmax;y = y + 1)
9  {
10     if ((x == 0) and (y == 0))
11       {obj = p((0,0), matrix[0,0]);}
12     else
13       {obj = conc(p((x,y), matrix[x,y]),obj);}
14 };
15 };
16 };
```

Da die Erweiterung mit Funktionen und anderen Objekten optional ist, wird mit dem oben angegebenen Algorithmus ein Objekt dargestellt.

Zu jedem Punkt in der Matrix gibt es einen gleichfarbigen Punkt im fib-Bild mit der entsprechenden Koordinate, aber es gibt keine Punkte im fib-Bild der nicht in der Matrix vorhanden ist. Jede Koordinate der Matrix wird mit den zwei for-Schleifen durchlaufen. Somit wird für jede Koordinate und damit auch für jeden Punkt im Bild ein Punkt im fib-Objekt eingefügt. Somit gibt es für jeden Punkt im Bild einen entsprechenden Punkt in dem fib-Objekt. Da nur Koordinaten der Matrix in den for-Schleifen durchlaufen werden und die entsprechenden Punkte in das fib-Objekt eingefügt werden, existieren nur Punkte die auch in der Matrix auftauchen und damit auch nur Punkte im fib-Objekt, die auch im Bild auftauchen. Es gibt also nur die Punke aus dem Originalbild im fib-Objekt und nur Diese. Damit repräsentieren das fib-Objekt und das Originalbild das gleiche Bild. Außerdem können alle Bilder, bei denen es möglich ist sie mit einer euklidischen, zweidimensionalen und gequantelten (es gibt für Koordinaten und Farben nur diskrete Werte und keine kontinuierlichen) Matrix darzustellen, auch mit fib dargestellt werden.

Ein mit dem Algorithmus generierte fib-Objekte, das einem Originalbild entspricht, stellt eine obere Grenze der minimalen Größe möglicher entsprechender fib-Objekte dar. Das heißt, jedes Bild kann durch ein fib-Objekt repräsentiert werden, das genauso groß ist, wie ein fib-Objekt für das Bild, das mit dem oben aufgeführten Algorithmus generierten wurde, nämlich das generierte fib-Objekt. Es gibt aber wahrscheinlich noch kürzere fib-Objekte für das Bild.

Damit ist die minimale Größe eines fib-Objekts für ein Bild maximal:
fib_max_{min} = (Anzahl der Pixel im Bild) $*$[(Größe eines Punktes) + (Größe eines conc Objektes)] - (Größe eines conc Objektes)

Es gibt für jeden Pixel im Bild ein Punktobjekt und ein conc Objekt, mit dem das Punktobjekt am Rest des Objekts angefügt wird. Mit Ausnahme des ersten Punktes, für den kein conc existiert.

Beispiel: Dargestellt werden soll ein 8 mal 8 = 256 Pixel Bild, mit 3 Byte für die Farbe (RGB), 4 Byte für die Position (für jede Richtung x und y 2 Byte) und 1

Byte für den Objektnamen (z.B. ‚c' für conc und ‚p' für Punkt). Klammern werden nicht benötigt, da alle Teile eine feste Länge haben. Für einen Punkt werden 8 Byte benötigt (3 Byte Farbe + 4 Byte Position + 1 Byte Objektnamen). Für ein conc wird nur 1 Byte benötigt (seinen Namen).

Rechnung: $256(Pixel) * (8Byte + 1Byte) - 1Byte = 2303Byte$

Das Bild kann also auf jeden Fall mit einem 2303 Byte großen fib-Objekt dargestellt werden. Es sind davon abweichende kürzere fib-Objekt Darstellungen möglich. Die 2303 Byte sind also die Obergrenze für die minimale Größe mit der das Bild mit einem fib-Objekt dargestellt werden kann.

Zum Vergleich, der Speicherbedarf eines Bitmapbildes (nur Punkt Informationen) beträgt: (Anzahl der Pixel im Bild) * (Größe eines Farbwertes)

Für das obige Beispiel ergibt sich: $256(Pixel) * 3Byte = 768Bytes$

Das ist ungefähr ein Drittel der 2303 Byte der Obergrenze für das minimale fib-Objekt Darstellung. Ein Faktor der akzeptabel ist.

Zusammenfassend kann gesagt werden, dass die minimale Größe eines fib-Objekts für ein Bild maximal um den, mit der Formel unten ermittelten, Faktor größer ist, als die Bitmapdarstellung.
Faktor = [(Größe eines Punktes) + (Größe eines conc Objekts)] / (Größe eines Farbwertes)

8.3.2 Mächtigkeit von fib

Die Menge der fib-Objekte ist abzählbar unendlich.

Beweis Skizze:
Abzählbar: Jedes fib-Objekt kann mit einer endlichen Anzahl von Buchstaben und damit auch Bits oder Zahlen repräsentiert werden und die Menge dieser ist abzählbar. Das folgt daraus, dass die Anzahl der fib-Elemente in einem fib-Objekt immer abzählbar ist und jedes fib-Element aus abzählbar vielen Teilen besteht und an sich abzählbar ist, es gibt z.B. nur ganze Zahlen und auch die Menge der Variablen ist abzählbar.

Unendlich: Es können alle natürlichen Zahlen durch fib-Objekte repräsentiert werden. Im Nachfolgendem ist eine mögliche Darstellungsart von beliebigen natürlichen Zahlen beschrieben.

Ein Punktobjekt alleine stellt die natürliche Zahl 0 dar. Wird ein fib-Objekt in ein neues Funktionsobjekt eingesetzt, stellt das entstehende fib-Objekt den Nachfolger des ursprünglichen fib-Objekts dar. Damit wird die 0 und die Nachfolgerfunktion in fib nachgebildet und es können damit alle natürlichen Zahlen dargestellt werden. Da die Menge der natürlichen Zahlen unendlich ist, muss die Menge der fib-Objekte auch (mindestens) unendlich sein.

Jedes Bild wird sogar durch eine abzählbar unendliche Menge von fib-Objekten repräsentiert, denn es kann an ein fib-Objekt, das ein Bild repräsentiert, jedes beliebige fib-Objekt angehängt werden, solange dieses die Bilddarstellung nicht verändert. Es könnte z.B. beliebig oft an ein fib-Objekt eine Kopie von sich selbst, mit Hilfe eines conc Objekts, angefügt werden, ohne das Bild zu verändern.

8.3.3 Warum kann jedes mit den fib-Elementen gebildete Objekt als Bild dargestellt werden?

Es wird aufgezeigt, dass es möglich ist, fib-Objekte so zu interpretieren (in ein Bild zu übersetzen), dass keine ungültigen Bilder entstehen können, solange das fib-Objekt gültig ist. Es ist also dann nicht nötig zu prüfen, ob das Bild gültig ist und es kann zu jedem fib-Objekt mit einer geeignet definierten Abstandsfunktion immer bestimmt werden, wie nahe/ähnlich das vom fib-Objekt erzeugte Bild an einem Originalbild liegt und damit auch wie "gut" das fib-Objekt ist, bzw. dessen Fitness.

Anmerkung: Bei einigen anderen Darstellungsformen, die durch genetische Algorithmen erzeugt werden, können ungültige Objekte (z.B. Programme) entstehen, bei denen eine genauere Bewertung nicht möglich ist. Eine Population in dieser Darstellungsform kann dann, z.B. eine große Klasse von ungültigen Objekten haben, die alle gleich schlecht sind und damit bei der Selektion gleich berücksichtigt werden.

Ausgangspunkt ist wieder, das ein Bild (euklidisch, zweidimensional, gequantelt) als Matrix dargestellt werden kann. In der die Spalte die x-Koordinate und die Zeile die y-Koordinate des Punktes angibt und die Werte die Farbe des Punktes. Diese Matrix ist endlich mit x-Koordinaten von 0 bis x-max und y-Koordinaten von 0 bis y-max. Die Werte, der einzelnen Punkte der Matrix für die Farben, können nur Werte aus einem endlichen Wertebereich annehmen und jeder dieser Werte wird als Farbe interpretiert. Weiterhin muss eine Hintergrundfarbe festgelegt werden, die jedem Punkt der Matrix, dem durch das fib-Objekt keine Farbe zugeordnet wurde, eine Farbe zuordnet. Damit ist eine "leere" Matrix eine gültige Matrix, die als ein Bild dargestellt werden kann. Vereinbart wird weiterhin, dass Punkte aus dem fib-Objekt, die außerhalb der Matrix liegen, in ihr nicht dargestellt werden und dass Punkte aus dem fib-Objekt, dessen Farbwert außerhalb des Wertebereichs für Farben in der Matrix liegen, entweder als der Farbwert interpretiert wird, der ihnen am nächsten liegt und im Wertebereich liegt, oder als ein anderer fester Farbwert aus dem Wertebereich.

Punkte: Mit diesen Festlegungen liegt jeder beliebige fib-Punkt entweder innerhalb oder außerhalb der Matrix. Wenn er außerhalb liegt, wird er nicht in die Matrix eingefügt, es ändert sich durch ihn die Matrix nicht und sie stellt weiterhin ein Bild dar, wenn sie vorher ein Bild dargestellt hat. Wenn ein fib-Punkt innerhalb der Matrix liegt, wird er eingefügt. Da sein Farbwert als ein gültiger Farbwert interpretiert

wird, wird an einer Koordinate in der Matrix ein gültiger Farbwert gesetzt und die Matrix stellt weiterhin ein Bild dar. Da von einer leeren Matrix ausgegangen wird, die ein Bild darstellt, dann nach und nach die fib-Punkte einzeln eingefügt (oder nicht) werden, wobei kein einzelnes Einfügen dieser fib-Punkte die Matrix ungültig machen kann, so dass sie kein Bild mehr darstellt, stellt die Matrix auch am Ende, nach dem Einfügen beliebig vieler fib-Punkte, ein Bild dar.

Conc Objekte, Bereichsobjekte und Funktionsobjekte werden an sich nicht in der Matrix eingefügt und können fib-Punkte nicht so verändern, dass sie ungültig werden. Sie können nur mehr fib-Punkte zu einem fib-Objekt zusammenfassen (conc und Bereichsobjekte) oder Werte in diesen ändern (Bereichsobjekte und Funktionsobjekte). Damit können sie die Matrix auch nicht direkt oder indirekt so verändern, dass diese kein Bild mehr darstellt.

Kapitel 9

Die genetischen Operationen auf fib

Im Nachfolgenden werden die einzelnen fib-Objekte als Individuen bezeichnet. Die Menge aller Individuen, die zu einem Zeitpunkt im Algorithmus vorhanden sind, wird kurz als Population bezeichnet.

Alle Operatoren lassen auch meist eine große Freiheit bei ihrer Realisierung und der Parameter mit denen sie arbeiten. Da aber die einzelnen Möglichkeiten, in der zur Verfügung stehenden Zeit, nicht alle untersucht werden können, werden die zu treffenden Entscheidungen vielfach auf Vermutungen und Willkür beruhen.

9.1 Die Fitness eines Individuums

Die Fitness eines Individuums ist dadurch gegeben, inwieweit das Programm dem gewünschtem Originalbild ähnelt. Je mehr das Individuum (der Phänotyp dessen) dem Originalbild ähnelt, um so höher sollte die Fitness sein und um so geringer ist der Fehler den das Individuum für die Darstellung des Originalbildes macht.

Dieser Fehler (und damit die Fitness des Individuums) kann z.B. über die Summe der (quadratischen) Abweichungen (nicht definierte Punkte liefern einen maximalen Fehler) in den Farben zu einem Punkt zwischen dem Originalbild und dem vom Individuum erzeugtem Bild bestimmt werden oder über ein anderes selbstbestimmtes Abstandsmaß.

Wenn die Fitness für einzelne Teilobjekte des Individuums bestimmt wird, kann dies z.B. dadurch realisiert werden, dass in die Rechnung nur der Bereich einbezogen wird, der durch das Teilobjekt überdeckt wird, einen Rand um diesen Bereich noch mit einbezogen wird, oder auch nur das kleinste Quadrat benutzen wird, das dieses Objekt umschließt.

Weiterhin ist es sinnvoll auch die Größe (steigt mit der Anzahl der Elemente) der einzelnen Individuen in der Fitness zu berücksichtigen, um größeren Objekten eine geringere Fitness zu geben als kleineren Objekten, mit dem gleichen Fehler auf dem Originalbild, und die kleineren Objekte so zu bevorzugen. Ein weiterer

Faktor der einbezogen werden kann, ist eine Schätzung über die Zeit, die für das Individuum zum Berechnung eines Bildes (Phänotyp), benötigt wird. Damit kann eventuell sogar die Ausführungsgeschwindigkeit des Algorithmusses erhöht werden.

9.2 Selektion

9.2.1 Ein Programm Löschen

Um die Ressourcen (Arbeitspeicher, Rechenzeit) zu schonen, ist es notwendig die Anzahl der Individuen (fib-Objekte) im Bearbeitungsprozess (die noch am genetischem Algorithmus teilnehmen) zu beschränken. Deshalb müssen Individuen aus diesem nach Bedarf entfernt werden. Dabei sind Individuen mit einer niedrigen Fitness zu bevorzugen.

Dies kann geschehen, indem nacheinander neue Generationen generiert werden, wobei von der Vorgängergeneration nur eine bestimmte Anzahl von Individuen übernommen wird, oder das kontinuierlich, immer nachdem eine bestimmte Anzahl (z.B. eins) von Individuen erzeugt wurde, welche aus der Population gelöscht werden.

Es ist auch möglich einige Individuen als unsterblich, bzw. nicht löschbar, zu deklarieren, indem z.B. die n ($n > 0$) besten Individuen als unsterblich deklariert werden, kann vermieden werden, dass diese gelöscht werden und somit eins der besten Individuen verloren geht.

9.3 Vermehrung

9.3.1 Erzeugung eines Individuums

Bei der ”Vermehrung“ wird im allgemeinen ein Individuum aus der Population genommen, bevorzugt Individuen mit hoher Fitness, dieses kopiert und eine der nachfolgenden ”Mutationsoperationen“ auf dieses angewandt. Ist das entstehende Individuum ein ”neues“ Objekt (es kein gleiches in der Population gibt), wird es zur Menge hinzugenommen. Es können auch bereits vorhandene Individuen übernommen werden.

Wenn die Mutationsoperationen fehlschlägt, kann eine andere Mutationsoperationen ausgewählt werden.

9.4 Ein Teilobjekt einfügen (Genimport)

Um zu gewährleisten, dass sich das kopierte Individuum an das Originalbild ”anpassen“ kann, können in dieses neue Teilobjekte aus anderen Individuen einfügt werden.

Dies kann realisiert werden, indem an einer ausgewählten Stelle, an der ein Objekt steht, im Individuum ein neuer conc Zweig einfügt wird. Wobei ein Zweig das Objekt, das an dieser Stelle stand, weiterführt und das andere ein neues Objekt enthält, welches z.B. dadurch gebildet wird, dass ein neuer Punkt mit zufällig gewählten Werten einfügt wird oder ein Teilobjekt, aus einem anderen Individuum, hineinkopiert wird. Auswahlkriterien für Teilobjekte können beliebig spezifizieren werden.

9.5 Mutation

Im Nachfolgenden sind einige einfache mögliche genetische Operatoren aufgelistet, die realisiert werden sollen. Für weitere Operatoren sind der Fantasie natürlich keine Grenzen gesetzt.

9.5.1 Löschung eines Teilobjektes

Um einzelne Individuen möglichst klein zu halten, damit sie nicht über alle Grenzen wachsen ohne den Fehler zum Originalbild zu verkleinern, ist es sinnvoll ein neues Individuum zu erzeugen, indem das Originalindividuum kopiert und aus dieser Kopie ein einzelnes Teilobjekt gelöscht wird. Dabei sind Teilobjekte mit niedriger Fitness zu bevorzugen. Wenn das neu entstandene Individuum eine höhere Fitness als das Alte hat, erhält es auch höhere "Überlebenschancen“ und das alte Individuum wird eher gelöscht.

Ein Objekt wird dadurch gelöscht, indem die conc Verzweigung gelöscht wird, in der es vorkommt, und nur das andere Objekt aus dieser conc Verzweigung an der Stelle einfügt wird, an der zuvor das conc Objekt stand.

9.5.2 Ändern eines Wertes

Bei dieser Operation wird aus dem kopierten Individuum ein Wert ausgewählt und dieser entweder verändert, indem z.B. ein zufälliger Wert hinzu addiert wird, oder für ihn eine Variable eingefügt wird.

9.5.3 Änderung einer Variablen

Das Ändern einer Variablen erfolgt analog zum Ändern eines Wertes. Die Variable kann entweder durch einen Wert, der z.B. zufällig bestimmt wird oder im Wertebereich der Variablen liegt, oder durch das Einfügen einer anderen Variablen ersetzt werden.

9.5.4 Einfügen einer Variablen

Beim Einfügen einer Variablen, wird entweder eine schon unter dem Objekt (in den Elementen die dieses enthalten) existierende Variable eingefügt oder eine neue Variable gebildet.

Beim Bilden einer neuen Variablen wird ein Element, das die Variable definiert, in der Kopie an einer ausgewählten Stelle einfügt und zwar oberhalb der Stelle an der seine Variable eingefügt wurde. So dass es das Objekt enthält, in dem seine Variable eingefügt wurde.

Es können Funktions- und Bereichsobjekte eingefügt werden.

Beim Einfügen einer Funktion ist es sinnvoll, sie so zu bilden, dass sie den gleichen Wert repräsentiert, der ersetzt wurde. Dies kann z.B. dadurch erreicht werden, dass die Funktion nur eine Teilfunktion mit dem Vektor (Wert,1,x) oder (1,Wert,1) enthält.

Beim Einfügen eines Bereichsobjekts ist es sinnvoll, dass der Bereich, der damit überdeckt wird, (nur) den Wert, der ersetzt wird, überdeckt. Dies kann z.B. dadurch erreicht werden, dass dieses Bereichsobjekt nur ein Bereich mit dem Vektor (Wert, Wert) oder ($Wert - x, Wert + y$) enthält.

9.5.5 Löschung eines Terms

Funktions- und Bereichselemente können gelöscht werden, wenn kein Objekt unter ihnen mehr ihre Variable, die sie definieren, enthält. Beim Anwenden dieser Operation, ist es sinnvoll, mehrmals ein Funktions- oder Bereichselement aus dem Individuum herauszugreifen um zu versuchen dieses zu entfernen.

9.5.6 Hinzufügen eines Vektors zu einer Liste

Listen, die in Funktions- und Bereichselementen enthaltenen sind, können durch neue Vektoren erweitert werden.

Um einen Gradientenanstieg leichter zu ermöglichen, ist es sinnvoll die Vektoren zum Erweitern sinnvoll zu wählen. Bei Funktionen ist es z.B. sinnvoll nur Vektoren der Form (1; 1; 1) einzufügen, da diese nur eine kleine Verschiebung der Funktion realisieren. Bei Bereichsoperatoren ist es z.B. sinnvoll nur Vektoren einzufügen, die in der Nähe der schon vorhandenen Bereiche liegen, z.B. vorher [(1; 5)] nachher [(1; 5),(7; 8)], da so die Bereiche die "Umgebung ertastend" erweitert werden.

9.5.7 Löschen eines Vektors aus einer Liste

Vektoren, die in Listen von Funktions- und Bereichselementen enthalten sind, können gelöscht werden. Bei Funktionselementen ist es sinnvoll, eventuell eher "unwichtig" erscheinende Vektoren zu bevorzugen, z.B. Vektoren der Art (0; x; y), (x; 0; y) oder Teilfunktionen mit niedrigen Funktionswerten. Bei Bereichselementen ist es sinnvoll, Vektoren die aneinander grenzen, überschneiden oder nahe beieinander liegen zusammenzufügen (neuer Vektor z.B. (kleinste untere Grenze beider Vektoren; größte obere Grenze beider Vektoren)) oder Vektoren für kleine Bereiche beim Löschen zu bevorzugen z.B. (6; 6).

9.5.8 Verschieben eines Elements

Einzelne Elemente (außer Punkte) können in der Hierarchie der Elemente des Objekts auch nach oben oder unten verschoben werden.

Dabei muss beachtet werden, dass die Objekte nicht fehlerhaft werden, d.h.:

- es darf kein Funktions- oder Bereichselement nach unten über ein Funktions- oder Bereichselement verschoben werden, das die Variable enthält, die das verschobene Objekt definiert
- es darf kein Funktions- oder Bereichselement nach oben über ein Funktions- oder Bereichselement verschoben werden, das eine Variable enthält, die das verschobene Objekt benötigt
- wenn conc nach unten verschoben wird, werden die Elemente, die es enthält, abwechselnd in das Element über ihn verschoben (die Variablenmenge, die durch Elemente unter conc realisiert werden, sind disjunkt)
- wenn conc nach oben verschoben wird, werden Elemente, über die es wandert, in die Objekte, die es enthält, verschoben in denen die Variable, die diese Elemente definieren, verwendet wird
- wenn ein Funktions- oder Bereichselement nach unten über ein conc Element verschoben wird, muss es in die Objekte kopiert werden, die das conc Object enthält und die Variablen verwenden, die dieses Funktions- oder Bereichselement definiert

9.5.9 Vertauschen der Objekte in conc Objekten

Innerhalb der conc Elemente können die einzelnen Objekte, die es enthält, vertauscht werden. Da es für diese Objekte eine feste Reihenfolge der Auswertung gibt und das Vertauschen somit das produzierte Bild verändern kann, bewirkt diese Operation eventuell etwas.

Kapitel 10

Komplexitätsabschätzung

Hier soll eine Abschätzung der "Komplexität" oder besser des nötigen Aufwands zum Erzeugen eines entsprechenden fib-Objekts zu einem beliebigen Bild vorgenommen werden, um die Realisierbarkeit der Idee bewerten zu können.

Dabei sollen Annahmen (Parameter und das Wirken von Operationen) möglichst vorteilhaft so gewählt werden, wie sie im Teil IV auch realisiert werden können und die Abschätzung möglichst einfach wird. Wenn also die Annahmen, so wie in der Abschätzung im Teil IV gewählt werden, soll der nötige Aufwand, durchschnittlich über unendlich vielen Experimenten fast sicher (wegen den Zufallsauswahlen), maximal unter dem der Abschätzung liegen.

10.1 Abschätzung

10.1.1 Annahmen

Es soll ein Bild mit 100 mal 100 Bildpunkten, mit einer Farbtiefe von 8 Farben durch ein Individuum realisiert werden.

Die Farbwerte haben den Vektor (r, g, b) mit $r, g, b \in \{0; 1\}$, das sind 8 mögliche Farben, da $2^3 = 8$.

Individuen bestehen nur aus Punkten (die keine Variablen enthalten) und conc Objekten. Damit ist auch keine größere Komprimierung möglich.

Individuen haben keine Hintergrundfarbe. Punkte des Bildes, die keine Punkte im Individuum haben, werden mit dem maximalen Fehler zum gesamtem Fehler addiert.

Der Fehler eines Individuums ist die Summe der Abweichungen der Punkte, des Bildes das es darstellt, von den Punkten an den entsprechenden Positionen im Originalbild. Um so höher der Fehler ist, um so geringer ist die Fitness des Individuums. Der Fehler eines Teilobjekts ist die Summe der Abweichungen seiner Punkte (Punktobjekte) von den Punkten an den entsprechenden Positionen im Originalbild.

Alle Individuen bestehen anfangs nur aus Punkten, die mit völlig zufälligen Positions- und Farbvektoren initialisiert sind.

Die Menge der Individuen enthält 100 Individuen.

Es gibt ein unsterbliches (nicht löschbares) Individuum, das ist das Individuum mit der höchsten Fitness.

Die einzigen genetischen Operationen sind die Veränderung eines Wertes und die Verbindung zweier Objekte mit Hilfe von conc.

Die Wahrscheinlichkeit das die Operation zum Ändern eines Wertes bei einer Kopie eines Individuums angewand wird, ist gleich 0,9.

Bei der Operation für das Ändern eines Wertes, werden alle Werte eines Punktes im neuen Individuum auf vollkommen neue zufällig erzeugte Werte gesetzt, es wird also eigentlich ein neuer Punkt mit zufälligen Werten erzeugt.

Die Wahrscheinlichkeit, dass bei einer Kopie eines Individuums ein Objekt mit Hilfe von conc eingefügt wird, ist gleich 0,1.

Dabei kann nur ein Objekt, das keine gemeinsamen Teilobjekte mit dem Individuum besitzen, eingefügt werden. Beide müssen dafür weiterhin einen Fehler von 0 für ihre Teilobjekte haben, bzw. es wird als Individuum ein Individuum mit dem geringsten Fehler auf dem gesamten Bild und ein (Teil-) Objekt mit dem Fehler 0 seiner Teilobjekte, bei der Verbindung mit Hilfe von conc ausgewählt. Wenn ein Objekt und Individuum mit den geforderten Eigenschaften in der Menge existieren, wird die conc Operation auf diesen ausgeführt (Suche nach richtigen Objekten).

Um die Wahrscheinlichkeiten zu realisieren, werden die Operatoren ("ändern eines Wertes“ und "mit conc Verbindungen schaffen“) immer in der gleichen Reihenfolge nacheinander ausgeführt. Erst 9 mal Wert ändern und dann 1 mal 2 Objekte versuchen mit Hilfe von conc zu verbinden. Es gibt dabei also keinen Zufall.

Das Löschen von Individuen, um die Menge dieser wieder auf 100 Individuen nach der Vermehrung bei der Mutation, zu reduzieren, erfolgt direkt nach dem Ausführen der conc Operation. Es wird ein Individuum der Population mit dem höchsten Fehler, bzw. geringsten Fitness, gelöscht.

Kombinationsmöglichkeiten: Die möglichen Kombinationen mit 8 Farben und 100 mal 100 Bildpunkten sind $8 * 100 * 100 = 80000$

Es soll die Anzahl von Operationsanwendungen die durchschnittlich gebraucht werden, um ein Bild mit einer bestimmten Wahrscheinlichkeit zu erzeugen, bestimmt werden.

10.1.2 Die Sammelmaschine

Die Idee ist, dass alle korrekten Punkte in einem Individuum gesammelt werden.

Warum?

Die conc Operation verbindet nur zwei Objekte, die einen Teilobjektfehler von 0 haben. Also sind die Punkte, die von der conc Operation zusammenfügt werden, immer genau richtig. Somit sind die Punkte, die das erzeugte Individuum darstellt, auch genau richtig.

Außerdem wird immer nur ein Punkt in das Individuum eingefügt, der noch nicht in ihm vorhanden ist, d.h. nach der conc Anwendung wird ein Punkt mehr vom neuem Individuum richtig codiert.

Da nun dieses neu entstandene Individuum einen Punkt mehr richtig enthält, ist es nun ein Individuum mit einer höheren Fitness, als das Vorgängerindividuum. Bei jeder conc Anwendung entsteht also ein Individuum, das eine höhere Fitness hat als das Vorgängerindividuum, aus dem es entstanden ist. Da das Vorgängerindividuum eines der Individuen mit der höchsten Fitness war, ist das entstandene Individuum, da es eine höhere Fitness hat, dann das Individuum mit der höchsten Fitness und wird für die nächste conc Anwendung als Individuum benutzt. In dieser "Abstammungslinie" (später als Punktekorb bezeichnet) werden sozusagen alle richtigen Punkte gesammelt. Einzige Ausnahme davon ist, wenn auf dieses Objekt die Operation zur Werteänderung ausgeführt wird und im entstehendem Objekt auch alle Punkte richtig sind. Dann wird wieder eins der beiden Individuen ausgewählt und es existiert in jedem Fall ein Punkt der noch nicht im ausgewählten Individuum ist und richtig ist (im anderen Individuum).

Weiterhin bedeutet die Tatsache, dass nur neue richtige Punkte eingefügt werden, das nur maximal so oft der conc Operator angewendet wird, wie das Bild Punkte hat, also maximal 10.000 mal.

10.1.3 Würfel Annahme

Die Operation zum Verändern eines neuen Wertes wirkt schlimmstenfalls wie das zufällige Erzeugen eines der 80.000 möglichen Punkte (ein Würfel mit 80.000 Seiten).

Warum?

Die Operation zum Ändern eines Wertes erzeugt völlig neue Punkte, die sich bis zur Anwendung der conc Operation "gemerkt" werden. Denn erst danach werden wieder Punkte gelöscht.

Zusätzlich besteht die Möglichkeit, dass es außer diesen erzeugten Punkten, es Punkte in der Menge schon gab, die mit Hilfe von conc verwendet werden können.

Damit reduziert sich das Problem auf eine Art Bingoproblem. Man hat eine Liste (die Bildpunkte hintereinander aufgereiht) von Zahlen (10.000 mögliche, entspricht dem Bild mit seinen 10.000 möglichen Punkten) und einen Würfel (mit 80.000 Seiten, ein Wurf ist dann ein möglicher Punkt, der erzeugt werden kann)

und streicht immer nach neuen Würfen eine Zahl von der Liste, wenn eine neue in den Würfen aufgetreten ist.

10.2 Rechnung

q = Anzahl der noch zu suchenden (verschiedenen) Bildpunkte (vorteilhafte Ereignisse)
p = Anzahl der Punkte (Pixel) die erzeugt werden sollen
k = Anzahl der Punkte die erzeugt werden können (mögliche Ereignisse) alle gleichwahrscheinlich (siehe Würfel Annahme)
w = Anzahl der Würfe oder zufällig erzeugten Punkte pro Durchlauf
a_n = Anzahl der Durchläufe für den Versuch einen richtigen Punkt für das Bild zu erzeugen

Die Wahrscheinlichkeit, dass nach einer Operation zum Ändern eines Wertes kein Punkt, der noch zu suchenden Punkte des Bildes, erzeugt wurde ist:

$$P_{W1} = (1 - \frac{q}{k})$$

Die Wahrscheinlichkeit, dass nach w Wert Operationen zum Ändern eines Wertes kein Punkt, der noch zu suchenden Punkte des Bildes, erzeugt wurde ist:

$$P_W = (1 - \frac{q}{k})^w$$

Damit ist die Wahrscheinlichkeit, dass nach w Operationen zum Ändern eines Wertes mindestens ein Punkt der noch zu suchenden Punkte des Bildes erzeugt wurde und damit die conc Operation einen neuen Punkt zur Verfügung hat, der noch nicht im Punktekorb ist und somit durch die conc Operation dann hinzugefügt werden kann:

$$P_C = 1 - (1 - \frac{q}{k})^w$$

Die Wahrscheinlichkeit, das nach w mal a_n Operationen zum Ändern eines Wertes kein Punkt, der noch zu suchenden Punkte des Bildes, erzeugt wurde, ist:

$$P_W = (1 - \frac{q}{k})^{w * a_n}$$

Damit ist die Wahrscheinlichkeit, dass nach w mal a_n Operationen zum Ändern eines Wertes mindestens ein Punkt der noch zu suchenden Punkte des Bildes erzeugt wurde und damit die conc Operation einen neuen Punkt zur Verfügung hat, der noch nicht im Punktekorb ist und durch die conc Operation dann hinzugefügt werden kann:

$$P_C = 1 - (1 - \frac{q}{k})^{w * a_n}$$

Daraus folgt die Wahrscheinlichkeit alle Punkte des Bildes (das Originalbild) richtig zu finden ist:

$$P = \prod_{n=1}^{p} (1 - (1 - \frac{n}{k})^{w*a_n})$$

n steht hier für das aktuelle q.
(Es sind anfänglich 10.000 noch nicht gefundene Punkte zu suchen und dann mit jedem Durchlauf einer weniger, bis nur noch Einer fehlt.)
Die Anzahl der gesamten benötigen Operationsanwendungen ist:

$$o_{ges} = \sum_{n=1}^{\infty} (a_n * (w + 1))$$

Wenn a_n um 1 erhöht wird, werden $w + 1$ Operationen (w mal Wert ändern und 1 mal conc) ausgeführt.

Die Wahrscheinlichkeit mit der das Bild codiert werden soll, soll mindestens e sein, damit ist

$$e \leq \prod_{n=1}^{p} (1 - (1 - \frac{n}{k})^{w*a_n})$$

$$x_n = (1 - (1 - \frac{n}{k})^{w*a_n}) \leq 1$$

Um die Folge der a_n zu ermitteln wird die Vereinfachung gemacht, dass das mehrfache Produkt P ($\prod_{n=1}^{p} x_n$) durch den Exponenten (x_n^p) ersetzt wird. Dieser Austausch ist möglich, da damit in der Formel gefordert wird, das jedes x_n größer als $e^{(1/p)}$ ist und wenn das mehrfache Produkt genommen wird ($\prod_{m=1}^{p} x_n$), dieses größer als e ist. Es ist eine obere Abschätzung von P, da es mit der Produktformel noch Folgen von a_n geben kann, in der einzelne $(x_n)^p$ kleiner als e sind, aber ihr Produkt wieder größer, nach dieser Vereinfachung sind aber alle $(x_n)^p$ größer als e und damit werden nach der Vereinfachung einzelne a_n nur größer und niemals kleiner.

$$e \leq (1 - (1 - \frac{n}{k})^{w*a_n})^p \quad (1 \leq n \leq p) \mid \log x \tag{10.1}$$

$$\rightarrow \log(e) \leq \log(1 - (1 - \frac{n}{k})^{w*a_n}) * p \quad \mid : p\ (da\ p > 0) \tag{10.2}$$

$$\rightarrow \log(e) * 1/p \leq \log(1 - (1 - \frac{n}{k})^{w*a_n}) \quad \mid e^x \tag{10.3}$$

$$\rightarrow e^{1/p} \leq 1 - (1 - \frac{n}{k})^{w*a_n} \quad \mid -1 \tag{10.4}$$

$$\rightarrow e^{1/p} - 1 \leq -(1 - \frac{n}{k})^{w*a_n} \quad \mid *(-1) \tag{10.5}$$

$$\rightarrow 1 - e^{1/p} \geq (1 - \frac{n}{k})^{w*a_n} \quad \mid \log x \tag{10.6}$$

$$\rightarrow \log(1 - e^{1/p}) \geq \log(1 - \frac{n}{k}) * w * a_n \quad \mid : \log(1 - \frac{n}{k}) \tag{10.7}$$

(da $0 \leq (1 - \frac{n}{k}) < 1$ (da $n \leq k$ und $n, k > 0$) ist $\log{(1 - \frac{n}{k})} < 0$ und das größer gleich Zeichen wird zu einem kleiner gleich Zeichen, Division durch eine negative Zahl)

$$\rightarrow \frac{\log{(1 - e^{1/p})}}{\log{(1 - \frac{n}{k})}} \leq w * a_n \qquad | : w \ (w > 0) \tag{10.8}$$

$$\rightarrow \frac{\log{(1 - e^{1/p})}}{\log{(1 - \frac{n}{k})} * w} \leq a_n \tag{10.9}$$

für $a_n \geq x$ ist $a_n = \lceil x \rceil$ eine obere Abschätzung($\lceil x \rceil = x$ aufgerundet)

$$\rightarrow a_n = \left\lceil \frac{\log{(1 - e^{1/p})}}{\log{(1 - \frac{n}{k})} * w} \right\rceil \tag{10.10}$$

$$\text{und} \qquad o_{ges} = \sum_{n=1}^{p} \left(\left\lceil \frac{\log{(1 - e^{1/p})}}{\log{(1 - \frac{n}{k})} * w} \right\rceil * (w + 1) \right) \tag{10.11}$$

Zur Berechnung wurde in C++ ein Programm geschrieben und mit einigen Werten für e die Anzahl der maximal nötigen Operationsanwendungen ermittelt.

10.3 Ergebnis

Die erhaltenen Werte, für ein Bild mit 10.000 Pixel (p = 10.000) und 8 Farben (k = 80.000) und 9 mal die Operation zum Ändern eines Wertes (w = 9), sind in Tabelle 10.1 zu sehen.

Wahrscheinlichkeit mit der das Bild vollständig berechnet werden soll e	Durchschnittlich maximal benötigte Operationsanwendungen
0,1	7.291.420
0,5	8.329.130
0,9	9.956.210
0,99	11.987.520
0,999	13.981.600

Tabelle 10.1: Abschätzung der nötigen Operatorenanwendungen

Die berechneten Werte sind im Bereich des Machbaren.

Der eigentliche Algorithmus wird allerdings meist anders arbeiten, als hier beschrieben, deshalb sind durchaus größere Werte für die Anzahl der Operatorenanwendungen möglich.

Kapitel 11

Weitere Annahmen zu fib

11.1 Komprimierungsmöglichkeiten vom fib

Warum ist eine effizientere Codierung als beim ”Pixelbild“ möglich?
Mit Hilfe von fib lassen sich zumindest einfache (strukturierte) Bilder mit fib-Objekten kürzer darstellen.

Beispiel:
RGB Farben ((r,g,b) Vektor) Bild mit 1.000 Pixel horizontal und vertikal mit weißem Hintergrund und einem schwarzen Strich

Eine fib-Darstellung: conc(for(y,[(0;1000)],for(x,[(0;1000)], p((x,y), (255;255;255)))), for(x,[(5;500)], p((x,6), (0;0;0)))) Länge dieses fib-Objektes: 104 Bytes (Anzahl der Zeichen im String oberhalb)

Bitmapbild: 3 Bytes pro Punkt * 1.000 * 1.000 = 3.000.000 Bytes

An diesem Beispiel ist ersichtlich, dass das fib-Objekt eine wesentlich kürzere Darstellung ermöglicht. Zumindests für dieses Beispiel ist eine gute Komprimierung mit fib möglich (Faktor von rund 30.000 zu Bitmapbildern).

Eine Aussage im Allgemeinen über die Komprimierungsmöglichkeiten von fib, kann von mir aber leider wegen der möglichen großen Komplexität von fib-Objekten nicht getroffen werden. Es ist aber zu erwarten das die Komprimierungsmöglichkeiten mit Verringerung der Komplexität der Bilder, bzw. der Objekte auf diesen, zunimmt.

11.2 Überdeckung von fib-Objekten und Dichte der fib-Objekte im Hypothesenraum

Definiert wird eine Überdeckungsrelation $<_{fib}$ auf zwei korrekten fib-Objekten.

Dabei gilt: for(x,[..., (a;b),...], obj1) $<_{fib}$ conc(for(x,[...,(a;c), ...], obj1), for(x,[...,(e;b), ...],obj1)), dabei entspricht der Bereich (a;b) der Wertemenge [a, ..., c, e, ..., b].

Außerdem gilt for(x,[(a;b)],obj1) $<_{fib}$ conc(obj1', obj1"), dabei entspricht der Bereich (a;b) der Wertemenge [a,b] und im obj1' wurden alle Vorkommen von x durch a ersetzt und in obj1" alle Vorkommen von x durch b.

Es gilt weiterhin, wenn zwei Objekte (obj1 und obj2) je ein Objekt enthalten (obj_e1 ist enthalten in obj1 und obj_e2 ist enthalten in obj2), welche die Relation erfüllen (Beispiel 1: obj_e1 $<_{fib}$ obj_e2) und der Rest dieser Objekte (obj1,obj2), ohne die enthaltenden Objekte (obj_e1, obj_e2), identisch ist, dann erfüllen auch die beiden Objekte die Relation, in der gleichen Richtung (für das Beispiel 1: obj1 $<_{fib}$ obj2).

Zwei Objekte, welche die Relation $<_{fib}$ erfüllen, realisieren das gleiche Bild.

Jedem Objekt (obj) ist eine Klasse von Objekten (obj') zugeordnet, welche die Gleichung obj $<_{fib}$ obj' erfüllen.

Wenn nun angenommen wird, dass "natürliche" Bilder, die etwas darstellen sollen, nicht aus völlig unzusammenhängenden Pixeln bestehen, sondern Zusammenhänge zwischen den Pixeln enthalten sind und diese mit entsprechend vielen for und fkt Objekten dargestellt werden können. So gehören zu solchen Bildern viele fib-Objekte, mit relativ vielen for Objekten, die eine noch größere Menge von fib-Objekten überdecken, welche die $<_{fib}$ Relation erfüllen und damit zu den entsprechenden Klassen gehören.

Die daraus gefolgerte Vermutung ist nun, dass im Hypothesenraum der fib-Objekte, die fib-Objekte die "natürlicheren" Bildern entsprechen, dichter gestreut sind, als die fib-Objekte die Bilder mit völlig unzusammenhängenden Pixel entsprechen. Und damit, dass bei der Wanderung durch den Hypothesenraum "natürliche" Bilder häufiger angetroffen werden.

Im allgemeinen kann vermutet werden, dass je mehr Möglichkeiten es gibt ein Bild darzustellen, um so wahrscheinlicher ist es, eine Lösung zu finden.

Zur Dichte des Hypothesenraumes kann noch weiterhin folgendes gesagt werden:

Jedes Individuum symbolisiert ein Bild (siehe 8.3.3). Das heißt: Wenn es n mögliche Bilder gibt (z.B. $n = 100 * 100$ Bildpunkte $*8$ Farben$= 80.000$ Möglichkeiten) gibt es auch nur n Individuengruppen die unterschiedliche Bilder symbolisieren. Wenn jedem Bild eine Klasse von fib-Objekten, die dieses repräsentieren, zugeordnet wird, so gibt es im ganzen unendlichen Hypothesenraum nur endlich (n) viele verschiedene solcher Klassen. Womit das Finden eines fib-Objektes, das ein bestimmtes Bild repräsentiert, sich nicht mehr als ganz so unmöglich darstellt.

11.3 Annahme über verdeckte Objekte

Annahme: Verdeckte Objekte dienen als Ressource für neue Individuen.

Im Verdeckten Bildbereich können sich neue Teilobjekte bzw. Teilbilder entwickeln, die sich nicht negativ auf die Ähnlichkeit zum Originalbild auswirken. Wenn ”Superindividuen“ vorhanden sind (starkes lokales Optimum), die, wegen ihrer Ähnlichkeit zum Originalbild, die ganze Population dominieren, können sie das Bilden von neuen, weniger ähnlichen Individuen unterdrücken. Trotzdem können gleichähnliche Individuen entstehen, die verdeckte Teilobjekte bzw. Teilbilder enthalten, welche die Überwindung des aktuellen lokalen Optimums ermöglichen.

Dies sollte ”Monokulturen“ entgegenwirken.

Dieser Effekt hat eventuell Ähnlichkeit mit dem Effekt, den eine veränderbare Populationsgröße bei genetischen Algorithmen hat, nur wird nicht die Zahl der Individuen verändert, sondern die Anzahl der ”Teilgene“ dieser Individuen.

Kapitel 12

Parallelen zur natürlichen Evolution

Als Parallele seien hier die Bakterienstämme (z.B. E. coli) aufgeführt. Sie besitzen Gene, auf Grund derer bei ihnen genetische Evolution auftritt.

Bei diesen Bakterienstämmen (z.B. E. coli) kann es auch zu einem Gentransfer kommen, dabei wird ein Teil der Geninformationen (des Genmaterials oder dessen Kopien) von einem Bakterium zu einem anderem übertragen. Daneben gibt es natürlich auch Mutationsprozesse.

Ein fib-Objekt kann als Information betrachtet werden, die ein Bild codiert, so wie Bakteriengene ein Bakterium codieren (Aufbau und Verhalten).

Die einzelnen fib-Elemente sind dabei weniger als die Basen der Gene zu sehen, sondern vielmehr als die Funktionalität von Genen oder Genmengen. So wie bei den Bakterien durch Kombination der Gene, bzw. der Dinge die sie codieren z.B. Enzyme, erst komplexere Funktionalitäten definiert werden (z.B. Umsetzung von Zucker in Bewegungsenergie), entstehen bei fib-Objekten erst durch Kombination der Elemente (Teil-)Bilder.

Fib-Teilobjekte können auch, wie beim Gentransfer bei den Bakterien, in andere fib-Objekte übertragen werden und unterliegen einer Mutation. Wobei auch bei den Bakterien schwer zu sagen ist, ob die Mutation der Gene wirklich völlig zufällig ist oder ob im Laufe der Jahrmillionen nicht Mechanismen entstanden sind, die zu einer ”intelligenteren“ Mutation führen. Einzelne fib-Objekte können dabei nicht nur als einzelne Bakterien angesehen werden, sondern auch als alle Bakterien mit identischen Geninformationen.

Für weitere Informationen zur Genetik bei Bakterien siehe ”Einführung in die Mikrobiologie“([1])

Teil IV

Implementierung der Idee

Kapitel 13

Einleitung für die Implementierung

In diesem Teil soll ein Programmsystem entworfen und realisiert werden, das einen genetischen Algorithmus mit der in Teil III beschriebenen Bildbeschreibungssprache fib realisiert.

Da es sich bei dem Projekt um ein experimentelles System handeln soll, muss bei der Implementierung darauf geachtet werden, dass Erweiterungen und Änderungen in ihr möglichst leicht vorzunehmen sind. Besonders ist darauf bei der Klasse zu achten, die den eigentlichen genetischen Algorithmus mit den Operationen realisiert. Es sollte schon von vornherein eingeplant werden, dass weitere Operationen hinzukommen.

Kapitel 14

Entwurf des Programmsystems

Der Entwurf der Bildbeschreibungssprache fib und der genetischen Operatoren geschah bereits in Teil III, deshalb wird hier nur noch der Entwurf der Benutzerschnittstelle besprochen.

14.1 Entwurf der Benutzerschnittstelle

Eine wirkliche Benutzerschnittstelle soll noch nicht realisiert werden, sondern es wird zunächst mehr Wert darauf gelegt den genetischen Algorithmus zum Laufen zu bringen.

Die Steuerung des Algorithmus geschieht über die Parameter, die in einer Parameterklasse zusammengefasst werden. Das macht die Parameter besser handhabbar, z.B. in Bezug auf Persistenzhaltung und Parameterübergaben. Die Parameterklasse sollte so allgemein wie möglich gehalten werden und Platz für mehr Parameter lassen, als anfänglich benötigt, damit später neue Parameter problemlos integriert werden können.

Der Algorithmus sollte über die Möglichkeit verfügen, über Parameter die aktuelle Daten ausgeben zu können.

Daten des Algorithmus die auf jeden Fall ausgebar sein sollten, sind:

- die Zahl der aktuellen Iteration, um einen Fortschritt beobachten zu können
- das letzte beste Individuum, bzw. der Verlauf der besten Individuen, um ein Ergebnis zu haben

Weiterhin sollen auch die Parameter eines Parameter Objektes ausgegeben werden können.

Kapitel 15

Realisierung des Programmsystems

Die Realisierung des Programmsystems soll in C++ geschehen. Diese Sprache ist relativ verbreitet und gewährleistet damit eine relativ einfache Portierung auf andere Plattformen und vereinfacht eventuelle Überarbeitungen anderer Entwickler. Weiterhin unterstützt C++, durch ihre Objektorientiertheit, die Objektesicht von fib und gewährleistet eine relativ hohe Performance, die benötigt wird, da Operatoren sehr oft ausgeführt werden sollen.

15.1 Namenskonventionen

Alle Namen entstammen der englischen Sprache oder werden von dieser abgeleitet. Dadurch ist der Quelltext für einen weiten Bereich von Nutzern zugänglich. Die Namen wurden so gewählt, dass Rückschlüsse auf ihre Funktion möglich sind. Bei der Namensgebung habe ich versucht, mich an die folgenden Konventionen zu halten:

- Namen beginnen mit einem Großbuchstaben, es folgen Kleinbuchstaben
- wenn Namen aus elementareren Namen zusammengesetzt werden, dann beginnt jeder Namensbestandteil mit einem Großbuchstaben
- Namen von Datenobjekten, die innerhalb von Funktionen/Prozeduren oder nur kurzzeitig benötigt werden (z.B. Laufvariablen, Hilfsvariablen), werden beliebig gebildet und dann klein geschrieben
- Namen von Konstanten werden durchgängig groß geschrieben
- Namen von Nichtstandardtypen erhalten das Präfixsymbol T
- jede Funktion wird durch einen erweiterten Kopfkommentar erläutert, dieser enthält im allgemeinen eine Kurzbeschreibung, eine Vor- (pre:) und eine Nachbedingung (post:)

- jede Klasse besteht aus zwei Dateien der Headerdatei oder Interface gekennzeichnet mit <Klassennamen.h> und der Implementation gekennzeichnet mit <Klassennamen.cpp>

15.2 Realisierung der Bildbeschreibungssprache fib

Fib unterstützt das objektorientierte Paradigma von C++.

Jedes fib-Element wird als eine Klasse realisiert, die von einer Basisklasse GraphicObject erbt.

15.2.1 Ordnungen

Um auf einzelne Komponenten eines Individuums zugreifen zu können, müssen auf den Individuen entsprechende Ordnungen definiert werden. Alle nachfolgenden Ordnungen werden auf die Ordnung der natürlichen Zahlen abgebildet.

Solche Ordnungen werden benötigt für:

Objektpunkte: Auf allen Elementen ist eine Ordnung zu definieren. Diese kann dazu dienen, einzelne Elemente zu identifizieren.

Verschiebungspunkte: Auf den Elementen die verschoben werden können, ist eine Ordnung zu definieren.

Elementlöschpunkte: Auf alle Elemente die gelöscht werden können, das sind Funktions- und Bereichselement, muss je eine Ordnung definiert sein.

Vertauschpunkte für conc: Auf den conc Elementen ist eine Ordnung zu definieren, damit ein conc Element ausgewählt werden kann, in dem die enthaltenden Objekte vertauscht werden sollen.

Teilobjektpunkte: Um Teilobjekte zu löschen oder zu kopieren, muss eine Ordnung definiert werden. Für jedes zusammenhängende Teilobjekt, bzw. die in den conc Element enthaltenden Objekte, wird eine Nummer vergeben.

Punktteilobjektpunkte: Um die Berechnung der Bewertungsfunktion möglichtst effektiv zu gestalten, ist es vorteilhaft, zuerst nur für jeden Punkt zu berechnen wie gut er den Bereich den er überdeckt (sein Teilbild) beschreibt und dann diese Bewertungen der Punkteobjekte zu den Bewertungen der Teilobjekte zu kombinieren.

Die einzelnen Werte/Variablen: Um einen Wert oder eine Variable zu ändern, muss aus ihnen ausgesucht werden können. Dazu werden die Werte und Variablen fortlaufend nummeriert.

Listenelemente: Auf allen Elementen die eine Liste enthalten (je eine Ordnung für Funktions- und Bereichselemente), muss eine Ordnung definiert sein, um eine entsprechende Liste auszuwählen und einen neuen Vektor einzufügen oder einen Vektor zu löschen.

Listenvektoren: Auf allen Vektoren in Listen von Funktions- und Bereichselementen wird eine Ordnung definiert.

Ordnung der Objektpunkte O_O

Nur Objekte, die nicht Vektorobjekte sind, sind Objektpunkte.

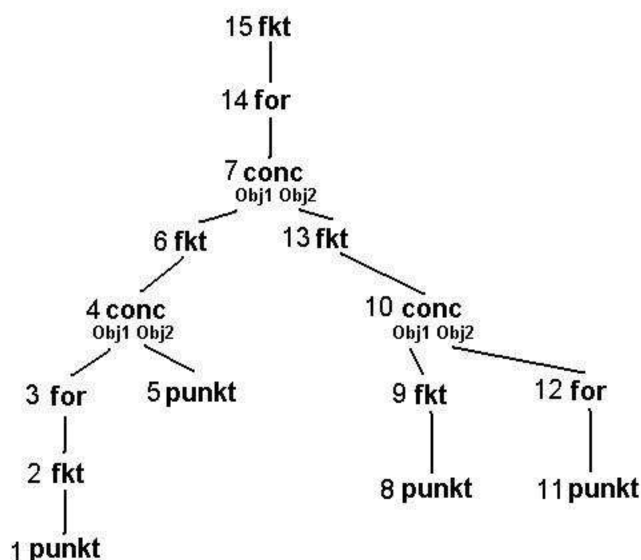

Abbildung 15.1: Ordnung der Objektpunkte

Das Objekt, dem die Zahl 1 zugeordnet wird, ist das Punktobjekt oder der Punkt, der von der Wurzel erreicht wird, indem immer das erste Objekt der conc Objekte ausgewählt wird (diesem "Ast" gefolgt wird).

Das Objekt, das ein Objekt enthält, wird die Nummer des enthaltenden Objektes um eins erhöht zugeordnet, außer eines der Objekte ist ein conc Objekt.

Dem conc Objekt wird eine Zahl zugeordnet, die um eins größer ist, als die Zahl seines ersten Objektes. Damit ist das conc Element sozusagen der Mittelpunkt. Dem Punkt, der erreicht wird, wenn das zweite Objekt eines conc Objektes ausgewählt wird und von, in diesem zweiten Objekt enthaltenden conc Objekten, immer das erste enthaltende Objekt gewählt wird, wird die Zahl des conc Objektes, bei dem das letzte mal das zweite Element ausgewählt wurde, um eins erhöht zugeordnet.

Einem Objekt, das ein conc Objekt enthält, wird die größte Zahl im conc Objekt, um eins erhöht zugeordnet.

Ein Beispiel für diese Ordnung ist im Bild 15.1 wiedergegeben, dies entspricht dem Objekt: fkt(for(conc(fkt(conc(for(fkt(p())),p())),fkt(conc(fkt(p()),for(p())))))))
(Der Einfachheit halber sind nur die Objektnamen aufgeführt.)

Alle anderen Ordnungen orientieren sich an dieser Ordnung.

Ordnung der Verschiebepunkte

Die Ordnung der Verschiebepunkte ist, wie die Ordnung der Objektpunkte, nur das Punktobjekte und conc Objekte keine Verschiebepunkte sind und damit nicht auftauchen.

Die conc Objekte wurden zur Vereinfachung als "kein Verschiebepunkt" klassifiziert.

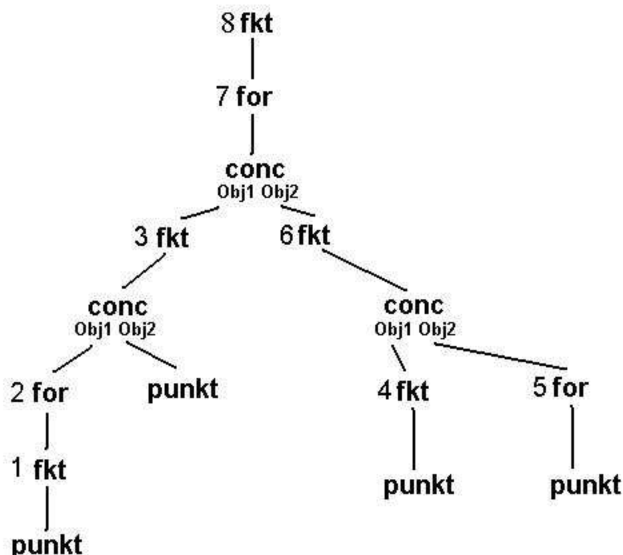

Abbildung 15.2: Ordnung der Verschiebepunkte

Das Objekt, dem die Zahl 1 zugeordnet wird, ist das nächste Objekt welches kein conc Objekt ist, über dem Punktobjekt, dem in der Objektordnung die Zahl 1 zugeordnet ist.

Das Objekt, das ein Objekt enthält, wird die Nummer des enthaltenden Objektes um eins erhöht zugeordnet, außer eines der Objekte ist ein conc Objekt.

Das Objekt, das über dem Punkt steht, der erreicht wird, wenn das zweite Objekt eines conc Objektes ausgewählt wird und dann bei dem, in diesem zweiten Objekt enthaltenden conc Objekten, immer das erste enthaltende Objekt gewählt wird, wird die Zahl des ersten Objektes des conc Objektes, bei dem das letzte mal das zweite Element ausgewählt wurde, um eins erhöht zugeordnet.

Ein Objekt, das ein conc Objekt enthält und selbst keines ist, wird die Zahl des Verschiebeobjektes um eins erhöht zugeordnet, welches das erste gefundene Verschiebeobjekt ist, wenn das conc Objekt von unten nach oben durchsucht wird, dabei wird zuerst immer das in einem eventuellen conc Objekt enthaltende zweite Objekt gewählt und dann das erste, bzw. das Verschiebeobjekt, dem im conc Objekt die höchste Zahl in der Verschiebeordnung zugeordnet ist.

Im Bild 15.2 ist für das Beispiel fib-Objekt aus die Ordnung der Objektpunkte die entsprechende Verschiebeordnung dargestellt.

Die Ordnung der Verschiebepunkte wird weiterhin benutzt für die Ordnung der Elementlöschpunkte.

Ordnung der Vertauschpunkte für conc

Nur conc Objekte sind conc Vertauschpunkte.

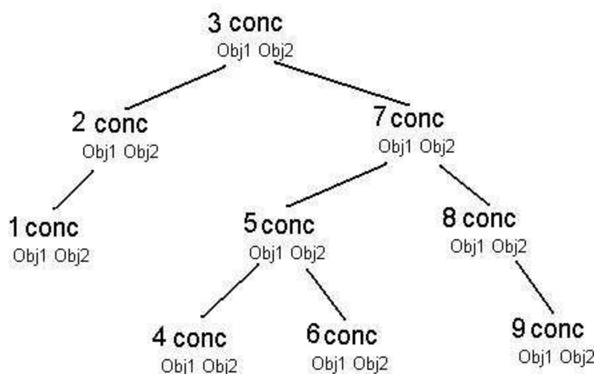

Abbildung 15.3: Ordnung der Vertauschpunkte für conc

Das conc Objekt, dem die Zahl 1 zugeordnet wird, ist das unterste conc Objekt, das von der Wurzel erreicht wird, indem immer das erste Objekt der conc Objekte ausgewählt (gefolgt) wird.

Besitzt das erste enthaltende Objekt des conc Objektes weitere conc Objekte, wird dem conc Objekt die Zahl zugeordnet, die um eins größer ist, als die größte Zahl der enthaltenden conc Objekte, bzw. die Zahl des conc Objektes, welches am

weitesten im Zweig (des ersten enthaltenden Objekt) unten steht und das erreicht wird, wenn immer das zweite Objekt der conc Objekte gewählt wird.

Besitzt das erste Objekt des conc Objektes keine weitere conc Objekte, wird ihm eine Zahl zugeordnet, die um eins größer ist, als die Zahl des conc Objektes, welches als nächstes über ihm steht und in dessen zweitem Objekt er sich befindet.

Ein Beispiel für diese Ordnung ist im Bild 15.3 wiedergegeben, dies entspricht dem Objekt: conc(conc(conc(...),...),conc(conc(conc(...),conc(...)),conc(...,conc(...)))
(Der Einfachheit halber sind nur die conc Elemente aufgeführt.)

Ordnung der Teilobjektpunkte

Die Ordnung der Teilobjektpunkte wird über die Ordnung der Vertauschpunkte für conc definiert. Dabei ist dem ersten Objekt eines conc Objektes die Zahl $x = y * 2 - 1$ zugeordnet und dem zweitem Objekt die Zahl $x = y * 2$, wobei y die Zahl ist, die dem jeweiligem conc Objekt in der Ordnung der Vertauschpunkte für conc zugeordnet ist.

Ordnung der Punktteilobjektpunkte

Jeder Punkt definiert ein Punktteilobjekt.

Das Punktteilobjekt dem die Zahl 1 zugeordnet wird, ist das Teilobjekt das nur das unterste Punktobjekt enthält, welches von der Wurzel erreicht wird, indem immer das erste Objekt der conc Objekte ausgewählt (gefolgt) wird.

Das Punktteilobjekt (P_2) dem die nächstgrößere Zahl, als einem aktuellen Punktteilobjekt (P_1), zugeordnet wird, ist das Punktteilobjekt, das erreicht wird, indem vom aktuellen Punktteilobjekt (P_1) solange aufwärts gewandert wird, bis der Zweig, auf dem gewandert wird, das erste Objekt in einem conc Objekt ist und dann von diesem conc Objekt das zweite Objekt ausgewählt wird und in diesem solange abwärts gewandert wird, wobei immer das erste Objekt in den conc Objekten ausgewählt wird, bis das Punktobjekt (P_2) angetroffen wird.

Ordnung der einzelnen Werte

Der Wert dem die 1 zugeordnet wird, ist der, welcher als erstes gefunden wird, wenn zuerst die Vektoren, des Elementes mit dem kleinsten Objektpunkt aus O_O, ihrer Reihenfolge nach durchsucht werden (der Reihenfolge ihrer Elemente nach) und, falls dort kein Wert gefunden wurde, jeweils die Vektoren, des jeweiligen Elementes mit dem nächst größerem Objektpunkt aus O_O, der Reihenfolge nach durchsucht werden (der Reihenfolge ihrer Elemente nach).

Der Wert (Nachfolgewert), dem die Zahl um eins erhöht zugeordnet ist, als einem anderem Wert (Vorgängerwert), ist der Wert, der gefunden wird, wenn vom Vorgängerwert ausgehend, zuerst die weiteren Elemente des Vektors ihrer Reihenfolge nach durchsucht werden, falls dort kein Wert gefunden wurde, die restlichen

Vektoren des Elements ihrer Reihenfolge nach durchsucht werden (der Reihenfolge ihrer Elemente nach) und, falls dort kein Wert gefunden wurde, jeweils die Vektoren der Reihenfolge nach durchsucht (der Reihenfolge ihrer Elemente nach), des Elementes dem der nächstgrößte Objektpunkt aus O_O zugeordnet ist, wie dem aktuellen Durchsuchtem.

Beispiel:
for(y,[(14;15),(16;17)],fkt(g,[(12;y;13)],conc(for(r,[(5;6)],fkt(x,[(r;1;2);(3;y;4)], p((r;g),(x;y)))),for(d,[(10;11)],p((d;7),(8;9))))))
(Die Zahlen stehen an den entsprechenden Stellen, an denen Werte stehen und haben den Zahlenwert, die den entsprechenden Werten in der Ordnung der Werte zugeordnet sind.)

Ordnung der einzelnen Variablen

Die Zuordnung der Variabelenpunkte ist analog zur Zuordnung der Wertepunkte, nur das nach Variablen gesucht und diese nummeriert werden.

Ordnung der Listenvektoren

Der Listenvektor dem die 1 zugeordnet ist, ist der erste Listenvektor, in der Liste des Objektes mit einer Liste, dem der kleinste Objektpunkt aus O_O zugeordnet ist.

Der Listenvektor (Nachfolgevektor), dem eine Zahl, die um 1 größer ist, als die Zahl eines andern Listenvektors (seines Vorgängervektors), zugeordnet ist, ist der Vektor (Nachfolgevektor), der entweder nach dem Vorgängervektor in der Liste folgt oder der erste Vektor in dem Objekt, welches eine Liste enthält und dem, bezüglich des Objektes, das den Vorgängervektor enthält, die nächstgrößere Zahl zugeordnet ist.

Ordnung der Funktionsobjekte und Bereichsobjekte

Alle, sowohl Funktionsobjekte als auch die Bereichsobjekte, sind in der Reihenfolge der Objektpunkte beginnend mit 1 geordnet.

Die Ordnung der Funktions- bzw. der Bereichsobjekte ist gleichzeitig auch die Ordnung der Listenelement der Funktions- bzw. Bereichsobjekte.

15.2.2 Benötigte Klassen

Folgende Klassen werden benötigt:

- Point für Punktobjekte
- Function für Funktionsobjekte
- Area für Bereichsobjekte
- Conc für conc Objekte

- ListObject realisiert die Methoden, die bei den Klassen Function und Area gemeinsam realisiert werden können, die Function und Area Klassen erben von ihr
- PictureObject ist die Oberklasse der Klassen ListObject, Point und Conc
- Vector realisiert die gemeinsamen Methoden der Vektoren, alle Vektorklassen erben von ihr
- Color für Farbvektoren im Punktobjekt
- Position für Koordinatenvektoren im Punktobjekt
- UnderFunction für die Teilfunktionsvektoren in den Funktionsobjekten
- UnderArea für die Bereichsvektoren in den Bereichsobjekten

15.2.3 Die Klassen im einzelnen

Basisklasse GraphicObject

Stellt alle gemeinsamen Methoden als rein virtuelle Funktionen (pure virtual functions) bereit. Damit auf sie, unabhängig von der Klasse, zugegriffen werden kann.

Die Vektorklassen Vector, Color, Position, UnderFunction und UnderArea

Alle Funktionalitäten, die allen Vektorklassen gemeinsam sind, werden in der Oberklasse Vector realisiert. Welche Methoden der Vektorklassen gemeinsam realisiert werden, wird im Folgenden nicht explizit aufgeführt, sondern folgt implizit daraus, dass die Funktionalität sich bei allen Klassen nicht unterscheidet.

Die Klassen Color, Position, UnderFunction und UnderArea speichern ihre Attribute in Form von Zeigern auf real Werte (`Value`). Zu jedem dieser Zeiger wird weiterhin ein bool Wert (`IsValue`) gespeichert, der angibt, ob es sich bei dem Zeiger um einen Zeiger auf einen Wert handelt (sonst, wenn sie false ist, ist es eine Variable). Jeweils eines dieser Paare (Zeiger, bool Wert) wird zu einem Typ zusammengefasst (`TComponent`) und in einem Array gespeichert, der soviel Komponenten enthält wie der Vektor benötigt.

Die Klassen Point, Function, Area, Conc, ListObject, PicturObject

Die PicturObject Klasse realisiert alle gemeinsamen Methoden der Klassen Point, Function, Area, Conc und ListObject. Methoden die nicht gemeinsam in ihr implementiert werden können, werden als rein virtuelle Funktionen (pure virtual functions) bereitgestellt. ListObject realisiert die Methoden, die bei den Klassen Function und Area gemeinsam realisiert werden können. Die Function und Area Klassen erben von ihr.

Die Klassen Point, Function, Area und Conc stellen die einzelnen Elementtypen der fib-Beschreibungssprache bereit.

Vererbungsgraph der fib-Klassen

Im Bild 15.4 ist der Vererbungsgraph der fib-Klassen dargestellt.

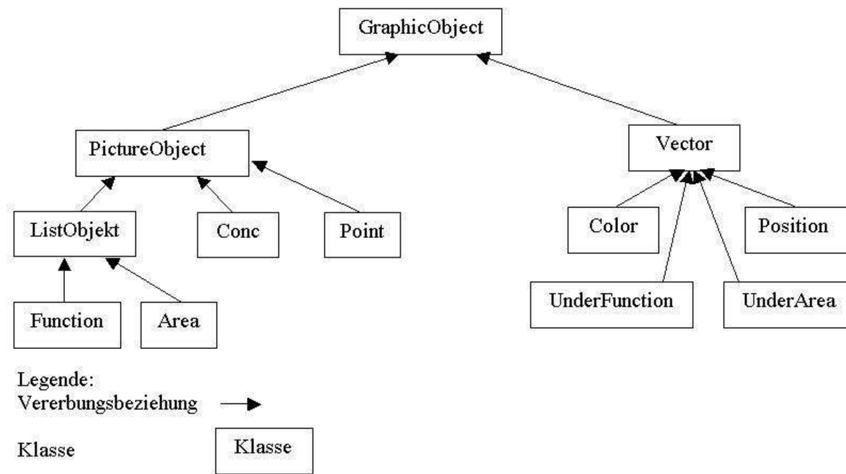

Abbildung 15.4: Vererbungsgraph der fib-Klassen

15.3 Benötigte Methoden der Bildbeschreibungssprache fib

Jedes Objekt muss die Methoden für die Operationen bereitstellen, die auf ihm ausgeführt werden können, inklusive fast aller Operationen, die auf Objekte die es eventuell enthält, angewendet werden können. Da Objekte von einer Klasse, Objekte anderer Klassen allen möglichen Typs enthalten können, muss jede Klasse auch über fast alle Methoden verfügen, die andere Klassen, für Objekte die es enthalten kann, realisieren.

Da es erwünscht ist, dass die Klassen modifiziert werden und eine effiziente Implementierung angestrebt wird, werden die Methoden, die einen kurzen Rumpf enthalten und nicht virtuell (virtual) sind, mit inline im Header realisiert.

Im Nachfolgenden werden die zu realisierenden Methoden aufgeführt. Dabei wird zuerst, wenn vorhanden, die Ein- und Ausgabe der Methode beschrieben. Dann meist eine kurze Beschreibung der Methode, also was sie macht, wofür sie da ist oder was ihre Funktionalität ist. Dann kommt eine umgangssprachliche Beschreibung wie die Funktionalität der Methode skizzenhaft zu realisieren ist, eventuell wenn nötig, separat für einige Klassen oder Klassengruppen. Leider sind einige Methoden relativ komplex, sodass die Verständlichkeit der Beschreibung der Methode darunter leidet.

15.3.1 Gemeinsame Methoden aller Klassen

Konstruktoren

Alle Klasse verfügen über einen Standardconstructor, einen Copyconstructor und einen Parameterconstructor.

Der Standardconstructor initialisiert das Objekt mit Standardwerten.

Der Copyconstructor kopiert das übergebene Objekt.

Der Parameterconstructor erhält als Parameter die Werte mit denen die Variablen des Objektes initialisiert werden sollen. Dabei werden Objekte, auf die übergebene Zeiger zeigen, nicht kopiert und die Vektor Klassen initialisieren alle ihre Komponenten mit 0 und true.

Methoden zur Bestimmung des Wertebereichs der Ordnungen

`unsigned int getNumberOfValue()`

Rückgabewert: Anzahl der Werte im Objekt

Liefert die Anzahl der Werte in einem Objekt zurück. Für jeden Wert der in einem Funktions-, Bereichs- oder Punktobjekt enthaltenden Vektorobjekt enthalten ist, wird der Wert um eins erhöht.

Funktions-, Bereichs- oder Punktobjekte addieren die von ihren Vektorobjekten und eventuell enthaltenden Objekten durch `getNumberOfValue()` ermittelten Werte zusammen und geben die Summe zurück.

Die conc Objekte addiert die Werte, die durch `getNumberOfValue()` der enthaltenden Objekte ermittelt wurden, zusammen und geben die Summe zurück.

Die Vektorobjekte ermitteln die Anzahl der Attribute `IsValue` die true sind und geben sie zurück.

`unsigned int getNumberOfVariable()`

Rückgabewert: Anzahl der Variablen im Objekt

Liefert die Anzahl der Variablen im Objekt zurück. Für jede Variable die in einem Funktions-, Bereichs- oder Punktobjekt enthaltenden Vektorobjekt enthalten ist, wird der Wert um eins erhöht.

Funktions-, Bereichs- und Punktobjekte addieren die Werte der von ihren Vektorobjekten und eventuell enthaltenden Objekte durch `getNumberOfVariable()` ermittelten Werte zusammen und geben die Summe zurück.

Die conc Objekte addieren die Werte, die durch `getNumberOfVariable()` der enthaltenden Objekten ermittelt wurden, zusammen und geben die Summe zurück.

Die Vektorobjekte ermitteln die Anzahl der Attribute `IsValue` die false sind und geben sie zurück.

Andere Abfragemethoden

`int getValue(unsigned int n)`

Rückgabe: der Wert dem in der Ordnung der Werte die Zahl n zugeordnet ist
Eingabe: eine Position n aus der Ordnung der Werte

Liefert den Wert, dem in der Ordnung der Werte die Zahl n zugeordnet ist.

Funktions-, Bereichs- und Punktobjekte ziehen, wenn n größer als x ist, zuerst fürs enthaltende Objekt und dann für jeden Vektor, den sie enthalten (wenn nötig mittels for-Schleife), von n x ab ($n = n - x$), dabei ist x der mittels des Aufrufs der `getNumberOfValue()` (= x) Methode des Objektes oder Vektors ermittelten Wert.

Wenn n kleiner gleich x ist, ruft das Objekt die `getValue(n)` Methode des Objektes oder aktuellen Vektors auf, gibt dessen Zurückgabe zurück und beendet das Durchmustern seiner Vektoren (beendet die for-Schleife). Ist der Wert weder im enthaltenden Objekt noch in einem Vector vorhanden, wird 0 zurückgegeben.

Die conc Objekte rufen die `getNumberOfValue()` (= x) Methode ihres ersten Objektes auf. Ist der ermittelte Wert (x) größer oder gleich dem übergebenen Wert n, so wird dessen `getValue(n)` Methode mit dem übergebenen Wert aufgerufen und deren Rückgabe zurückgegeben. Sonst wird die `getValue(nn)` Methode ihres zweiten Objektes aufgerufen und dessen Rückgabe zurückgegeben, wobei sich der übergebene Wert (nn) aus dem Wert n minus dem ermittelten Wert x ergibt.

Die Vektorobjekte zählen für jedes `IsValue`, welches true ist, den übergebenen Wert um eins herunter, ist n gleich 0 wird von der aktuellen Komponente Component der Wert von `Value` (dereferenzieren) zurückgegeben.

`real* getVariable(unsigned int n)`

Rückgabe: die Variable der in der Ordnung der Variablen die Zahl n zugeordnet ist
Eingabe: eine Position n aus der Ordnung der Variablen

Liefert die Variable, der in der Ordnung der Variablen die Zahl n zugeordnet ist.

Funktions-, Bereichs- und Punktobjekte ziehen, wenn n größer als x ist, zuerst fürs enthaltende Objekt und dann für jeden Vektor, den sie enthalten (wenn nötig mittels for-Schleife), von n x ab ($n = n - x$), dabei ist x der mittels des Aufrufs der `getNumberOfVariable()` (= x) Methode des Objektes oder Vektors ermittelten Wert.

Wenn n kleiner gleich x ist, ruft das Objekt die `getVariable(n)` Methode des Objektes oder aktuellen Vektors auf, gibt dessen Zurückgabe zurück und beendet das Durchmustern seiner Vektoren (beendet die for-Schleife). Ist die Variable weder im enthaltenden Objekt noch in einem Vektor vorhanden, wird 0 zurückgegeben.

Die conc Objekte rufen die `getNumberOfVariable()` (= x) Methode ihres ersten Objektes auf, ist der ermittelte Wert (x) größer oder gleich dem übergebenen Wert n, so wird dessen `getVariable(n)` Methode mit dem übergebenen Wert aufgerufen und deren Rückgabe zurückgegeben. Andernfalls wird die `getVariable(nn)` Methode ihres zweiten Objektes aufgerufen und dessen Rückgabe zurückgegeben, wobei sich der übergebene Wert (nn) aus dem Wert n minus dem ermittelten Wert x ergibt.

Die Vektorobjekte zählen für jedes `IsValue`, welches false ist, den übergebenen Wert um eins herunter, ist n gleich 0 wird von der aktuellen Komponente Component der Zeiger `Value` zurückgegeben.

`unsigned long getGreatness()`

Rückgabe: eine Abschätzung Größe/Länge des Objektes

Liefert eine Abschätzung der Größe/Länge des Objektes.

Funktions- und Bereichsobjekte addieren für jeden Vektor den sie enthalten (mittels for-Schleife) den mittels `getGreatness()` des Vektors ermittelten Wert auf, plus einmal den Wert den die `getGreatness()` Methode des enthaltenden Objektes zurückliefert und plus einmal eins (für sich selbst). Der erhaltende Wert wird zurückgegeben.

Punktobjekte addieren für jeden Vektor den sie erhalten den mit Hilfe der `getGreatness()` Methode des Vektors ermittelten Wert und einmal eins (für sich selbst) auf. Der erhaltende Wert wird zurückgegeben.

Die conc Objekte geben die Summe der `getGreatness()` Methoden Aufrufe seiner enthaltenden Objekte und 1 (für sich selbst auf) zurück.

Vektorobjekte geben die Anzahl ihrer Komponenten zurück.

(Anmerkung: Der Name der Methode `getGreatness()` beruht auf Übsetzungsschwierigkeiten. Im nachhinein wäre der Name `getSize()` besser gewesen. Die Änderung ist aber zurzeit zuviel Aufwand.)

`unsigned long getTimeNeed(unsigned int max)`

Eingabe: eine Zahl max für die Zahl ab der nicht mehr weiter gerechnet werden brauch
Rückgabe: eine Abschätzung, innerhalb eines Bereichs bis ungefähr max, für die Berechnungszeit der Matrix für des Objekt

Liefert eine Abschätzung, innerhalb eines ungefähren Bereichs von 0 bis max, der Zeit, die für die Berechnung der Matrix für das Objekt benötigt wird.

Die Zahl max dient dabei dazu, das nicht für die Zeitbestimmung sehr lange gerechnet wird, sondern diese beendet wird, wenn die Zahl max überschritten wird. Mit ein paar verschachtelten Bereichsobjekten, kann die wirkliche Zeit schon sehr schnell in die Höhe schnellen. Bei der Fitnessberechnung oder vor

der Berechnung der Matrix für das Objekt kann dann abgebrochen werden, wenn `getTimeNeed()` eine Zahl in der Nähe von max zurückgibt.

Funktionsobjekte addieren für jeden Vektor den sie enthalten (mittels for-Schleife) den mittels `getTimeNeed()` des Vektors ermittelten Wert auf, plus einmal den Wert den die `getTimeNeed()` Methode des enthaltenden Objektes zurückliefert und plus einmal eins. Vor dem Aufruf der `getTimeNeed()` Methode des enthaltenden Objektes, wird dabei noch der eigene Funktionswert berechnet (für den Fall, wenn ein Bereichsobjekt enthalten ist). Der erhaltende Wert wird zurückgegeben.

Bereichsobjekte ermitteln, für jeden Wert der in seinem Teilbereichen vorkommt (mittels for-Schleife), mittels `getTimeNeed()` die Zeit, die das enthaltende Objekt benötigt, und summiert dabei die zurückgegebenen Zeiten auf. Wenn eine Teilsumme den Wert von max überschreitet, wird diese Teilsumme, ohne weiter zu rechnen, zurückgegeben. Ansonsten wird das Endergebnis zurückgegeben.

Die conc Objekte geben die Summe der `getTimeNeed()` Methoden Aufrufe seiner enthaltenden Objekte und eins (für sich selbst) zurück.

Vektorobjekte geben die Anzahl ihrer Komponenten zurück.

Verändernde Methoden

`bool setValueToValue(unsigned int n, int val)`

Eingabe: eine Position n aus der Ordnung der Werte, des Wertes der auf den Wert val gesetzt werden soll, und val
Rückgabe: true wenn der Wert, dem in der Ordnung der Werte die Zahl n zugeordnet ist, auf den Wert val gesetzt wurde

Setzt den Wert, dem in der Ordnung der Werte die Zahl n zugeordnet ist, auf den Wert val.

Funktions-, Bereichs- und Punktobjekte ziehen, wenn n größer als x ist, zuerst für das enthaltende Objekt und dann für jeden Vektor, den sie enthalten, von n x ab ($n = n - x$), dabei ist x der mittels des Aufrufs der `getNumberOfValue()` (= x) Methode des Objektes oder Vektors ermittelte Wert.

Wenn n kleiner gleich x ist, ruft das Objekt die `setValueToValue(n,val)` Methode des Objektes oder aktuellen Vektors auf, gibt dessen Rückgabe zurück und beendet das Durchmustern seiner Vektoren. Ist der Wert weder im enthaltenden Objekt noch in einem Vektor vorhanden, wird false zurückgegeben.

Die conc Objekte rufen die `getNumberOfValue()` (= x) Methode ihres ersten Objektes auf. Ist der ermittelte Wert (x) kleiner oder gleich dem übergebenen Wert n, so wird dessen `setValueToValue(n,val)` Methode mit dem übergebenen Werten aufgerufen und deren Rückgabe zurückgegeben. Sonst wird die `setValueToValue(nn,val)` Methode ihres zweiten Objektes aufgerufen und dessen Rückgabe zurückgegeben, wobei sich der übergebene Wert (nn) aus dem Wert n minus dem ermittelten Wert x ergibt.

Die Vektorobjekte zählen für jedes `IsValue`, das true ist, den übergebenen Wert n um eins herunter. Ist n gleich 0, wird der `Value` Zeiger der aktuellen Komponente `Component` auf den Wert val gesetzt und true zurückgegeben. Sonst wird false zurückgegeben.

`bool setValueToVariable(unsigned int n, real* var)`

Eingabe: eine Position n aus der Ordnung der Werte, des Wertes der durch die Variable var ersetzt werden soll, und var
Rückgabe: true wenn der Wert, dem in der Ordnung der Werte die Zahl n zugeordnet ist, durch die Variable var ersetzt wurde

Setzt anstelle des Wertes, dem in der Ordnung der Werte die Zahl n zugeordnet ist, eine Variable var ein.

Funktions-, Bereichs- und Punktobjekte ziehen, wenn n größer als x ist, zuerst für das enthaltende Objekt und dann für jeden Vektor, den sie enthalten, von n x ab ($n = n - x$), dabei ist x der mittels des Aufrufs der `getNumberOfValue()` (= x) Methode des Objektes oder Vektors ermittelte Wert.

Ist n kleiner gleich x, ruft das Objekt die `setValueToVariable(n,var)` Methode des Objektes oder aktuellen Vektors auf, gibt dessen Rückgabe zurück und beendet das Durchmustern seiner Vektoren. Ist der Wert weder im enthaltenden Objekt noch in einem Vektor vorhanden, wird false zurückgegeben.

Die conc Objekte rufen die `getNumberOfValue()` (= x) Methode ihres ersten Objektes auf. Ist der ermittelte Wert (x) kleiner oder gleich dem übergebenen Wert n, so wird dessen `setValueToVariable(n,var)` Methode mit dem übergebenen Werten aufgerufen und deren Rückgabe zurückgegeben. Sonst wird die `setValueToVariable(nn,var)` Methode ihres zweiten Objektes aufgerufen und dessen Rückgabe zurückgegeben, wobei sich der übergebene Wert (nn) aus dem Wert n minus dem ermittelten Wert x ergibt.

Die Vektorobjekte zählen für jedes `IsValue`, das true ist, den übergebenen Wert n um eins herunter. Ist n gleich 0, wird der `Value` Zeiger der aktuellen Komponente `Component` durch die Variable var ersetzt, der alte Wert `Value` gelöscht, das zugehörige `IsValue` auf false gesetzt und true zurückgegeben. Sonst wird false zurückgegeben.

`bool setVariableToVariable(unsigned int n, real* var)`

Eingabe: eine Position n aus der Ordnung der Variablen, der Variable die durch die Variable var ersetzt werden soll, und var
Rückgabe: true wenn die Variable, der in der Ordnung der Variablen die Zahl n zugeordnet ist, durch die Variable var ersetzt wurde

Setzt anstelle der Variable, der in der Ordnung der Variablen die Zahl n zugeordnet ist, eine Variable var ein.

Funktions-, Bereichs- und Punktobjekte ziehen, wenn n größer als x ist, zuerst für das enthaltende Objekt und dann für jeden Vektor, den sie enthalten, von n x ab ($n = n - x$), dabei ist x der mittels des Aufrufs der `getNumberOfVariable()` (= x) Methode des Objektes oder Vektors ermittelte Wert.

Ist n kleiner gleich x, ruft das Objekt die `setVariableToVariable(n,val)` Methode des Objektes oder aktuellen Vektors auf, gibt dessen Rückgabe zurück und beendet das Durchmustern seiner Vektoren. Ist der Variable weder im enthaltenden Objekt noch in einem Vektor vorhanden, wird false zurückgegeben.

Die conc Objekte rufen die `getNumberOfVariable()` (= x) Methode ihres ersten Objektes auf. Ist der ermittelte Wert (x) kleiner oder gleich dem übergebenen Wert n, so wird dessen `setVariableToVariable(n,var)` Methode mit dem übergebenen Werten aufgerufen und deren Rückgabe zurückgegeben. Sonst wird die `setVariableToVariable(nn,var)` Methode ihres zweiten Objektes aufgerufen und dessen Rückgabe zurückgegeben, wobei sich der übergebene Wert (nn) aus dem Wert n minus dem ermittelten Wert x ergibt.

Die Vektorobjekte zählen für jedes `IsValue`, das false ist, den übergebenen Wert n um eins herunter. Ist n gleich 0 wird, wird der `Value` Zeiger der aktuellen Komponente `Component` durch die Variable var ersetzt und true zurückgegeben. Sonst wird false zurückgegeben.

`unsigned int setVariableToVariable(real* var1, real* var2)`

Eingabe: ein Variabelenpointer var1 der Variable die ersetzt werden soll und ein Pointer var2 der Variable durch die sie ersetzt werden soll
Rückgabe: die Anzahl der Stellen an der die Variable var1 durch die Variable var2 ersetzt wurden

Setzt anstelle der Variable var1 eine Variable var2 ein.

Nicht Vektorobjekte rufen die Methoden `setVariableToVariable(real* var1, real* var2)` der enthaltenden Objekte und Vektoren auf und geben die Summe der zurückgelieferten Zahl zurück.

Vektorobjekte setzen für jeden `Value` Zeiger der Komponenten, der gleich dem var1 real Pointer ist, den var2 Pointer ein. Die Anzahl der ersetzten Variablen `Value` wird zurückgegeben.

`bool setVariableToValue(unsigned int n, int val)`

Eingabe: eine Position n, aus der Ordnung der Variablen, der Variable die durch den Wert val ersetzt werden soll, und val
Rückgabe: true wenn die Variable, der in der Ordnung der Variablen die Zahl n zugeordnet ist, durch den Wert val ersetzt wurde

Setzt anstelle der Variable, der in der Ordnung der Variablen die Zahl n zugeordnet ist, ein Wert val ein.

Funktions-, Bereichs- und Punktobjekte ziehen, wenn n größer als x ist, zuerst für das enthaltende Objekt und dann für jeden Vektor, den sie enthalten, von n x ab ($n = n - x$), dabei ist x der mittels des Aufrufs der `getNumberOfVariable()` (= x) Methode des Objektes oder Vektors ermittelte Wert.

Ist n kleiner gleich x, ruft das Objekt die `setVariableToValue(n,var)` Methode des Objektes oder aktuellen Vektors auf, gibt dessen Rückgabe zurück und beendet das Durchmustern seiner Vektoren (beendet die for-Schleife). Ist der Wert weder im enthaltenden Objekt noch in einem Vektor vorhanden, wird false zurückgegeben.

Die conc Objekte rufen die `getNumberOfVariable()` (= x) Methode ihres ersten Objektes auf. Ist der ermittelte Wert (x) kleiner oder gleich dem übergebenen Wert n, so wird dessen `setVariableToValue(n,val)` Methode mit dem übergebenen Werten aufgerufen und deren Rückgabe zurückgegeben. Sonst wird die `setVariableToValue(nn,val)` Methode ihres zweiten Objektes aufgerufen und dessen Rückgabe zurückgegeben, wobei sich der übergebene Wert (nn) aus dem Wert n minus dem ermittelten Wert x ergibt.

Die Vektorobjekte zählen für jedes `IsValue` das false ist, den übergebenen Wert n um eins herunter. Ist n gleich 0, wird der `Value` Zeiger der aktuellen Komponente `Component` durch einen neuen Zeiger, auf einen Wert der gleich val ist, ersetzt, der zugehörige `IsValue` Wert auf true gesetzt und true zurückgegeben. Sonst wird false zurückgegeben.

`unsigned int setVariableToValue(real* var1, int val)`

Eingabe: ein Variabelenpointer var1 der Variable die ersetzt werden soll und ein Wert val durch den sie ersetzt werden soll
Rückgabe: die Anzahl der Stellen an der die Variable var1 durch den Wert val ersetzt wurden

Setzt anstelle der Variable var1 eine Wert val ein.

Nicht Vektorobjekte rufen die Methoden `setVariableToValue(real* var1, int val)` der enthaltenden Objekte und Vektoren auf und geben die Summe der zurückgelieferten Zahl zurück.

Vektorobjekte setzen jeden `Value` Zeiger der Komponenten, die gleich dem var1 real Pointer ist, auf den Wert val, wofür ein neuer real Pointer erzeugt wird, und setzen den zugehörigen Wert von `isValue` auf true. Die Anzahl der ersetzten Variablen wird zurückgegeben.

`bool changeValueAbout(unsigned int n, int dist)`

Eingabe: eine Position n aus der Ordnung der Werte, des Wertes der um den Wert dist geändert werden soll, und dist
Rückgabe: true wenn der Wert, dem in der Ordnung der Werte die Zahl n zugeordnet ist, um den Wert dist geändert wurde

Verändert den Wert, dem in der Ordnung der Werte die Zahl n zugeordnet ist, um den Wert dist.

Funktions-, Bereichs- und Punktobjekte ziehen, wenn n größer als x ist, zuerst für das enthaltende Objekt und dann für jeden Vektor, den sie enthalten, von n x ab ($n = n - x$), dabei ist x der mittels des Aufrufs der `getNumberOfValue()` (= x) Methode des Objektes oder Vektors ermittelte Wert.

Ist n kleiner gleich x, ruft das Objekt die `changeValueAbout(n,dist)` Methode des Objektes oder aktuellen Vektors auf, gibt dessen Rückgabe zurück und beendet das Durchmustern seiner Vektoren. Ist der Wert weder im enthaltenden Objekt noch in einem Vektor vorhanden, wird false zurückgegeben.

Die conc Objekte rufen die `getNumberOfValue()` (= x) Methode ihres ersten Objektes auf. Ist der ermittelte Wert (x) kleiner oder gleich dem übergebenen Wert n, so wird dessen `changeValueAbout(n,dist)` Methode mit dem übergebenen Werten aufgerufen und deren Rückgabe zurückgegeben. Sonst wird die `changeValueAbout(n,dist)` Methode ihres zweiten Objektes aufgerufen und dessen Rückgabe zurückgegeben, wobei sich der übergebene Wert (nn) aus dem Wert n minus dem ermittelten Wert x ergibt.

Die Vektorobjekte zählen für jedes `IsValue` das true ist den übergebenen Wert n um eins herunter. Ist n gleich 0, wird zum Wert, auf den der `Value` Zeiger der aktuellen Komponente `Component` zeigt, der Wert dist addiert und true zurückgegeben.

Allgemeine Methoden

`bool isUsedVariable(real* var)`

Eingabe: die zu prüfende Variable var
Rückgabe: true wenn die Variable var im Objekt noch benötigt wird

Gibt true zurück, wenn die Variable var im Objekt noch benötigt wird.

Wenn ein, der in Funktions-, Bereichs- oder Punktobjekt enthaltenden, Vektorobjekte oder enthaltenden Objekte die Variable enthält, wird true zurückgegeben. Wird mit Hilfe der `isUsedVariable(var)` der Vektorobjekte geprüft, wobei die erhaltenden Werte der Vektorobjekte mit `or` verknüpft werden und das Ergebnis zurückgegeben wird. Wenn es true ist, sonst wird noch, wenn vorhanden, das enthaltendes Objekt überprüft, mittels Aufruf von `isUsedVariable(var)`, wobei der erhaltende Wert var übergeben wird und die Rückgabe zurück gegeben wird, denn wenn die Variable im enthaltendes Objekt benötigt wird, wird sie

auch im Objekt benötigt. Wenn es kein enthaltendes Objekt gibt, wird false zurückgegeben.

Die conc Objekte geben die durch `isUsedVariable(var)` ermittelten Werte, ihrer enthaltenden Objekte, verknüpft durch `or` zurück, wenn in einem Teilobjekt die Variable benötigt wird, wird sie auch im conc Objekt benötigt.

Vektoren prüfen mit Hilfe der == Operation auf ihren Variablen (Zeigern), ob sie diese Variable enthalten und geben true zurück falls ja, sonst false.

`GraphicObject* copy() oder PictureObject* copy()`

Rückgabewert: eine Kopie des Objektes

Um Objekte in andere Hineinzukopieren ist es nötig Objekte zu kopieren. Dabei erzeugt jedes Objekt einen Zeiger auf ein neues Objekt, das mit ihm identisch ist. Enthält das Objekt ein Unterobjekt ist für dessen Kopierung dessen copy Methode zuständig.

Funktions-, Bereichs-, Punktobjekte, conc Objekte und Vektorobjekte rufen ihren Copykonstruktor mit sich selbst als Parameter auf. Der wiederum den Copykonstruktor der im übergebenen Objekt enthaltenden Objekte aufruft. Eine eventuelle definierte Variable im aktuelle Objekt wird durch eine neue Variable im ganzen Objekt (auch enthaltende Vektoren) ersetzt.

`operator=(GraphicObject* obj)`

Eingabe: ein Objekt dessen Werte kopiert werden sollen

Alle Werte (inklusive Vektoren) des übergebenen Objektes werden ins eigene Objekt kopiert. Alle Objekte prüfen zuerst, ob das übergebene Objekt vom gleichen Typ ist und, nur wenn dies der Fall ist, kopieren sie die Werte.

Funktions- und Bereichsobjekte prüfen, ob das übergebene Objekt vom gleichen Typ ist. Wenn ja löschen sie all ihre Listvektoren und kopieren dann alle Listvektoren aus dem übergebenen Objekt. Enthaltende Objekte werden nicht kopiert. So können die Listvektorwerte eines anderen ListObjektes übernommen werden.

Punktobjekte prüfen, ob das übergebene Objekt vom gleichen Typ ist. Wenn ja löschen sie all ihre Vektoren und kopieren dann alle Vektoren aus dem übergebenen Objekt.

Die conc Objekte prüfen, ob das übergebene Objekt vom gleichen Typ ist. Wenn ja löschen sie all ihre enthaltenden Objekte und kopieren dann alle enthaltenden Objekte aus dem übergebenen Objekt. Enthaltende Objekte werden hier also kopiert. Wenn sie nicht kopiert würden, wie bei Listobjekten, hätte diese Operation bei conc Objekten keine Auswirkungen und damit wenig Sinn.

Vektorobjekte prüfen, ob das übergebene Objekt vom gleichen Typ ist. Wenn ja, werden alle Komponentenwerte dieses kopiert. Für Werte wird ein neuer Zeiger erzeugt und Variablenzeiger werden übernommen.

`bool equal(GraphicObject* Obj)`

Eingabe: ein Zeiger auf ein zu vergleichende GraphicObject
Rückgabewert: true wenn das übergebene Objekt mit dem Objekt identisch ist, wenn sie also auf das gleiche Objekt zeigen, sonst false

Funktions-, Bereichs-, Punktobjekte, conc Objekte und Vektorelemente vergleichen ihren Speicherplatz mit dem übergebenen Zeiger.

`bool equalValue(GraphicObject* Obj) und bool operator== (GraphicObject* Obj)`

Eingabe: ein Zeiger auf ein zu vergleichendes GraphicObject
Rückgabewert: true wenn das übergebene Objekt mit dem Objekt übereinstimmt, sonst false

Damit nur unterschiedliche Individuen in dem Individuensatz (set) eingeordnet werden, muss getestet werden können, ob zwei Individuen nach ihren Werten gleich sind.

Zwei Funktions-, Bereichs-, Punktobjekte, conc Objekte und Vektorobjekte vergleichen dazu die Objekte die sie enthalten über deren `equalValue()` Methode, bzw. == Operator. Zuerst wird dabei mit der `classnameOf()` Methode verglichen, ob sie Instanzen der gleichen Klasse sind, und wenn nein sofort false zurückgegeben. Wenn ja werden enthaltende Objekte mit Hilfe ihrer `equalValue()` Methode verglichen und enthaltende Werte oder Zeiger mit Hilfe der == Operation.

`string classnameOf()`

Rückgabe: liefert den Klassennamen der Klasse aus der die Instanz erzeugt wurde

Gibt den Klassennamen des Objektes als string zurück.

`bool store(unsigned& int n=0, stream& stream)`

Eingabe: die Zahl n bei der mit der Nummerierung der Variablen begonnen wird und der stream in den das Objekt geschrieben werden soll
Rückgabewert: true wenn das schreiben erfolgreich war, sonst false

Die einzelnen Objekte legen ihre Werte entsprechend der möglichen Syntax aus Teil III im stream ab.

Da mir zurzeit kein Fall einfällt, bei dem das schreiben nicht erfolgreich sein kann, gibt die Methode immer true zurück. Ich belasse es trotzdem bei der `bool` Rückgabe, da vielleicht später ein möglicher Fall auftreten kann und ich dann die Signatur aller store Methoden nicht ändern muss.

Variablen werden im stream durch ein "x" und eine Nummer gekennzeichnet. Wobei beim store Aufruf von den Listobjekten diese Nummer in die Variable geschrieben wird, so das auch Vektorobjekte die Nummern für ihre Variablen einfach aus ihnen auslesen können.

Funktions- und Bereichsobjekte erhöhen den Wert n um eins und schreiben anstelle ihrer Variable "x"n in den stream. Ihre Variable wird auf den Wert von n gesetzt.

Es wird ihr Typ mit schließender Klammer "fun("oder "for(" in den stream geschrieben, dann "x"n und ",[" , dann werden der Reinfolge nach die `store(n, stream)` Methoden der enthaltenden Vektorobjekte aufgerufen und deren Rückgabe getrennt durch ein ";" in den stream geschrieben, dann "],", dann wird die `store(n, stream)` Methoden des enthaltenden Objektes aufgerufen und seine Rückgabe in den stream geschrieben und als letztes noch ")".

Punktobjekte schreiben zuerst ihren Typ mit schließender Klammer"p(" in den stream, dann werden der Reinfolge nach (zuerst Position, dann Color) die `store(n, stream)` Methoden der enthaltenden Vektorobjekte aufgerufen und deren Rückgabe getrennt durch ein "," in den stream geschrieben und als letztes noch ")".

conc Objekte schreiben zuerst ihren Typ mit öffnender Klammer "conc(" in den stream, dann rufen sie die `store(n, stream)` Methoden ihrer enthaltenden Objekte der Reinfolge nach (erst Obj1 dann Obj2) auf und schreiben deren Rückgabe, getrennt durch ein ",", in den stream und zum Schluss schreiben sie noch eine schließende Klammer in den stream ")".

Vektorobjekte schreiben zuerst eine öffnende Klammer "(" in den stream, dann der Reinfolge nach getrennt durch Komma "," ihre Elemente, Werte werden direkt in den stream geschrieben und für Variablen wird "x" gefolgt von dem Wert den sie enthält, als integer konvertiert, und als letztes wird die schließende Klammer geschrieben ")".

`bool restore(stream& in, List<real *var>& varl=NULL)`

Eingabe: eine Liste mit Variablen, die schon definiert wurden, und ihre zugehörige Nummer aus dem stream enthalten, ein stream aus dem das Objekt gelesen werden soll, dabei muss der Name und die öffnende Klammer (z.B. "fun(") für das aktuelle Objekt schon entfernt worden sein
Rückgabewert: true wenn das Objekt erstellt wurde

Erstellt das Objekt, indem es seine Werte und Objekte einrichtet und das erste fib-Element, das im stream steht, durch den entsprechenden Konstruktor dieses erstellt und dann dessen restore Methode mit dem Rest des streams aufruft.

Variablen werden im stream durch ein "x" und eine Nummer gekennzeichnet. Wobei beim restore Aufruf von den Listobjekten, diese Nummer in ihre Variable geschrieben wird und der Pointer zur Variable in die Variabelenliste varl abgelegt wird. So das auch Vektorobjekte die richtige Variablen aus der Variabelenliste heraussuchen können und diese an der richtigen Stelle, da wo im stream das "x"... steht, im eigenen Objekt einfügen können.

Wenn ein restore Methodenaufruf eines enthaltenden Objektes oder Vektors false zurückgibt oder ein Fehler beim wiederherstellen des aktuellen Objektes auftritt (z.b. ein Teil des streams entspricht nicht der Syntax des Objektes) wird false zurückgegeben, sonst true.

Beim erstellen eines Objektes, wird der Einfachheit halber davon ausgegangen, dass das Objekt mit dem Standardkonstruktor neu erzeugt wurde und damit noch "leer" ist.

Funktions- und Bereichsobjekte erstellen eine Variable und setzen sie auf den Wert der hinter dem nächsten "x" im stream steht. Diese wird dann in die Variabelenliste varl eingefügt. Dann wird das ",[" entfernt und für jeden folgenden Abschnitt, der zwischen zwei runden Klammern steht "(" , ")", die `restore(in, varl)` Methode eines dafür erzeugten entsprechenden Vektors aufgerufen, der die Variabelenliste varl und der stream übergeben wird. Die Semikolons ";" werden dabei im stream "überlesen"/gelöscht. Wenn "]," kommt, wird es entfernt und der Konstruktor, des entsprechenden Objektes dahinter, aufgerufen. Der Objektbezeichner und die folgende Klammer (z.b. "p(") werden entfernt. Der Rest des streams wird der `restore(in, varl)` des neu erzeugtem Objektes übergeben und dann die nächste runde Klammer aus dem stream gelöscht.

Punktobjekte erzeugen ein Color und Position Objekt. Dann wird zuerst die `restore(in, varl)` Methode des Positions Objektes aufgerufen und danach `restore(in, varl)` Methode des Color Objektes das zwischen den zweiten Klammern stand. Das Komma zwischen den Klammerpaaren und die schließende runde Klammer, die am Ende im stream steht, wird entfernt.

conc Objekte erzeugen für ihr erstes Objekt das entsprechende Objekt, das als nächstes im stream steht. Entfernen dessen entsprechenden Objektbezeichner und die folgende Klammer (z.B. "p("). Rufen dann dessen `restore(in, varl)` Methode mit den erhaltenden Werten auf. Entfernen dann das Komma und machen das gleiche für ihr zweites Objekt. Am Ende wird die nächste runde schließende Klammer im stream gelöscht.

Vektoren lesen für jedes ihrer Elemente, in deren Reinfolge von vorn beginnend, im übergebenen stream (Kommas werden "überlesen") jeweils den zugehörigen Teil aus (sind durch Kommas getrennt). Ist der Teil eine natürliche Zahl wird ein Zeiger auf ein `int` für das Element erzeugt und der Wert, auf den er zeigt, auf den Wert, der gleichen natürliche Zahl ist, gesetzt und der zugehörige Wert `IsValue` auf true gesetzt. Beginnt der Teil mit einem "x", wird die Zahl dahinter in eine natürliche Zahl umgewandelt und der Zeiger aus Variabelenliste varl herausgesucht, welcher auf eine natürliche Zahl zeigt, die gleich der ist, die hinter dem x stand. Dann wird der zugehörige Wert des Elements `IsValue` auf false gesetzt.

15.3.2 Zusätzliche Methoden der Klassen Point, Function, Area, Conc, ListObject und PictureObject

Die Methoden der Klasse ListObject werden implizit dadurch gekennzeichnet, dass sie bei den Klassen Function und Area gleich sind. Die Klassen Function und Area erben dann diese Methoden. Methoden die von der Klasse ListObject noch nicht realisiert werden können, sind dann in ihr als pure virtuell functions definiert.

Methoden zur Bestimmung des Wertebereichs der Ordnungen

`unsigned int getNumberOfListVectors()`

Rückgabewert: die Anzahl der Vektoren in Listen die in dem untersuchten Objekt vorhanden sind

Funktions- und Bereichsobjekte addieren zu dem Wert, der durch den Aufruf der `getNumberOfListVectors()` Methode des enthaltendem Objektes erhalten wurde, die Anzahl der Elemente in der Liste, die sie enthalten.

Punkte geben 0 zurück.

Die conc Objekte addieren die Werte zusammen, welche durch die Aufrufe `getNumberOfListVectors()` Methoden der enthaltenden Objekte ermittelt wurden.

`unsigned int getNumberOfMovePoints()`

Rückgabewert: Anzahl der Punkte des untersuchten Objektes, an denen sich ein Element befindet das verschoben werden kann

Die conc Objekte addieren die Werte, die durch die `getNumberOfMovePoints()` Methodenaufrufe der enthaltenden Objekte ermittelt wurden. Es gibt an jeder conc Abzweigung immer für jedes enthaltende Objekt soviel Elemente die verschoben werden können, wie diese zusammen enthalten. Das conc Objekt selbst kann nicht verschoben werden.

Punkte geben 0 zurück, denn Punkte können nicht verschoben werden.

Funktions- und Bereichsobjekte geben den Wert, der durch den Aufruf der `getNumberOfMovePoints()` Methode des enthaltenden Objektes erhalten wurde, erhöht um eins weiter, da sie selbst verschoben werden können.

`unsigned int getNumberOfObjects()`

Rückgabewert: die Anzahl der Punkte des untersuchten Objektes, an denen sich ein Element befindet

Die conc Objekte addieren die Werte, die durch den `getNumberOfObjects()` Methodenaufrufe der enthaltenden Objekte, die ungleich Null sind (wenn einer der Zeiger für ein Objekt Null ist, wird er nicht berücksichtigt), ermittelt wurden,

und eins zusammen. Es gibt an jeder conc Abzweigung immer für jedes enthaltende Objekt soviel Elemente wie diese zusammen enthalten, plus dem conc Objekt selbst.

Punkte geben 1 zurück, da sie selbst ein Objekt sind.

Funktions- und Bereichsobjekt geben den, durch den `getNumberOfObjects()` Methodenaufruf des enthaltenden Objektes, erhaltenden Wert erhöht um eins weiter, da sie selbst ein Objekt sind.

`unsigned int getNumberOfConc()`

Rückgabewert: Anzahl der conc Elemente des untersuchten Objektes

Die conc Objekte addieren die Werte die durch die `getNumberOfConc()` Methode der enthaltenden Objekte ermittelt wurden und eins zusammen, für es selbst.

Punkte geben 0 zurück.

Funktions- und Bereichsobjekt geben den Wert weiter, der durch den Aufruf der `getNumberOfConc()` Methode des enthaltenden Objektes ermittelt wurde.

`unsigned int getNumberOfPoints()`

Rückgabewert: Anzahl der Punktobjekte des untersuchten Objektes

Liefert den Wert des `getNumberOfConc()` Methodenaufrufes plus eins zurück. Denn jedes conc Objekt fügt ein neues Punktobjekt hinzu und ein Punktobjekt muss mindestens vorhanden sein.

`unsigned int getNumberOfObjectPoints()`

Rückgabewert: Anzahl der zusammenhängenden echten Teilobjekte des untersuchten Objektes

Liefert den Wert des `getNumberOfConc()` Methodenaufrufes multipliziert mit zwei zurück, da jedes conc zwei neue echte Teilobjekte definiert.

`unsigned int getNumberOfFunction()`

Rückgabewert: gibt die Anzahl der Funktionsobjekte des untersuchten Objektes zurück

Die conc Objekte geben die Summe der Werte zurück, welche durch die Aufrufe der `getNumberOfFunction()` Methoden der enthaltenden Objekte ermittelt wurden. Es gibt an jeder conc Abzweigung immer für jedes enthaltende Objekt soviel Funktionselemente, wie diese zusammen enthalten.

Punkte geben 0 zurück, denn Punkte sind keine Funktionsobjekte.

Funktionsobjekte geben den, durch den `getNumberOfFunction()` Methodenaufruf des enthaltenden Objektes, ermittelten Wert erhöht um eins weiter, da sie selbst ein Funktionsobjekt sind.

Bereichsobjekte geben den, durch den `getNumberOfFunction()` Methodenaufrufes des enthaltenden Objektes, ermittelten Wert weiter, da sie selbst keine Funktionsobjekte sind.

`unsigned int getNumberOfArea()`

Rückgabewert: Anzahl der Bereichsobjekte des untersuchten Objektes

Die conc Objekte geben die Summe der Werte zurück, welche durch die Aufrufe der `getNumberOfArea()` Methoden der enthaltenden Objekte ermittelt wurden. Es gibt in jedem conc Objekt soviel Bereichsobjekte, wie die beiden enthaltenden Objekte zusammen enthalten.

Punkte geben 0 zurück, denn Punkte sind keine Bereichsobjekte.

Bereichsobjekte geben den, durch den `getNumberOfArea()` Methodenaufruf des enthaltenden Objektes, ermittelten Wert erhöht um eins weiter, da sie selbst ein Bereichsobjekt sind.

Funktionsobjekte geben den, durch den `getNumberOfArea()` Methodenaufruf des enthaltenden Objektes, ermittelten Wert weiter, da sie selbst keine Bereichsobjekte sind.

Andere Abfragemethoden

`PictureObject* getObjectFrom(unsigned int n)`

Eingabe: die Position n in der Objektordnung, von welcher der Zeiger auf das Objekt geliefert werden soll
Rückgabewert: liefert das PicturObject zurück dem die Zahl n in der Objektpunkteordnung zugeordnet ist

Die conc Objekte ermittelt mit Hilfe von `getNumberOfObjects()` die Anzahl (x) der Objekte in seinem ersten Objekt und Anzahl (y) der Objekte in seinem zweiten Objekt.

Ist die Anzahl (x) gleich n ($n == x$), dann wird ein der Zeiger auf das erste Objekt zurückgeliefert.

Ist die Anzahl (x) plus die Anzahl (y) plus 1 gleich n ($n == (x+y+1)$), dann wird ein Zeiger auf das zweite Objekt zurückgeliefert.

Ist der übergebene Wert kleiner als die Anzahl (x), wird die `getObjectFrom(n)` Methode des ersten Objektes mit den erhaltenden Werten aufgerufen und dessen Rückgabewert zurückgegeben.

Sonst wird die `getObjectFrom(nn)` Methode des zweiten Objektes mit den Wert nn aufgerufen und dessen Rückgabewert zurückgegeben, nn ergibt sich dabei aus dem übergebenen Wert n minus der Anzahl (x) minus 1 ($nn = n-x-1$). Punkte sollten nicht auftreten und geben deshalb Null zurück.

Funktions- und Bereichsobjekt ermittelt mit `getNumberOfObjects()` die Anzahl (x) der Objekte in dem enthaltenden Objekt.

Ist die Anzahl (x) gleich n ($n == x$), dann wird ein Zeiger auf das enthaltende Objekt zurückgegeben. Ist der übergebene Wert kleiner als die Anzahl (x), wird die `getObjectFrom(n)` Methode des enthaltenden Objektes mit den erhaltenden Werten aufgerufen und dessen Rückgabewert zurückgegeben. Sonst wird Null zurückgegeben.

`Vector* getListVectorFrom(unsigned int n)`

Eingabe: die Position n, in der Ordnung der Listenvektoren, von welcher der Zeiger auf den Vector geliefert werden soll
Rückgabewert: liefert den Vector zurück dem die Zahl n in der Ordnung der Listenvektoren zugeordnet ist

Die conc Objekte ermittelt mit Hilfe von `getNumberOfListVectors()` die Anzahl (x) der Listenvektoren in seinem ersten Objekt.

Ist der übergebene Wert n kleiner oder gleich der Anzahl x ($n \leq x$), wird die `getListVectorFrom(n)` Methode des ersten Objektes mit den erhaltenden Werten aufgerufen und dessen Rückgabewert zurückgegeben.

Sonst wird die `getListVectorFrom(nn)` Methode des zweiten Objektes mit den Wert nn aufgerufen und dessen Rückgabewert zurückgegeben, nn ergibt sich dabei aus dem übergebenen Wert n minus der Anzahl (x) ($nn = n - x$). Punkte sollten nicht auftreten und geben deshalb Null zurück.

Funktions- und Bereichsobjekt ermittelt mit `getNumberOfListVectors()` die Anzahl (x) der Listvektoren in den enthaltenden Objekten.

Ist der übergebene Wert n kleiner oder gleich der Anzahl x ($n = x$), wird die `getListVectorFrom(n)` Methode des enthaltenden Objektes mit den erhaltenden Werten aufgerufen und dessen Rückgabewert zurückgegeben. Sonst wird ein Zeiger auf den Vektor der enthaltenden Liste zurückgegeben, der an der Stelle $n - x$ steht (dem ersten Vector in der Liste ist die 1 zugeordnet, bzw. er steht an der Stelle 1).

`PictureObject* getNext(Bool b=true)`

Eingabe: ein bool Wert n der bei conc Objekten angibt, ob das erste Objekt zurückgeliefert werden soll (true) oder das zweite (false), standardmäßig ist b true
Rückgabe: bei Funktions- und Bereichsobjekten einen Zeiger auf das enthaltende Objekt, bei conc Objekten, wenn b true ist einen Zeiger auf das erste enthaltende Objekt, sonst ein Zeiger auf das zweite enthaltende Objekt

Liefert bei Funktions- und Bereichsobjekten einen Zeiger auf das nächste Objekt, unter dem Objekt von dem die Methode aufgerufen wurde, bzw. einen Zeiger auf das enthaltende Objekt.

Bei conc Objekten wird, wenn b true ist, ein Zeiger auf das erste enthaltende Objekt zurückgegeben, sonst ein Zeiger auf das zweite enthaltende Objekt. Standardmäßig ist b true und es wird somit standardmäßig immer ein Zeiger auf das erste enthaltende Objekt zurückgeliefert.

Funktions- und Bereichsobjekten liefern einen Zeiger auf ihr enthaltendes Objekt zurück.

Die conc Objekte liefern, wenn b true ist, einen Zeiger auf das erste enthaltende Objekt zurück, sonst ein Zeiger auf das zweite enthaltende Objekt.

Punkte liefern Null zurück, da sie keine Objekte enthalten.

`unsigned int partObjectToPointPartObject(unsigned int n)`

Eingabe: eine Zahl n die einem Teilobjekt in der Teilobjektpunktordnung zugeordnet ist
Ausgabe: eine Zahl n die dem entsprechenden Punktteilobjekt in der Punktteilobjektordnung zugeordnet ist oder 0, falls das Teilobjekt kein Punktteilobjekt ist oder nicht existiert

Wandelt eine Zahl n, die einem Teilobjekt in der Teilobjektpunktordnung zugeordnet ist, in eine Zahl um, die dem entsprechenden Punktteilobjekt in der Punktteilobjektordnung zugeordnet ist oder 0, falls das Teilobjekt kein Punktteilobjekt ist oder nicht existiert.

Die conc Objekte ermitteln zuerst durch Aufruf der `getNumberOfConc()` Methode die Anzahl (x) der conc Elemente im ersten Objekt. Ist diese Zahl x mal 2 (= Anzahl der Teilobjekte in diesem) größer oder gleich der übergebene Zahl n ($x * 2 = n$), wird die `partObjectToPointPartObject(n)` des ersten Objektes aufgerufen und dessen Rückgabe zurückgegeben, da das Teilobjekt sich im ersten Objekt des conc Objektes befindet. Ist diese Zahl x mal 2 (= Anzahl der Teilobjekte in diesem) plus 2 kleiner oder gleich der übergebene Zahl n ($x*2+2 \leq n$), wird die `partObjectToPointPartObject(nn)` Methode des zweiten Teilobjektes aufgerufen, wobei der übergebene Wert nn gleich dem erhaltenden Wert n minus des durch `getNumberOfConc()` ermittelten Wertes mal 2 minus 2 ist ($nn = n - x * 2 - 2$). Es werden also die Teilobjekte des ersten Objektes im conc Objekt abgezogen. Zurückgegeben wird der Wert, der erhalten wird, wenn zum durch `partObjectToPointPartObject(nn)` erhaltenden Wert der durch `getNumberOfConc()` ermittelten Werte plus 1 addiert wird (`return partObjectToPointPartObject(nn)+x+1`). Also die Anzahl der Punktobjekte (Punkte) im ersten Objekt plus die Position die aus dem zweitem Objekt ermittelt wurde.

Sonst wird 0 zurückgegeben. Damit wird für das Teilobjekt, welches das conc ganz enthält, (aber keine conc Objekte über ihm) und dem die Zahl $x * 2 + 1$ in der Ordnung der Teilobjekte zugeordnet ist, als nicht Punktteilobjekt klassifiziert und für dies wird 0 zurückgegeben.

Punkte geben 1 zurück, da sie selbst ein Punktobjekt darstellen.

Funktions- und Bereichsobjekt geben den Wert weiter, der durch den Aufruf der `partObjectToPointPartObject(n)` Methode des enthalten Objektes ermittelt wurde.

`unsigned int pointPartObjectToPartObject(unsigned int n)`

Eingabe: die Zahl die einem Punktteilobjekt in der Punktteilobjektpunktordnung zugeordnet ist
Ausgabe: die Zahl die dem selben Objekt in der Teilobjektordnung zugeordnet ist

Wandelt eine Zahl n, die einem Punktteilobjekt in der Punktteilobjektordnung zugeordnet ist, in eine Zahl um, die dem entsprechenden Teilobjekt in der Teilobjektordnung zugeordnet ist.

Die conc Objekte ermitteln zuerst, durch Aufruf der `getNumberOfConc()` Methode, die Anzahl (x) der conc Elemente im ersten Objekt.

Ist diese Zahl x plus eins (= Anzahl der Punktteilobjekte bzw. Punkte in diesem) größer oder gleich der übergebene Zahl n ($x + 1 \geq n$), wird die Methode `pointPartObjectToPartObject(n)` des ersten Objektes aufgerufen und deren Rückgabe zurückgegeben, da das Punktteilobjekt sich im ersten Objekt des conc Objektes befindet. Sonst wird die `pointPartObjectToPartObject(nn)` Methode des zweiten Teilobjektes aufgerufen, wobei der übergebene Wert nn gleich dem erhaltenden Wert n, minus des durch `getNumberOfConc()` ermittelten Wertes, plus eins ($nn = n - (x + 1)$) ist. Es werden also die Punktteilobjekte des ersten Objektes im conc Objekt abgezogen. Zurückgegeben wird der Wert, der erhalten wird, wenn zum durch `pointPartObjectToPartObject(nn)` erhaltenden Wert der durch `getNumberOfConc()` ermittelten Werte mal zwei plus zwei addiert wird (`return pointPartObjectToPartObject(nn) + x * 2 + 2`). Also die Anzahl der Teilobjekte im ersten Objekt, plus die Position, die aus dem zweitem Objekt ermittelt wurde, plus die eigenen zwei Teilobjekte.

Punkte geben den übergebenen Wert n minus zwei zurück, da sie selbst ein Objekt in einem conc Objekt darstellen und n angibt ob das erste oder das zweite, aber das conc zwei aufaddiert.

Funktions- und Bereichsobjekt geben den Wert weiter, der durch den Aufruf der `pointPartObjectToPartObject(n)` Methode des enthaltenden Objektes ermittelt wurde.

`unsigned int movePointToObjectPoint(unsigned int n)`

Eingabe: die Zahl die einem Verschiebepunkt in der Verschiebepunktordnung zugeordnet ist
Ausgabe: die Zahl die dem selben Objekt in der Objektpunktordnung zugeordnet ist

Wandelt eine Zahl eines Objektes aus der Verschiebepunktordnung in die entsprechende Zahl des gleichen Objektes in der Objektpunktordnung um.

Die conc Objekte ermitteln zuerst, durch die `getNumberOfMovePoints()` Methode, die Anzahl (x) der Verschiebepunkt im ersten Objekt.

Ist diese Zahl x (= Anzahl der Verschiebepunkte in diesem) größer oder gleich wie die übergebene Zahl n ($x = n$), wird die `movePointToObjectPoint(n)` des ersten Objektes aufgerufen und dessen Zurückgabe zurückgegeben, da das Verschiebeobjekt sich im ersten Objekt des conc Objektes befindet.

Sonst wird die `movePointToObjectPoint(nn)` Methode des zweiten Teilobjektes aufgerufen, wobei der übergebene Wert nn gleich dem erhaltenden Wert n, minus des durch `getNumberOfMovePoints()` ermittelten Wertes ($nn = n - x$) ist. Es werden also die Verschiebeobjekte des ersten Objektes im conc Objekt abgezogen. Zurückgegeben wird der Wert, der erhalten wird, wenn zum durch `movePointToObjectPoint(nn)` erhaltenden Wert der Wert x, welcher durch `getNumberOfObjects()` des ersten Objektes ermittelt wurde, und eins addiert wird (`return movePointToObjectPoint(nn) + getNumberOfObjects() + 1`), also die Anzahl der Objektpunkt im ersten Objekt, plus die Position, die aus dem zweitem Objekt ermittelt wurde, plus sich selbst.

Punkte geben 0 zurück, sie sollten allerdings nicht auftreten, da sie keine verschiebbaren Objekte sind.

Funktions- und Bereichsobjekt geben, wenn n gleich dem Wert plus eins (für sich selbst) ist, welcher durch die `getNumberOfMovePoints()` Methode des enthaltenden Objektes ermittelt wurde ($n == getNumberOfMovePoints() + 1$), den durch `getNumberOfObjects()`, des enthaltenden Objektes, ermittelten Wert plus eins (für sich selbst) zurück. Sonst wird der Wert zurückgegeben, welcher durch den Aufruf der `movePointToObjectPoint(n)` Methode des enthaltenden Objektes ermittelt wurde.

`bool isDeletableElement()`

Rückgabe: true wenn das Element löschbar ist (es keine Variable definiert die noch benötigt wird), ansonsten false

Prüft ob dieses Element gelöscht werden kann.

Die conc Objekte und Punkte geben false zurück, da sie nicht löschbare Elemente sind.

Funktions- und Bereichsobjekt prüfen mit Hilfe von `isUsedVariable(var)`, ob die Variable var die sie definieren, noch benötigt wird, wenn ja geben sie false zurück, sonst true.

`bool hasUnderAllObjects()`

Rückgabe: true wenn das Objekt noch alle seiner Unterobjekte hat, sonst false

Prüft ob im aktuellen Objekt ein Unterobjekte fehlt.

Die conc Objekte, Punkte-, Funktions- und Bereichsobjekt prüfen, ob einer ihrer Zeiger auf ihre Unterobjekte mit Null referenziert ist. Wenn ja geben sie false zurück, sonst true.

`bool isUsedVariableInElement(real* var)`

Eingabe: eine zu prüfende Variable var
Rückgabe: true wenn die Variable var im Element (also ohne Unterobjekte) noch benötigt wird

Prüft ob die übergebene Variable im Element noch benötigt wird, also im Objekt ohne enthaltende fib-Objekte.

Wenn in einem, der in Funktions-, Bereichs- oder Punktobjekt enthaltenden, Vektorobjekt die Variable enthalten ist, wird true zurückgegeben. Wird mit Hilfe der `isUsedVariable(var)` der Vektorobjekte geprüft, wobei die erhaltenden Werte der Vektorobjekte mit `or` verknüpft werden und das Ergebnis zurückgegeben wird.

Die conc Objekte geben false zurück.

`real* getDefineVariableFrom(unsigned int n)`

Eingabe: die Zahl des Objektes, aus der Verschiebeordnung der Objekte, dessen Variable, die es definiert, zurückgegeben werden soll
Rückgabe: die Variable die das Objekt definiert, bzw. einen Zeiger auf einen real Wert

Liefert die Variable die das Objekt definiert, dem die Zahl n in der Ordnung der Verschiebepunkt zugeordnet ist.

Die conc Objekte ruft die Methode `getNumberOfMovePoints()` (= x) seines ersten Objektes auf um die Anzahl (x) der enthaltenden Verschiebepunkte zu ermitteln. Ist die Anzahl x größer oder gleich dem übergebenen Wert n ($x == n$), wird die `getDefineVariableFrom(n)` des ersten enthaltenden Objektes mit dem erhaltenden Wert n aufgerufen und dessen Rückgabe zurückgegeben. Sonst wird die `getDefineVariableFrom(nn)` des zweiten enthaltenden Objektes aufgerufen und dessen Rückgabe zurückgegeben, wobei sich der übergebene Wert nn aus dem erhaltenden Wert n minus dem ermittelten Wert x ergibt ($nn = n - x$).

Punkte geben Null zurück, denn Punkte definieren keine Variablen. (Sollte nicht auftreten.)

Wenn bei Funktions- und Bereichsobjekt n gleich dem mittels der Methode `getNumberOfMovePoints()` des enthaltenden Objekt ermittelten Wert x plus eins ist ($n == x + 1$), geben sie die Variable, welche sie definieren, zurück. Andernfalls wird die `getDefineVariableFrom(n)` Methode des enthaltenden Objektes mit dem erhaltenden Wert n aufgerufen und dessen Rückgabe zurückgegeben.

list<real*> getAllDefineVariableOver(unsigned int n)

Eingabe: die Zahl, aus der Verschiebeordnung, des Objektes bis zu dem alle Variable, die über ihm definiert werden, zurückgeliefert werden sollen
Rückgabe: eine Liste aller Variablen (Zeiger auf real Werte), die über dem Objekt definiert werden

Liefert alle Variable die über dem Objekt definiert werden, dem die Zahl n in der Ordnung der Verschiebepunkt zugeordnet ist, und die dieses somit verwenden darf.

Die conc Objekte ruft die Methode `getNumberOfMovePoints()` (= x) seines ersten Objektes auf, um die Anzahl (x) der enthaltenden Verschiebepunkte (jeder Verschiebepunkt definiert eine eigene Variable) zu ermitteln. Ist die ermittelte Anzahl x größer oder gleich dem übergebenen Wert n ($x \geq n$), wird die `getAllDefineVariableOver(n)` Methode des ersten enthaltenden Objektes, mit dem erhaltenden Wert n, aufgerufen und dessen Rückgabe zurückgegeben.

Sonst wird die `getAllDefineVariableOver(nn)` Methode des zweiten enthaltenden Objektes aufgerufen und dessen Rückgabe zurückgegeben, wobei sich der übergebene Wert nn aus dem erhaltenden Wert n, minus dem ermittelten Wert x ergibt ($nn = n - x$).

Punkte geben eine leere Liste zurück, denn Punkte definieren keine Variablen. (Sollte nicht auftreten.)

Wenn bei Funktions- und Bereichsobjekt n kleiner als der, mittels des Aufrufes der `getNumberOfMovePoints()` Methode des enthaltenden Objekt, ermittelten Wert (x) ist ($n < x$), rufen sie die `getAllDefineVariableOver(n)` Methode des enthaltenden Objektes auf, fügen in die zurückgegebene Liste ihre eigene Variable ein und geben dann diese Liste zurück. Sonst wird eine leere Liste zurückgegeben.

list<real*> getAllDefineVariableOverValue(unsigned int n)

Eingabe: die Zahl n, aus der Ordnung der Werte, des Wertes aus dem Objekt bis zu dem alle Variable, die über dem zugehörigen Wert definiert werden, zurückgeliefert werden sollen
Rückgabe: eine Liste aller Variablen (Zeiger auf real Werte), die über dem Wert definiert werden

Liefert alle Variable die über dem Wert definiert werden, dem die Zahl n in der Ordnung der Werte zugeordnet ist, und durch die dieser Wert somit ersetzt werden kann.

Die conc Objekte ruft die Methode `getNumberOfValue()` (= x) des ersten enthaltenden Objektes auf, um die Anzahl (x) der enthaltenden Werte zu ermitteln. Ist die Anzahl x größer oder gleich dem übergebenen Wert n ($x \geq n$), wird die

`getAllDefineVariableOverValue(n)` des ersten enthaltenden Objektes, mit dem erhaltenden Wert n, aufgerufen und dessen Rückgabe zurückgegeben.

Sonst wird die `getAllDefineVariableOverValue(nn)` Methode des zweiten enthaltenden Objektes aufgerufen und dessen Rückgabe zurückgegeben, wobei sich der übergebene Wert nn aus dem erhaltenden Wert n, minus dem ermittelten Wert x ergibt ($nn = n - x$).

Punkte geben eine leere Liste zurück, denn Punkte definieren keine Variablen. (Sollte nicht auftreten.)

Wenn bei Funktions- und Bereichsobjekt n kleiner als der, mittels Aufrufes ihrer `getNumberOfValue()` Methode ermittelten Wert (x), ist ($n < x$), rufen sie die `getAllDefineVariableOverValue(n)` Methode des enthaltenden Objektes auf, fügen in die zurückgegebene Liste ihre eigene Variable ein und geben dann diese Liste zurück. Der Wert ist dann weder in diesem Element noch in Elementen über ihm. Sonst wird eine leere Liste zurückgegeben.

`list<real*> getAllDefineVariableOverVariable(unsigned int n)`

Eingabe: die Zahl n, aus der Ordnung der Variablen, der Variable aus dem Objekt bis zu dem alle Variable, die über der zugehörigen Variable definiert werden, zurückgeliefert werden sollen
Rückgabe: eine Liste aller Variablen (Zeiger auf real Werte), die über der Variablen definiert werden

Liefert alle Variable die über der Variablen definiert werden, dem die Zahl n in der Ordnung der Variablen zugeordnet ist, und durch die diese Variable somit ersetzt werden kann.

Die conc Objekte ruft die Methode `getNumberOfVariable()` (= x) des ersten enthaltenden Objektes auf, um die Anzahl (x) der enthaltenden Variablen zu ermitteln. Ist die Anzahl x größer oder gleich dem übergebenen Wert n ($x \geq n$), wird die `getAllDefineVariableOverVariable(n)` des ersten enthaltenden Objektes, mit dem erhaltenden Wert n, aufgerufen und dessen Rückgabe zurückgegeben.

Sonst wird die `getAllDefineVariableOverVariable(nn)` Methode des zweiten enthaltenden Objektes aufgerufen und dessen Rückgabe zurückgegeben, wobei sich der übergebene Wert nn aus dem erhaltenden Wert n, minus dem ermittelten Wert x ergibt ($nn = n - x$).

Punkte geben eine leere Liste zurück, denn Punkte definieren keine Variablen. (Sollte nicht auftreten.)

Wenn bei Funktions- und Bereichsobjekt n kleiner als der, mittels Aufrufes ihrer `getNumberOfVariable()` Methode, ermittelten Wert(x) ist ($n < x$), rufen sie die `getAllDefineVariableOverVariable(n)` Methode des enthaltenden Objektes auf, fügen in die zurückgegebene Liste ihre eigene Variable ein

und geben dann diese Liste zurück. Die Variable ist dann weder in diesem Element noch in Elementen über ihm. Sonst wird eine leere Liste zurückgegeben.

Verändernde Methoden

`bool insertObjectInObject(unsigned int n, PictureObject* obj, bool ov)`

Eingabe: ein Zeiger auf ein einzufügende PictureObject obj, die Position n in der Objektordnung direkt oberhalb der an der das Objekt eingefügt werden soll und eine boolsche Variable, die angibt ob das Objekt, durch das Objekt, welches an der Stelle n steht, überdeckt werden darf
Rückgabe: true wenn das Einfügen gelungen ist, false sonst

Fügt ein Teilobjekt obj an der Stelle direkt über dem Objekt, dem in der Objektordnung die Zahl n zugeordnet ist, mit Hilfe eines conc Objektes ein.

Die conc Objekte ermittelt mit Hilfe der `getNumberOfObjektes()` Methode die Anzahl (x) der Objekte in seinem ersten Objekt und Anzahl (y) der Objekte in seinem zweiten Objekt.

Ist die Anzahl (x) gleich n ($n == x$), dann wird ein neues conc Objekt anstelle des ersten Objektes eingefügt, ist ov true wird das aktuelle erste conc Objekt das erste Objekt des eingefügten conc Objektes und das übergebene das zweite, ansonsten andersherum, es wird true zurückgegeben.

Ist die Anzahl (x) plus die Anzahl (y) plus eins gleich n ($n == (x + y + 1)$), dann wird ein neues conc Objekt anstelle des zweiten Objektes eingefügt. Ist ov true wird das aktuelle zweite conc Objekt das erste Objekt des eingefügten conc Objektes und das Übergebene das Zweite, ansonsten andersherum, es wird true zurückgegeben. Ist der übergebene Wert n kleiner als die Anzahl (x) ($n < x$), wird die `insertObjectInObject(n,obj, ov)` Methode des ersten Objektes mit den erhaltenden Werten aufgerufen und dessen Rückgabewert zurückgegeben.

Sonst wird die `insertObjectInObject(nn,obj, ov)` Methode des zweiten Objektes mit den erhaltenden Werten, bis auf nn, aufgerufen und dessen Rückgabewert zurückgegeben, nn ergibt sich dabei aus dem übergebenen Wert n, minus der Anzahl (x), minus eins ($nn = n - x - 1$).

Punkte sollten nicht auftreten und geben deshalb false zurück.

Funktions- und Bereichsobjekt ermittelt die Anzahl (x) der Objekte in dem Objekt, welches sie enthalten, mit Hilfe von `getNumberOfObjects()`.

Ist die Anzahl (x) gleich n ($n == x$), wird ein neues conc Objekt anstelle des enthaltenden Objektes eingefügt, ist ov true wird das aktuell enthaltende Objekt das erste Objekt des eingefügten conc Objektes und das übergebene das zweite, ansonsten andersherum, es wird true zurückgegeben.

Ist der übergebene Wert n kleiner als die Anzahl (x) ($n < x$), wird die Methode `insertObjectInObject(n,obj, ov)` des enthaltenden Objektes mit den erhaltenden Werten aufgerufen und dessen Rückgabewert zurückgegeben.

Sonst wird false zurückgegeben.

`bool overwriteObjectWithObject(unsigned int n, PictureObject* obj)`

Eingabe: ein Zeiger auf ein einzufügende PictureObject obj, die Position n in der Objektordnung an der das Objekt eingefügt werden soll und an der das Objekt steht das überschrieben werden soll
Rückgabe: true wenn das Überschreiben gelungen ist, false sonst

Überschreibt das Objekt, dem die Zahl n in der Objektordnung zugeordnet, ist mit den gegebenen Objekt obj.
Achtung: Das überschriebene Objekt wird nicht gelöscht!

Die conc Objekte ermittelt, mit Hilfe von der `getNumberOfObjects()` Methode, die Anzahl (x) der Objekte in seinem ersten Objekt und die Anzahl (y) der Objekte in seinem zweiten Objekt.

Ist die Anzahl (x) gleich n ($n == x$), dann wird das übergebene Objekt anstelle des ersten Objektes eingefügt und true wird zurückgegeben.

Ist die Anzahl (x), plus die Anzahl (y), plus eins gleich n ($n == (x + y + 1)$), dann wird das übergebene Objekt anstelle des zweiten Objektes eingefügt, true wird zurückgegeben.

Ist der übergebene Wert n kleiner als die Anzahl (x) ($n < x$), wird die Methode `overwriteObjectWithObject(n, obj, ov)` des ersten Objektes mit den erhaltenden Werten aufgerufen und dessen Rückgabewert zurückgegeben. Sonst wird die `overwriteObjectWithObject(nn, obj)` Methode des zweiten Objektes mit den erhaltenden Werten, bis auf nn, aufgerufen und dessen Rückgabewert zurückgegeben, nn ergibt sich dabei aus dem übergebenen Wert n, minus der Anzahl (x), minus eins ($nn = n - x - 1$).

Punkte sollten nicht auftreten und geben deshalb false zurück.

Funktions- und Bereichsobjekt ermittelt die Anzahl (x) der Objekte in dem Objekt, welches sie enthalten, mit Hilfe von `getNumberOfObjects()`.

Ist die Anzahl (x) gleich n ($n == x$), wird das übergebene Objekt anstelle des enthaltenden Objektes eingefügt. Ist der übergebene Wert n kleiner als die Anzahl (x) ($n < x$), wird die `overwriteObjectWithObject(n, Obj)` Methode des enthaltenden Objektes mit den erhaltenden Werten aufgerufen und dessen Rückgabewert zurückgegeben. Sonst wird false zurückgegeben.

`bool removeObject(unsigned int n)`

Eingabe: die Position n in der Teilobjektordnung dessen zugeordnetes Objekt gelöscht werden soll
Rückgabe: true wenn das Löschen gelungen ist, false sonst

Löscht das Teilobjekt dem die Zahl n Teilobjektordnung zugeordnet ist.

Die conc Objekte ermittelt, mit Hilfe von der `getNumberOfObjectPoints()` Methode, die Anzahl (x) der Teilobjekte in seinem ersten Objekt.

Ist die Anzahl (x) plus eins gleich n ($n == x+1$), wird die `deleteObject()` Methode auf dem ersten Objekt ausgeführt, dann dessen Destruktor, danach das zweite Objekt an dessen Stelle gesetzt (Objekte werden vertauscht) und der Zeiger des zweiten Objektes wird auf Null gesetzt.

Ist die Anzahl (x) plus zwei (die beiden eigenen Teilobjekte) gleich n ($n == (x+2)$), wird die `deleteObject()` Methode des zweiten Objektes ausgeführt, dann dessen Destruktor und der Zeiger des Objektes wird auf Null gesetzt.

Ist der übergebene Wert n kleiner als die Anzahl (x) ($n < x$), wird die Methode `removeObject(n)` des ersten Objektes mit den erhaltenden Werten aufgerufen und dessen Rückgabewert zurückgegeben.

Sonst wird die `removeObject(nn)` Methode des zweiten Objektes mit den Wert nn aufgerufen und dessen Rückgabewert zurückgegeben, nn ergibt sich dabei aus dem übergebenen Wert n minus der Anzahl (x) minus zwei ($nn = n - x - 2$), für die eigenen beiden Objekte.

Wenn `removeObject(x)` true zurückgegeben hat, wird mit Hilfe der Methode `hasAllUnderObjects()` geprüft, ob das entsprechenden enthaltende Objekt noch alle Unterobjekte besitzt. Wenn ja wird true zurückgegeben. Sonst ist es ein conc Objekt, in dem ein Objekt gelöscht wurde. Dann wird die Methode `getNumberOfObjects()` (= x) des entsprechenden enthaltenden Objektes (wo das Teilobjekt gelöscht wurde) aufgerufen und mit dem erhaltenden Wert minus eins ($nn = x-1$ es wird das conc abgezogen) die Methode `getObjectFrom(nn)` des entsprechenden enthaltenden Objektes aufgerufen. Das von ihr gelieferte Objekt wird als neues enthaltende Objekt genommen und vom altem Objekt wird der Destruktor aufgerufen (das conc Objekt wird zerstört). Zum Schluss wird true zurückgegeben.

Funktions- und Bereichsobjekt rufen die `removeObject(n)` Methode ihres UnterObjektes auf.

Wenn `removeObject(n)` true zurückgegeben hat, wird mit Hilfe der Methode `hasAllUnderObjects()` geprüft, ob das enthaltende Objekt noch alle Unterobjekte besitzt. Wenn ja wird true zurückgegeben. Andernfalls ist das enthaltende Objekt ein conc Objekt, in dem ein Objekt gelöscht wurde. Dann wird die Methode `getNumberOfObjects()` (= x) des enthaltenden Objektes aufgerufen und mit dem erhaltenden Wert minus eins ($nn = x - 1$ es wird das conc abgezogen) die Methode `getObjectFrom(nn)` aufgerufen. Das von ihr gelieferte Objekt wird als neues enthaltende Objekt genommen und vom altem enthaltenden Objekt wird der Destruktor aufgerufen (das conc Objekt wird zerstört). Zum Schluss wird true zurückgegeben.

Punktobjekte geben false zurück, da sie kein Teilobjekt sind und auch keins enthalten das gelöscht werden kann.

bool removeElement(unsigned int n)

Eingabe: die Position n, in der Verschiebeordnung, dessen zugeordnetes Element gelöscht werden soll
Rückgabe: true wenn das Löschen gelungen ist, sonst false

Löscht das Element dem die Zahl n in der Verschiebeordnung zugeordnet ist, wenn es löschbar ist.

Es wird die eigene Methode `cutElement(n)` aufgerufen. Wenn sie ein Zeiger auf ein Objekt zurückgibt, wird dieses gelöscht und true zurückgegeben, sonst wird nur false zurückgegeben.

bool removeListVector(unsigned int n)

Eingabe: die Position n in der Ordnung der Listenvektoren, dessen zugehöriger Vector gelöscht werden soll
Rückgabewert: true wenn der Vector gelöscht wurde, sonst false

Löscht den Listvektor dem die Zahl n in der Ordnung der Listvektoren zugeordnet ist.

Die conc Objekte ermittelt mit Hilfe von `getNumberOfListVectors()` die Anzahl (x) der Listenvektoren in seinem ersten Objekt. Ist der übergebene Wert n kleiner oder gleich der Anzahl x ($n \leq x$), wird die `removeListVector(n)` Methode des ersten Objektes, mit den erhaltenden Werten, aufgerufen und dessen Rückgabewert zurückgegeben. Sonst wird die `removeListVector(nn)` Methode des zweiten Objektes, mit den Wert nn aufgerufen, und dessen Rückgabewert zurückgegeben, nn ergibt sich dabei aus dem übergebenen Wert n minus der Anzahl (x) ($nn = n - x$).

Punkte sollten nicht auftreten und geben deshalb false zurück.

Funktions- und Bereichsobjekt ermittelt mit `getNumberOfListVectors()` die Anzahl (x) der Objekte in dem enthalten Objekten. Ist der übergebene Wert n kleiner oder gleich der Anzahl x ($n \leq x$), wird die `removeListVector(n)` Methode des enthaltenden Objektes mit den erhaltenden Werten aufgerufen und dessen Rückgabewert zurückgegeben. Sonst wird der Vector in der enthaltenden Liste gelöscht, der an der Stelle $n - x$ steht (dem ersten Vector in der Liste ist die 1 zugeordnet, bzw. er steht an der Stelle 1). Wenn das nicht möglich ist, wird false zurückgegeben.

PictureObject* cutElement(unsigned int n, bool b)

Eingabe: die Position n in der Verschiebeordnung, dessen zugeordnetes Element ausgeschnitten werden soll und eine boolscher Wert der angibt, ob auch ohne zu prüfen, ob das Element löschbar ist, das Element gelöscht werden darf
Rückgabe: einen Zeiger auf das ausgeschnittene Element (noch mit Unterobjekt), wenn das ausschneiden gelungen ist, sonst Null

Schneidet das Element aus, dem die Zahl n in der Verschiebeordnung zugeordnet ist.

Achtung: Wenn b true ist wird nicht geprüft, ob die Variable die es eventuell definiert, noch benötigt wird. Sonst wird dies geprüft.

Die conc Objekte ermittelt, mit Hilfe der `getNumberOfMovePoints()` Methode, die Anzahl (x) der verschiebe Objekte in seinem ersten Objekt und Anzahl (y) der verschiebe Objekte in seinem zweiten Objekt.

Ist die Anzahl (x) gleich n ($n == x$), wird versucht das erste Objekt auszuschneiden.

Ist die Anzahl (x) plus die Anzahl (y) gleich n ($n == (x + y)$), wird versucht das zweite Objekt auszuschneiden. Ist der übergebene Wert n kleiner als die Anzahl (x) ($n < x$), wird die `cutElement(n, b)` Methode des ersten Objektes mit den erhaltenden Werten aufgerufen und dessen Rückgabewert zurückgegeben.

Sonst wird die `cutElement(nn, b)` Methode des zweiten Objektes mit den Wert nn aufgerufen und dessen Rückgabewert zurückgegeben, nn ergibt sich dabei aus dem übergebenen Wert n minus der Anzahl (x) der Verschiebepunkt im ersten Objekt ($nn = n - x$).

Punkte sollten nicht auftreten und geben deshalb Null zurück.

Funktions- und Bereichsobjekt ermittelt mit `getNumberOfMovePoints()` die Anzahl (x) der Objekte im enthaltendem Objekt. Ist die Anzahl (x) gleich n ($n == x$), wird versucht das enthaltende Objekt auszuschneiden. Ist der übergebene Wert kleiner als die Anzahl (x), wird die `cutElemnet(n, b)` Methode des enthaltenden Objektes mit den erhaltenden Werten aufgerufen und dessen Rückgabewert zurückgegeben.

Beim Versuch ein Objekt auszuschneiden, wird zuerst überprüft, ob das auszuschneidende Objekt wirklich das enthaltende Objekt ist und dieses kein conc Objekt ist. Dafür wird geprüft ob, die `classnameOf()` Methode des Objektes "conc" zurückgibt. Wenn ja, wird die `cutElemnet(n, b)` Methode des Objektes aufgerufen. Sonst wird, wenn b false ist, mit Hilfe der `isDeletableElement()` Methode überprüft, ob das Element entfernt/ausgeschnitten werden kann. Wenn nein, wird Null zurückgegeben. Sonst wird mit Hilfe der `getNext()` Methode das Objekt ermittelt, das das auszuschneidende Objekt enthält. Dann wird anstelle des auszuschneidenden Objekt im aktuellen Objekt, das mit Hilfe von `getNext()`, ermittelte Objekt eingefügt und das nun ausgeschnittene Objekt zurückgegeben.

Sonst wird Null zurückgegeben.

`void deleteObject()`

Löscht alle enthaltenden Objekte.

Die conc Objekte ruft zuerst die `deleteObject()` Methoden ihrer Teilobjekte auf und dann deren Destruktoren, referenziert danach die beiden enthaltenden Zeiger der enthaltenden Objekte mit Null.

Punkte tun nichts. (ihre enthaltenden Vektoren werden automatisch mit ihnen vernichtet)

Funktions- und Bereichsobjekt rufen zuerst die `deleteObject()` Methoden des enthaltenden Objektes auf, dann deren Destruktor, die Destruktoren der enthaltenden Vektoren und referenzieren danach den Zeiger des enthaltenden Objektes mit Null.

`bool flipObjects(unsigned int n)`

Eingabe: eine Zahl n aus der Ordnung der conc Vertauschpunkte
Rückgabe: true wenn zwei enthaltenden Objekte des conc Objektes, dem die Zahl n in der Ordnung der conc Vertauschpunke zugeordnet ist, vertauscht wurden

Vertauscht die enthaltenden Objekte des conc Objektes, dem die Zahl n in der Ordnung der conc Vertauschpunke zugeordnet ist. Damit kann ein verdecktes Teilobjekt zum verdeckenden Teilobjekt werden und andersherum.

Die conc Objekte ermitteln mit Hilfe des Aufrufes der `getNumberOfConc()` Methode des ersten enthaltenden Objektes die Anzahl (x) der in ihm enthaltenden conc Objekte. Ist die ermittelte Anzahl x größer oder gleich n ($x \geq n$) wird die `flipObjects(n)` Methode, mit dem erhaltenden Wert n, des ersten Objektes aufgerufen und dessen Rückgabewert zurückgegeben. Ist die ermittelte Anzahl x plus eins (für das aktuelle conc Objekt) kleinern n ($x + 1 < n$), wird die `flipObjects(nn)` Methode des zweiten Objektes aufgerufen und dessen Rückgabewert zurückgegeben, dabei ergibt sich der übergebene Wert nn aus dem erhaltenden Wert n, minus dem ermittelten Wert x, minus eins ($nn = n - x - 1$). Sonst ($n == x + 1$) werden die beiden enthaltenden Objekte vertauscht und true zurückgegeben.

Punkte geben false zurück. (Sollte nicht auftreten, wenn n Element der Ordnung der conc Vertauschpunke ist.)

Funktions- und Bereichsobjekt rufen die `flipObjects(n)` Methode des enthaltenden Objektes mit dem erhaltenden Wert n auf und geben dessen Rückgabewert zurück.

`bool moveElement(unsigned int n, int howfar)`

Eingabe: die Position eines Elements in der Ordnung der Verschiebepunkte n und die Anzahl howfar der Elemente, über die es verschoben werden soll (wenn negativ nach oben sonst nach untern)
Rückgabe: true wenn das Element verschoben wurde (um wie viel auch immer, auch 0)

Verschiebt das Element, dem in der Ordnung der Objektpunkte die Zahl n zugeordnet ist, um howfar Schritte, wenn möglich.

Objekte aller Klassen rufen ihre `moveElementTo(n, obj, howfar)` Methode auf, wobei n und howfar die erhaltenden Werte sind und obj ein PictureObject Null Zeiger ist.

`bool moveElementTo(unsigned& int n=0, PictureObject* obj =Null, int& howfar)`

Eingabe: die Position n an der das Objekt obj stand oder steht, in der Ordnung der Verschiebepunkte (wenn 0 ist das Objekt neu), das verschoben werden soll, das Objekt obj das verschoben werden soll (wenn Null wurde das Objekt noch nicht gefunden) und die Anzahl der Schritte howfar, um die das Objekt verschoben werden soll
Rückgabe: true wenn das Element verschoben wurde, sonst false

Verschiebt das Element obj, das an der Position n in der Verschiebeordnung steht, wenn möglich um howfar Schritte nach unten, wenn howfar negativ dann nach oben. (Hilfsfunktion von `moveElement()`)

Wenn howfar gleich 0 ist und obj nicht Null wird das Objekt an der aktuellen Stelle eingefügt. Wenn howfar gleich 0 ist wird immer true zurückgeben.

Wenn howfar größer 0 ist wird das Objekt nach unten verschoben. Dabei muss geprüft werden, ob es in den Objekt, über das es geschoben wird, noch benötigt wird (heißt: dieses eine Variable enthält, die das verschobene Objekt definiert) und wenn ja darf es nicht verschoben werden.

Wenn howfar kleiner 0 ist, wird das Objekt nach oben verschoben. Es muss geprüft werden, ob die Objekte, über das das zu verschiebende Objekt verschoben werden soll, Variablen definieren die das zu verschiebende Objekt noch benötigt. Über solche Objekt darf das zu verschiebende Objekt nicht verschoben werden.

Das zu verschiebende Objekt darf nicht über die Grenzen des ganzen fib-Objektes hinausgeschoben werden.

howfar gleich 0:

Funktions- und Bereichsobjekt prüfen, ob das übergebene Objekt gleich Null ist. Wenn ja existiert kein Objekt das Verschoben werden soll und es wird true zurückgegeben.

Sonst wird das Objekt unter dem aktuellen Objekt eingefügt. Dazu wird das enthaltende Objekt in das übergebene zu verschiebende Objekt, mittels Aufrufes der `insertObject(obje)` Methode des zu verschiebenden Objektes, wobei obje das im aktuellen Objekt enthaltende Objekt ist (Zeiger), eingefügt und das zu verschiebende als das aktuelle enthaltendes gesetzt, dann wird true zurückgegeben.

conc Objekt ist analog zu dem Aufruf bei den Funktions- und Bereichsobjekten, nur die zwei Teilobjekte zu berücksichtigen sind.

Beim einfügen des Objektes, wird geprüft in welchem Objekt die Variable, die das zu verschiebende Objekt definiert, benötigt wird, mit `isUsedVariable(var)`, und dann in die/das Objekt eingefügt, in der sie benötig wird. Wenn sie in Beiden

benötigt wird, wird das zu verschiebende Element mit Hilfe `copyElement(n)` kopiert (nicht das Objekt) und in jeden Zweig eine Kopie eingefügt. Wenn sie in keinen benötigt wird, wird das zu verschiebende Element nur in das erste Objekt eingefügt.

Punkte geben false zurück.

howfar größer 0 (verschieben nach unten):

Bei Funktions- und Bereichsobjekten wird zuerst, mit Hilfe der `getNumberOfMovePoints()` Methode des enthaltenden Objektes, die Anzahl (x) der Verschiebepunkte über dem aktuellen Objekt ermittelt.

Ist die ermittelte Zahl x gleich der übergebenen Zahl n ($x == n$), wird geprüft ob das enthaltende Objekt das zu verschiebende Objekt ist und dieses kein conc Objekt ist. Dafür wird geprüft ob, die `classnameOf()` Methode des enthaltenden Objektes "conc“ zurückgibt.

Wenn es nicht das zu verschiebende Objekt ist, wird die `moveElementTo(n, obj, howfar)` Methode des enthaltenden Objektes mit den erhaltenden Werten aufgerufen und deren Rückgabe zurückgegeben.

Sonst wird der Zeiger auf das zu verschiebende (enthaltende) Objekt gemerkt (in Variable obj1) und mit Hilfe des `getObjectFrom(UnderObject->movePointToObjectPoint(n) - 1)` Methodenaufrufes das Objekt (obj2) direkt unter diesem ermitteln. Dann wird geprüft ob verschieben möglich ist (siehe unten).

Wenn nein wird true zurückgegeben.

Wenn ja wird das ermittelte Objekt obj2 anstelle des alten enthaltenden Objektes gesetzt und das alte enthaltene, zu verschiebende Objekt nach unten verschoben. Dafür wird die Methode `moveElementTo(n, obj1, howfar1)` aufgerufen, wobei obj1 der gemerkte Zeiger auf das zu verschiebende Objekt ist und howfar1 sich aus howfar minus 1 ergibt ($howfar1 = howfar - 1$), da das Objekt jetzt eine Stufe tiefer ist. Wenn true zurückgegeben wird, wird true weiter zurückgegeben, ansonsten wird das Objekt an der aktuellen Stelle, mittels Aufruf der eigenen Methode `moveElementTo(n, obj1, 0)`, eingefügt und deren Rückgabe zurückgegeben.

Ist die ermittelte Zahl x größer als die übergebenen Zahl n ($x > n$), wird die Methode `moveElementTo(n, obj, howfar)` des enthaltenden Objektes aufgerufen, die Stelle direkt über dem zu verschiebenden Objektes liegt noch unter der des aktuellen Objektes.

Ist die ermittelte Zahl x kleiner als die übergebenen Zahl n ($x < n$), soll das Objekt obj weiter nach unten verschoben werden. Dafür wird geprüft, ob verschieben möglich ist (siehe unten).

Wenn ja wird die Methode `moveElementTo(n, obj, howfar1)` aufgerufen, wobei obj der gemerkte, übergebene Zeiger auf das zu verschiebende Objekt ist und howfar1 sich aus howfar minus 1 ergibt ($howfar1 = howfar - 1$), da das Objekt jetzt eine Stufe tiefer ist. Wenn true zurückgegeben wird, wird true weiter zurückgegeben, ansonsten wird das Objekt an der aktuellen Stelle, mittels

Aufruf der eigenen `moveElementTo(n, obj1, 0)` Methode, eingefügt und deren Rückgabe zurückgegeben.

Wenn das verschieben nicht möglich ist, wird das übergebene, gemerkte Objekt obj an der aktuellen Stelle, mittels Aufruf der eigenen `moveElementTo(n, obj, 0)` Methode, eingefügt und deren Rückgabe zurückgegeben.

Die conc Objekte sind analog zu dem Aufruf bei den Funktions- und Bereichsobjekten, nur das die zwei Teilobjekte zu berücksichtigen sind.

Dabei wird mittels Aufruf der Methoden `getNumberOfMovePoints()` der beiden enthaltenden Objekte die Anzahl (x1 und x2) der Verschiebepunkte in ihnen überprüft.

Beim weiterschieben des Objektes nach unten ($x1 + x2 > n$), wird geprüft in welchem Objekt die Variable, die das zu verschiebende Objekt definiert, benötigt wird, mit Hilfe `isUsedVariable(var)`, und dann nur in die/das Objekt verschoben, bzw. versucht zu verschieben und eventuell eingefügt, in der sie benötig wird. Wenn sie in Beiden benötigt wird, wird das zu verschiebende Element, mit Hilfe seiner `copyElement(n)` Methode, kopiert (nicht das Objekt) und an jeden Zweig eine Kopie weitergegeben. Wobei im Zweig, der das kopierte Element bekommt, die Variable welche das zu verschiebende Objekt definiert, auf die Variable gesetzt wird, die das kopierte Element definiert. Wenn das zu verschiebende Objekt in keinen enthaltendem Objekt benötigt wird, wird das zu verschiebende Element nur an das erste Objekt weitergegeben.

Punkte geben false zurück.

Prüfen ob verschieben möglich ist: Mittels Aufrufes der `getDefineVariableFrom(n)` Methode, des zu verschiebenden Objektes (n ist dessen Verschiebepunkt Position), wird die Variable (var) ermittelt, die dieses definiert und mittels Aufrufs der `isUsedVariableInElement(var)` Methode des Objektes (obj2), über das verschoben werden soll, geprüft, ob dieses die ermittelte Variable noch benötigt.

howfar kleiner 0:

Einleitende Anmerkungen: Aktiv ist immer das Objekt zwei Positionen über dem zu verschiebenden Objekt, howfar wird bis auf 0 hochgezählt und n wird jeweils auf die neue Position des zu verschiebenden Objektes gesetzt. Deshalb wird eine Referenz auf howfar und n übergeben. Wenn conc Objekte (als aktives Objekt) verschieben soll und howfar größer 0 ist, das Objekt soll weiter verschoben werden, wird das zu verschiebende Objekt in ihr zweites Objekt verschoben. Dadurch ist es für Objekte darunter egal, das über ein conc verschoben wird, das nächste Objekt in der Objektordnung unter ihm ist das zu verschiebende Objekt, siehe Objektordnung. Es wird jeweils getestet, ob das zu verschiebende Objekt Variablen benötigt, die Objekte definieren, über die es verschoben wird.

Wenn howfar innerhalb dieses Teils größer als 0 wird, heißt dies das Objekt soll nicht weiter verschoben werden.

Bei Funktions- und Bereichsobjekt wird zuerst, mittels der `getNumberOfMove-Points()` Methode des enthaltenden Objektes, die Anzahl (x) der Verschiebepunkte über dem aktuellen Objekt ermittelt.

Ist die ermittelte Zahl x gleich der übergebenen Zahl n ($x == n$), wird geprüft, ob das enthaltende Objekt das zu verschiebende Objekt oder ein conc Objekt ist. Dafür wird geprüft, ob die `classnameOf()` Methode des enthaltenden Objektes "conc" zurückgibt.

Wenn nein, es also das zu verschiebende Objekt ist, wird true zurückgegeben.

Wenn es nicht das zu verschiebende Objekt ist, stehen nur noch conc Objekte zwischen diesem und dem zu verschiebenden Objekt, d.h. es kann nach oben über diese verschoben werden, ohne zu prüfen, ob die Variable des zu verschiebenden Objektes noch benötigt wird.

Dafür wird das zu verschiebende Element mittels `cutElement(n, true)` aus der aktuellen Position ausgeschnitten. Wenn das ausschneiden gelungen ist, wird in ihm das enthaltende Objekt (uobj) mittels `insertObject(uobj)` eingefügt und es als enthaltendes Objekt eingefügt.

Ist die ermittelte Zahl x größer als die übergebenen Zahl n ($x > n$), wird zuerst die `moveElementTo(n, obj, howfar)` Methode des enthaltenden Objektes mit den erhaltenden Werten aufgerufen, um das Objekt an die richtige Position nach oben zu verschieben. Ist dies nicht erfolgreich (Rückgabe false) wird false zurückgegeben.

Ist howfar danach größer 0, ist das zu verschiebende Objekt schon an der richtigen Position und es wird true zurückgegeben.

Sonst wird ermittelt, ob das zu verschiebende Objekt zwei Positionen über dem aktuellen Objekt ist. Dafür wird mittels erneuten Aufruf (n kann sich verändert haben) der Methode `movePointToObjectPoint(n)` (=o) des enthaltenden Objektes, der Objektpunkt (o) das Verschiebepunktes ermittelt und dann mit Hilfe der Zahl y überprüft, die der Aufruf der Methode `getNumberOfObjects()` (=y) des enthaltenden Objektes ermittelt, ob das zu verschiebende Objekt direkt unter dem enthaltendem Objekt ist. Dies ist der Fall, wenn der Objektpunkt des zu verschiebenden Objektes plus eins gleich dem des enthaltendem Objektes ist ($(o + 1) == y$).

Wenn nein hat als letztes ein conc Objekt etwas verschoben. Dann wird das zu verschiebende Objekt über das conc Objekt oder die conc Objekte verschoben.

Dafür wird mittels `cutElement(o, true)` (= obv) das zu verschiebende Element an der alten Stelle ausgeschnitten. Der Zeiger bzw. das zu verschiebende Objekt obv, der vom Aufruf zurück gegeben wird, wird sich gemerkt. Wenn `cut-Element()` erfolgreich war, wird das alte enthaltende Objekt mittels Aufruf der `insertObject(obje)` Methode des Objektes obv in dieses eingefügt, wobei obje das im aktuellen Objekt (alte) enthaltende Objekt ist (Zeiger), und das Objekt obv als das eigene enthaltende Objekt genommen. Dann wird true zurückgegeben.

Wenn `cutElement()` nicht erfolgreich war, wird false zurückgegeben.

Wenn ja ($(o + 1) == y$), wird das im aktuellen Objekt enthaltende Objekt und das Objekt direkt unter diesem, welches das Objekt, das verschoben werden

soll ist, vertauscht. Dafür wird zuerst ein Zeiger auf das zu verschiebende Objekt (obj1) mittels `getObjectFrom(o)` (= obj1) ermittelt. Dann wird geprüft, ob das Objekt verschoben werden kann. Dafür wird mittels Aufrufes `getDefineVariableFrom(y)` (= var) des enthaltenden Objekt, y ist dessen Position in der Objektordnung, wird mit Hilfe von `getNumberOfMovePoints()` des enthaltenden Objektes ermittelt, die Variable (var) ermittelt, die dieses definiert, und mittels Aufrufs der `isUsedVariableInElement(var)` Methode, des zu verschiebenden Objektes (obj1), geprüft, ob dieses die ermittelte Variable noch benötigt.

Wenn ja, wird true zurückgegeben und howfar auf 1 gesetzt. (Keine weiteren Verschiebungen.)

Wenn nein, wird mittels `cutElement(n, true)` (= obv) das zu verschiebende Element an der alten Stelle ausgeschnitten. Wenn dies nicht geling wird false zurückgegeben. Sonst wird der Zeiger bzw. das zu verschiebende Objekt obv, der vom Aufruf zurück gegeben wird, gemerkt. Das alte enthaltende Objekt mittels Aufruf der `insertObject(obje)` Methode des Objektes obv eingefügt, wobei obje das im aktuellen Objekt (alte) enthaltende Objekt ist (Zeiger), und das Objekt obv als das eigenes enthaltendes Objekt genommen.

Es wird howfar um 1 erhöht ($howfar = howfar + 1$), wenn howfar vorher kleiner als minus 1 war (mehr als um eine Position verschieben), wenn howfar vorher -1 war, wird howfar auf 1 gesetzt, da das Objekt fertig verschoben wurde, n wird auf den von der `getNumberOfMovePoints()` des zu verschiebenden Objektes zurückgegebenen Wert gesetzt, dann wird true zurückgegeben.

Die conc Objekte ermitteln, ob sich das zu verschiebende Objekt im ersten oder im zweiten enthaltenden Objekt befindet. Dafür wird `getNumberOfMovePoints()` (= x) des ersten Objekt aufgerufen, ist der erhaltende Wert x größer oder gleich dem übergebende Wert n ($x \geq n$), befindet sich das zu verschiebende Objekt im ersten Objekt sonst im zweiten.

Zu verschiebendes Objekt im ersten Objekt:

Ist die ermittelte Zahl x gleich der übergebenen Zahl n ($x == n$), ist das zu verschiebende Objekt das erste Verschiebeobjekt im ersten Objekt. Deshalb wird dann das zu verschiebende Objekt in das zweite Objekt verschoben.

Beim Verschieben des zu verschiebende Objekt in das zweite Objekt, wird zuerst ein Zeiger auf das zu verschiebende Objekt (obj1) mittels `cutElement(n, true)` (= objv) ermittelt und damit auch das zu verschiebende Objekt ausgeschnitten. Wenn dies nicht erfolgreich ist, wird false zurückgegeben.

Sonst wird mittels Aufrufes der `insertObject(obje)` Methode des Objektes objv, wobei obje das (alte) zweite enthaltende Objekt ist, das enthaltende Objekt in das zu verschiebende eingefügt und dann das zu verschiebende an die Stelle des zweiten enthaltenden Objektes gesetzt.

Es wird n auf die Zahl gesetzt, die sich aus der Summe der Zahlen ergibt, die durch den erneuten Aufruf der `getNumberOfMovePoints()` Methoden der

beiden enthaltenden Objekte ermittelt werden. (Das ist die neue Position des zu verschiebenden Objektes.) Dann wird true zurückgegeben.

Ist die ermittelte Zahl x größer als die übergebene Zahl n ($x > n$), wird zuerst die `moveElementTo(n, obj, howfar)` Methode des enthaltenden Objektes mit den erhaltenden Werten aufgerufen, um das Objekt an die richtige Position nach oben zu verschieben. Ist dies nicht erfolgreich (Rückgabe false) wird false zurückgegeben.

Ist howfar danach größer 0, ist das zu verschiebende Objekt schon an der richtigen Position und es wird true zurückgegeben.

Sonst wird ermittelt, ob das zu verschiebende Objekt zwei Positionen über dem aktuellen Objekt im ersten Objekt ist. Dafür wird mittels erneuten Aufruf (n kann sich verändert haben) der Methode `movePointToObjectPoint(n)` (=o) des ersten enthaltenden Objektes, der Objektpunkt(o) das Verschiebepunktes ermittelt und dann mit Hilfe der Zahl y überprüft, die der Aufruf der Methode `getNumberOfObjects()` (=y) des ersten enthaltenden Objektes ermittelt, ob das zu verschiebende Objekt direkt unter dem enthaltendem Objekt ist. Dies ist der Fall, wenn der Objektpunkt des zu verschiebenden Objektes plus 1 gleich dem des enthaltendem Objektes ist ($(o + 1) == y$).

Wenn nein, hat als letztes ein conc Objekt etwas verschoben. Dann wird das zu verschiebende Objekt über das conc Objekt oder die conc Objekte verschoben. Dafür wird mittels `cutElement(o, true)` (= obv) das zu verschiebende Element an der alten Stelle ausgeschnitten. Der Zeiger bzw. das zu verschiebende Objekt obv, der vom Aufruf zurück gegeben wird, wird sich gemerkt. Wenn `cutElement()` erfolgreich war, wird das zweite, alte enthaltende Objekt, mittels Aufruf der `insertObject(obje)` Methode des Objektes obv, in dieses eingefügt, wobei obje das im aktuellen Objekt zweite (alte) enthaltende Objekt ist (Zeiger) und das Objekt obv als das eigene zweite, enthaltendes Objekt genommen. Dann wird true zurückgegeben.

Wenn cutElement nicht erfolgreich war, wird false zurückgegeben.

Wenn ja, ($(o{+}1) == y$) wird versucht das zu verschiebende Objekt eine Position nach oben zu verschieben. Dafür wird zuerst ein Zeiger auf das zu verschiebende Objekt (obj1) mittels `getObjectFrom(o)` (= obj1) des ersten enthaltenden Objektes ermittelt.

Dann wird geprüft, ob das Objekt verschoben werden kann. Dafür wird mittels Aufrufes `getDefineVariableFrom(y)` (= var) des ersten enthaltenden Objekt, y ist dessen Position in der Objektordnung, wird mit Hilfe von `getNumberOfMovePoints()` des ersten enthaltenden Objektes ermittelt, die Variable (var) ermittelt, die dieses definiert, und mittels Aufrufs der `isUsedVariableInElement(var)` Methode, des zu verschiebenden Objektes (obj1), geprüft, ob dieses die ermittelte Variable noch benötigt.

Wenn ja, wird true zurückgegeben und howfar auf 1 gesetzt. (Keine weiteren Verschiebungen.)

Wenn nein, wird mittels `cutElement(n, true)` (= obv) das zu verschiebende Element an der alten Stelle ausgeschnitten. Wenn dies nicht geling, wird

false zurückgegeben.

Sonst wird der Zeiger bzw. das zu verschiebende Objekt obv, der vom Aufruf zurück gegeben wird, gemerkt.

Wenn howfar gleich minus eins ist (nur eins nach oben verschieben), wird das erste, alte enthaltende Objekt (obje), mittels Aufruf der `insertObject(obje)` Methode des Objektes obv, in das zu verschiebende Objekt eingefügt, wobei obje das im aktuellen Objekt (alte) erste enthaltende Objekt ist (Zeiger), und das Objekt obv als das eigenes erstes enthaltendes Objekt genommen. Es wird howfar auf 1 gesetzt, da das Objekt fertig verschoben wurde, n um eines erhöht und true zurückgegeben.

Sonst (howfar ungleich minus 1) , wird das zweite alte enthaltende Objekt mittels Aufruf der `insertObject(obje)` Methode des Objektes obv eingefügt, wobei obje das im aktuellen Objekt (alte) zweite enthaltende Objekt ist (Zeiger) und das Objekt obv als das eigene, zweite enthaltende Objekt genommen. Es wird howfar um eins erhöht und n wird auf die Summe der von der `getNumberOfMovePoints()` Methoden der enthaltenden Objekte zurückgegebenen Werte gesetzt. Dann wird true zurückgegeben.

Zu verschiebendes Objekt im zweiten Objekt:

Es wird, mittels der `getNumberOfMovePoints()` Methode des zweiten enthaltenden Objektes, die Anzahl (x_2) der Verschiebepunkte im zweitem Objekt ermittelt. Dann ergibt sich das neue x aus dem alten x plus x_2 ($x = x + x_2$) und n_1 ergibt sich aus n minus den altem x (also der Verschiebepunkte im ersten Objekt) ($n_1 = n - x$).

Ist die ermittelte Zahl x gleich der übergebenen Zahl n ($x == n$, das nächste verschiebe Objekt im zweiten enthaltenden Objekt ist das zu verschiebendes Objekt), wird true zurückgegeben.

Ist die ermittelte Zahl x größer als die übergebenen Zahl n ($x > n$), wird die `moveElementTo(n, obj, howfar)` Methode des enthaltenden Objektes mit den erhaltenden Werten aufgerufen, um das Objekt an die richtige Position nach oben zu verschieben. Ist dies nicht erfolgreich (Rückgabe false) wird false zurückgegeben.

Ist howfar danach größer 0, ist das zu verschiebende Objekt schon an der richtigen Position und es wird true zurückgegeben.

Sonst wird ermittelt, ob das zu verschiebende Objekt zwei Positionen über dem aktuellen Objekt ist. Dafür wird mittels erneuten Aufruf (n kann sich verändert haben) der Methode `movePointToObjectPoint(n)` (= o) des zweiten enthaltenden Objektes, der Objektpunkt (o) das Verschiebepunktes ermittelt und dann mit Hilfe der Zahl y überprüft, die der Aufruf der Methode `getNumberOfObjects()` (= y) des zweiten enthaltenden Objektes ermittelt, ob das zu verschiebende Objekt direkt unter dem enthaltendem Objekt ist. Dies ist der Fall, wenn der Objektpunkt des zu verschiebenden Objektes plus eins gleich dem des enthaltendem Objektes ist ($(o + 1) == y$).

Wenn nein, hat als letztes ein conc Objekt etwas verschoben. Dann wird true zurückgegeben, da das zu verschiebende Objekt auch für dieses conc Objekt noch das erste Verschiebeobjekt ist (das mit der größten Zahl in der Verschiebeordnung).

Wenn ja ($(o+1) == y$), wird das im aktuellen Objekt enthaltende zweiten Objekt und das Objekt direkt unter diesem, welches das Objekt ist, das verschoben werden soll, vertauscht. Dafür wird zuerst ein Zeiger auf das zu verschiebende Objekt (obj1) mittels `getObjectFrom(o)` (= obj1) des zweiten enthaltenden Objektes ermittelt. Dann wird geprüft, ob das Objekt verschoben werden kann. Dafür wird mittels Aufrufes `getDefineVariableFrom(y)` (= var) des zweiten enthaltenden Objekt, y ist dessen Position in der Objektordnung, wird mit Hilfe von `getNumberOfMovePoints()` des zweiten enthaltenden Objektes ermittelt, die Variable (var) ermittelt, die dieses definiert, und mittels Aufrufs der `isUsedVariableInElement(var)` Methode, des zu verschiebenden Objektes (obj1), geprüft, ob dieses die ermittelte Variable noch benötigt.

Wenn ja, wird true zurückgegeben und howfar auf 1 gesetzt. (Keine weiteren Verschiebungen.)

Wenn nein, wird, mittels `cutElement(n, true)` (= obv) des zweiten enthaltenden Objektes, das zu verschiebende Element an der alten Stelle ausgeschnitten. Wenn dies nicht geling, wird false zurückgegeben. Sonst wird der Zeiger bzw. das zu verschiebende Objekt obv, der vom Aufruf zurück gegeben wird, gemerkt. Das alte enthaltende Objekt wird, mittels Aufruf der `insertObject(obje)` Methode des Objektes obv, in das zu verschiebende Objekt obv eingefügt, wobei obje das im aktuellen Objekt zweite (alte) enthaltende Objekt ist (Zeiger), und das Objekt obv als das eigenes zweites enthaltendes Objekt genommen.

Es wird howfar um eins erhöht ($howfar = howfar + 1$), wenn howfar vorher kleiner als minus eins war (mehr als um eine Position verschieben), wenn howfar vorher minus 1 war, wird howfar auf 1 gesetzt, da das Objekt fertig verschoben wurde, n wird auf die Summe der von den `getNumberOfMovePoints()` Methoden der enthaltenden Objekte zurückgegebenen Werte gesetzt. Dann wird true zurückgegeben.

Punkte liefern false zurück.

`bool insertObject(PictureObject* obj)`

Eingabe: das einzufügende Objekt obj
Rückgabe: true wenn das Objekt eingefügt wurde, sonst false

Fügt das gegebene Objekt obj in das Objekt ein.
Achtung: Das überschriebene Objekt wird nicht gelöscht!

Die conc Objekte und Punktobjekte geben false zurück, in sie kann kein einzelnes Objekt eingefügt werden.

Funktions- und Bereichsobjekt setzen den Zeiger ihres enthaltendes Objekt auf den übergebenen Wert obj.

bool insertFunctionVector(unsigned int n, UnderFunction* fkt)

Eingabe: eine Zahl, die dem Funktionsobjekt in der Ordnung der Funktionsobjekte zugeordnet ist, in die ein der Vektor eingefügt werden soll und der Zeiger auf den Vektor fkt der eingefügt werden soll
Rückgabe: true wenn der Vektor eingefügt wurde, sonst false

Fügt einen Vektor in das Funktionsobjekt ein, dem die Zahl n in der Funktionsordnung zugeordnet ist.
Achtung: Der Vektor wird nicht auf Gültigkeit geprüft! Es wird also nicht geprüft ob eventuell enthaltende Variablen über dem Objekt definiert werden.

Die conc Objekte ermitteln mittels Aufruf der `getNumberOfFunction()` Methode die Anzahl (x) der Funktionsobjekte in ihrem ersten Objekt. Ist der erhaltende Wert x größer oder gleich dem übergebenen Wert n ($x \geq n$), wird die `insertFunctionVector(n,fkt)` des ersten Objektes mit dem erhaltenden Werten aufgerufen und dessen Rückgabe zurückgegeben. Sonst wird die `insertFunctionVector(nn,fkt)` des zweiten Objektes mit dem erhaltenden Wert fkt und dem ermittelten Wert nn aufgerufen und dessen Rückgabe zurückgegeben. Dabei ergibt sich der übergebene Wert nn aus dem erhaltenden Wert n minus dem ermittelten Wert x ($nn = n - x$).

Punkte geben false zurück. Denn Punkte sind keine Funktionsobjekte und enthalten auch keine.

Funktionsobjekte ermitteln die Anzahl (x), mit `getNumberOfFunction()`, der Function Objekte im enthaltenden Objekte. Ist diese Anzahl plus eins (für sich selbst) gleich dem erhaltendem Wert n ($x+1 == n$) wird der Vector fkt in die eigene Liste eingefügt und true zurückgegeben. Sonst wird die `insertFunctionVector(n, fkt)` Methode des enthaltenden Objektes mit den erhaltenden Werten aufgerufen und dessen Rückgabe zurückgegeben.

Bereichsobjekt rufen die `insertFunctionVector(n, fkt)` des enthaltenden Objektes mit den erhaltenden Werten auf und geben dessen Rückgabe zurück.

bool insertAreaVector(unsigned int n, UnderArea* area)

Eingabe: eine Zahl, die dem Bereichsobjekt in der Ordnung der Bereichsobjekte zugeordnet ist, in die ein der Vektor eingefügt werden soll und der Zeiger auf den Vektor area der eingefügt werden soll
Rückgabe: true wenn der Vektor eingefügt wurde, sonst false

Fügt einen Vektor in das Bereichsobjekt ein, dem die Zahl n in der Funktionsordnung zugeordnet ist.
Achtung: Der Vektor wird nicht auf Gültigkeit geprüft! Es wird also nicht geprüft ob eventuell enthaltende Variablen über dem Objekt definiert werden.

Die conc Objekte ermitteln mittels Aufruf der `getNumberOfArea()` Methode die Anzahl (x) der Bereichsobjekte in ihrem ersten Objekt. Ist der erhaltende Wert x größer oder gleich dem übergebenen Wert n ($x \geq n$), wird die `insertAreaVector(n,area)` des ersten Objektes mit dem erhaltenden Werten aufgerufen und dessen Rückgabe zurückgegeben. Sonst wird die `insertAreaVector(nn, area)` des zweiten Objektes mit dem erhaltenden Werten area und dem ermittelten Wert nn aufgerufen und dessen Rückgabe zurückgegeben. Dabei ergibt sich der übergebene Wert nn aus dem erhaltenden Wert n minus dem ermittelten Wert x ($nn = n - x$).

Punkte geben false zurück. Denn Punkte sind keine Bereichsobjekte und enthalten auch keine.

Bereichsobjekte ermitteln die Anzahl (x), mit `getNumberOfArea()`, der Area Objekte im enthaltenden Objekte. Ist diese Anzahl plus eins (für sich selbst) gleich dem erhaltendem Wert n ($x + 1 == n$) wird der Vektor in die eigene Liste eingefügt und true zurückgegeben. Sonst wird die `insertAreaVector(n, fkt)` Methode des enthaltenden Objektes mit den erhaltenden Werten aufgerufen und dessen Rückgabe zurückgegeben.

Funktionsobjekt rufen die `insertAreaVector(n, fkt)` des enthaltenden Objektes mit den erhaltenden Werten auf und geben dessen Rückgabe zurück.

Allgemeine Methoden

`PictureObject* copy(unsigned int n)`

Rückgabewert: eine Kopie des Teilobjektes, dem die Zahl n in der Teilobjektordnung zugeordnet ist
Eingabe: die Position n in der Teilobjektordnung, von der das Teilobjekt kopiert werden soll

Kopiert das Teilobjekt dem die Zahl n in der Teilobjektordnung zugeordnet ist.

Die conc Objekte ermittelt mit Hilfe der `getNumberOfObjectPoints()` Methode die Anzahl (x) der Teilobjekte in seinem ersten Objekt. Ist die Anzahl (x) plus eins gleich n ($n == x + 1$), dann wird die `copy()` Methode des ersten Objekt ausgeführt und dessen Rückgabe zurückgegeben. (Das aktuelle conc Objekt entfällt im neuem Objekt.) Ist die Anzahl (x) plus zwei (die beiden eigenen Teilobjekte) gleich n ($n == (x + 2)$), wird die `copy()` Methode des zweiten Objektes ausgeführt und dessen Rückgabe zurückgegeben. Ist der übergebene Wert kleiner als die Anzahl (x) ($n < x$), wird die `copy(n)` Methode des ersten Objektes mit den erhaltenden Werten aufgerufen und dessen Rückgabewert zurückgegeben.

Sonst wird die `copy(nn)` Methode des zweiten Objektes mit den Wert nn aufgerufen und dessen Rückgabewert zurückgegeben, nn ergibt sich dabei aus dem übergebenen Wert n minus der Anzahl (x) minus zwei ($nn = n - x - 2$).

Funktions- und Bereichsobjekt rufen die `copy(n)` Methode des enthaltenden Objektes auf. Wenn `copy(n)` kein Null Zeiger zurückgibt, wird der eigene Parameterkonstruktor mit den Werten der eigenen Objekte aufgerufen (die Vektoren

werden vorher copiert), außer dem eigenen enthaltenden Objekt, für welches das von `copy(n)` zurückgelieferte Objekt eingesetzt wird. Das erzeugte Objekt wird zurückgegeben. Sonst, wenn Null zurückgegeben wurde, wird Null zurückgegeben.

Punktobjekte geben einen Null Zeiger zurück, da sie kein Teilobjekt sind und auch Keins enthalten das kopiert werden kann.

`PictureObject* copyElement(unsigned int n)`

Eingabe: die Position n in der Objektordnung von der das Element kopiert werden soll
Rückgabewert: eine Kopie des Elements, dem die Zahl n in der Objektordnung zugeordnet ist (hat keine enthaltenden Objekte)

Liefert eine Kopie des Elements, dem die Zahl n in der Objektordnung zugeordnet ist.

Die conc Objekte ermittelt mit Hilfe ihrer `getNumberOfObjects()` Methode die Anzahl (x) der Objekte in seinem ersten Objekt. Ist die Anzahl (x) plus eins (für sich selbst) gleich n ($n == x + 1$), dann wird der eigene Standartkonstructor aufgerufen und das erzeugte Objekt zurückgegeben. Ist der übergebene Wert n kleiner oder gleich der Anzahl (x) ($n < x$), wird die `copyElement(n)` Methode des ersten Objektes mit den erhaltenden Werten aufgerufen und dessen Rückgabewert zurückgegeben. Sonst wird die `copyElement(nn)` Methode des zweiten Objektes mit den Wert nn aufgerufen und dessen Rückgabewert zurückgegeben, nn ergibt sich dabei aus dem übergebenen Wert n minus der Anzahl (x) minus eins ($nn = n - x - 1$).

Punkte rufen ihren eigenen copy Konstruktor auf. (Haben keine enthaltenden Objekte.)

Funktions- und Bereichsobjekt ermittelt mit Hilfe der Methode `getNumberOfObjects()` die Anzahl (x) der Objekte im enthaltendem Objekt. Ist die Anzahl (x) plus eins (für sich selbst) gleich n ($n == x + 1$), dann wird der eigene Konstruktor aufgerufen und ein Zeiger auf das erzeugte Objekt zurückgegeben.

Sonst wird die `copyElement(n)` Methode des enthaltenden Objektes mit den erhaltenden Werten aufgerufen und dessen Rückgabewert zurückgegeben. Als verwendeter Konstruktor findet hierbei der parametrisierte Konstruktor Anwendung, dem als einzufügendes enthaltendes Objekt ein Null Zeiger übergeben wird und als Vektorliste eine Liste mit den kopierten eigenen Vektoren.

`void makeMatrix(PicturMatrix *pm)`

Eingabe: einen Zeiger auf die Picturmatrix pm in welche die Bildmatrix des Objektes geschrieben werden soll
Rückgabewert: die entsprechende Matrix des Bildes in der PicturMatrix pm, das vom Objekt erzeugt wird

Erzeugt die Bildmatrix des Objektes.

Die conc Objekte rufen zuerst die `makeMatrix(pm)` Methode ihres zweiten Objektes (Zweiges) und dann die `makeMatrix(pm)` ihres ersten Objektes auf. Das Ergebnis ist dann auch gleich vereinigt, wobei das Ergebnis, welches das erste enthaltende Objekt zurückgeliefert hat, das Ergebnis des zweiten enthaltenden Objektes eventuell überdeckt.

Punkte ermitteln die Koordinate mit den gegebenen eigenen Positionswerten und tragen dann den ermittelten Farbwert, in die übergebene Matrix pm, an dieser Koordinate ein. Falls die Koordinate noch innerhalb der Matrix liegt, werden sie aufgenommen.

Funktionsobjekt ermitteln, mit Hilfe der schon gegebenen Werte, den Wert der Variable, welche sie definieren, dazu werden die UnderFunction Vektoren in der List berechnet und aufsummiert, und rufen dann die `makeMatrix(pm)` Methode des enthaltenden Objektes auf.

Bereichsobjekte durchlaufen mittels for-Schleifen die Bereiche ihrer Bereichsvektoren, für jede Zahl, die in den Bereichen der Bereichvektoren liegt, wird dabei ein Durchlauf gemacht. Bei jedem Durchlauf wird die eigene Variable, die definiert wird, auf die Zahl gesetzt, für die der Durchlauf gemacht wird. Danach wird im Durchlauf die `makeMatrix(pm)` Methode des enthaltenden Objektes aufgerufen, wobei die dabei erzeugten Punkte in der Matrix eventuell Punkte der alten Matrix Überdeckt.

`void makeMatrixOfObject(PicturMatrix *pm, unsigned int n)`

Eingabe: die Zahl n des Teilobjektes für das die Bildmatrix erzeugt werden soll und einen Zeiger auf die PicturMatrix pm in welche die Bildmatrix geschrieben werden soll
Rückgabewert: die entsprechende Matrix des Bildes in der PicturMatrix pm, das vom Objekt erzeugt wird, dem die Teilobjektzahl n zugeordnet ist

Erzeugt die Bildmatrix des Teilobjektes, dem in der Teilobjektordnung die Zahl n zugeordnet ist.

Die conc Objekte rufen zuerst die `getNumberOfObjectPoints()` Methode seines ersten Objektes auf, um die Anzahl (x) der Teilobjekte in diesem zu bestimmen.

Ist die Anzahl x größer als der übergebene Wert n ($x > n$), ruft conc die `makeMatrixOfObject(pm, n)` Methode seines ersten Objektes (Zweiges) auf.

Ist die Anzahl plus zwei (für die eigenen beiden enthaltenden Teilobjekt) kleiner als der übergebene Wert n ($(x + 2) < n$), ruft conc die `makeMatrixOfObject(pm, nn)` seines zweiten Objektes auf, wobei der Übergebene Wert nn gleich dem erhaltendem Wert n minus zwei, minus der Anzahl x der Teilobjekte im ersten Objekt ist ($nn = n - 2 - x$). Ist Anzahl x plus eins gleich dem übergebenen

Wert n ($x + 1 == n$), ruft conc die `makeMatrix(pm)` Methode seines ersten Objektes (Zweiges) auf, da von diesem die Matrix ermittelt werden soll.

Ist die Anzahl plus zwei gleich dem übergebenen Wert n ($x + 2 == n$), ruft conc die `makeMatrix(pm)` ihres zweiten Objektes auf, da von diesem die Matrix ermittelt werden soll.

Punkte ermitteln die Koordinate mit den gegebenen Werten und tragen dann den ermittelten Farbwert, in die übergebene Matrix pm, an dieser Koordinate ein. Falls die Koordinate noch innerhalb der Matrix liegt, werden sie aufgenommen.

Funktionsobjekt ermitteln, mit Hilfe der schon gegebenen eigenen Werte, den Wert der Variable, die sie definieren, die UnderFunction Vektoren in der List werden berechnet und aufsummiert, und rufen dann die `makeMatrixOfObject(pm, n)` Methode des enthaltenden Objektes auf.

Bereichsobjekte durchlaufen mittels for-Schleifen die Bereiche ihrer Bereichsvektoren, für jede Zahl, die in den Bereichen der Bereichvektoren liegt, wird dabei ein Durchlauf gemacht. Bei jedem Durchlauf wird die eigene Variable, die definiert wird, auf die Zahl gesetzt, für die der Durchlauf gemacht wird. Danach wird im Durchlauf die `makeMatrixOfObject(pm, n)` Methode des enthaltenden Objektes aufgerufen, wobei die dabei erzeugten Punkte in der Matrix eventuell Punkte der alten Überdeckt.

`void makeMatrixOfPoint(PicturMatrix *pm ,unsigned int n)`

Eingabe: die Zahl n des Punktteilobjektes, für das die Bildmatrix erzeugt werden soll und einen Zeiger auf die PicturMatrix pm in welche die Bildmatrix geschrieben werden soll
Rückgabewert: die entsprechende Matrix des Bildes in der PicturMatrix pm, das vom Punktobjekt erzeugt wird, dem die Punktteilobjektzahl n zugeordnet ist

Liefert die Matrix zurück, die durch das Punktteilobjekt erzeugt wird, dem die Zahl n in der Punktobjektordnung zugeordnet ist.

Es wird die eigene `pointPartObjectToPartObject(n)` Methode aufgerufen und mit deren Rückgabe (nn) die eigene `makeMatrixOfObject(pm, nn)` Methode aufgerufen.

`PicturObject* restoreElementTyp(stream& in)`

Eingabe: der stream in von dem das erste Element erstellt werden soll
Rückgabe: das neu erstelle Objekt

Stellt ein leeres Objekt vom gleichem Typ her wie das erste Objekt im stream in. Steht im stream als erstes z.B. ”conc(“ wird ein leeres (ohne Unterobjekte) conc Objekt erstellt.

Dient als Hilfsmethode der `restore()` Methode.

Die ersten Zeichen, die den Typ des nächsten Objects angeben und die öffnende Klammer, werden aus den stream in gelesen, danach wird der entsprechende Standardkonstruktor aufgerufen und ein Zeiger auf das erzeugte Objekt zurückgegeben.

15.3.3 Zusätzliche Methoden der Vector Klasse

Die zusätzlichen Methoden der Vector Klasse dienen dazu, eine Möglichkeit zu schaffen, die einzelnen Komponenten eines VektorObjektes gezielt verändern und abfragen zu können.

Abfragemethoden

`unsigned int getNumberOfComponents()`

Rückgabe: Anzahl der Komponenten im Vector Objekt

Liefert die Anzahl der Komponenten des Vector Objektes zurück.

`real* getComponentPointer(unsigned int n)`

Eingabe: eine Zahl n, für die Komponenten, von welcher der `Value` Zeiger zurückgegeben werden soll
Rückgabe: einen Zeiger auf den `Value` Zeiger der n'te Komponente des Vector Objektes

Gibt den `Value` Zeigers der n'te Komponenten des Vector Objektes zurück.

`bool isComponentValue(unsigned int n)`

Eingabe: eine Zahl n, für die Komponente, von welcher der `IsValue` Wert zurückgegeben werden soll
Rückgabe: der `IsValue` Wert der n'ten Komponente

Gibt den `IsValue` Wert der n'ten Komponente des Vector Objektes zurück.

Verändernde Methoden

`bool setComponentPointer(unsigned int n, real* ptr)`

Eingabe: eine Zahl n, für die Komponente, von welcher der `Value` Zeiger auf den übergebenen Zeiger ptr gesetzt werden soll und der Zeiger ptr auf den er gesetzt werden soll
Rückgabe: true wenn die n'te Komponente `Value` Zeiger auf den Zeiger ptr gesetzt wurde

Setzt den `Value` Zeiger der n'te Komponente auf den übergebenen Zeiger.

bool setIsComponentValue(unsigned int n,bool val)

Eingabe: eine Zahl n, für die Komponente, dessen `IsValue` Wert auf den übergebenen Wert val gesetzt werden soll und der Wert val auf den er gesetzt werden soll
Rückgabe: true wenn das Setzen erfolgreich war, sonst false

Setzt den `IsValue` Wert der n'ten Komponente des Vectors auf den übergebenen Wert val.

15.3.4 Zusätzliche Methoden der Color Vektorklassen

Allgemeine Methoden

Color* copyValue()

Rückgabe: eine Kopie des Color Objektes in denen alle Variablen durch die Werte, die sie enthalten, ersetzt worden sind

Liefert eine Kopie des Color Objektes in denen alle Variablen durch die Werte, die sie enthalten, ersetzt worden sind. Diese Methode dient dazu, dass Kopien von Color Objekten in eine PicturMatrix eingesetzt werden können, denn Color Objekte in der PicturMatrix dürfen keine Variablen enthalten.

15.3.5 Zusätzliche Methoden der Klasse Point

Abfragemethoden

Color* getColor()

Rückgabe: ein Zeiger auf das eigene Color Objekt

Liefert ein Zeiger auf das eigene Color Objekt zurück.

Position* getPosition()

Rückgabe: ein Zeiger auf das eigene Position Objekt

Liefert ein Zeiger auf das eigene Position Objekt zurück.

15.3.6 Zusätzliche Methoden der Klasse ListObject

Abfragemethoden

list<Vector*>* getVectorList()

Rückgabe: ein Zeiger auf die eigene Vektorliste

Liefert ein Zeiger auf die eigene Vektorliste zurück.

15.3.7 Realisierung der genetischen Operatoren mit der Klasse Environment

Die genetischen Operationen und der genetische Algorithmus werden in einer Klasse Environment realisiert.

Zentraler Bestandteil der Klasse Environment ist eine Liste von Individuen (IndividualList). Die Parameter werden mit Hilfe eines Parameterobjektes (ParameterV) verwaltet. Diese werden bei der Initialisierung der Klasse übergeben. Des weiteren gibt es noch einige Hilfsvariablen für den Algorithmus.

Die einzelnen Operatoren, die auf Individuen ausführbar sind, werden hier nicht näher beschrieben. Für nähere Informationen sei auf den Quelltext der Klasse Environment verwiesen. Die Operatoren sind für eine ständige Erweiterung und Veränderung angelegt, wenn ich sie hier näher erläutern würde, wären die Ausführungen schnell veraltet und das Risiko wäre groß das von falschen Vorraussetzungen ausgegangen wird. Die Informationen im Quelltext allerdings sollten immer aktuell sein. Indem ich also für die genaue Realisierung der Operatoren auf den Quelltext verweise, wird die Wahrscheinlichkeit geringer veraltete Informationen zu erhalten. Die Operationen gehören auch weniger zum Basissystem der Realisierung, sondern sind mehr optionale Erweiterungen.

Konstruktoren

Die Klasse Environment verfügt über einen Standardconstructor und einen Parameterconstructor.

Der Standardconstructor initialisiert die Variablen mit Standardwerten. Der Parameterconstructor erhält als Eingabe ein Parameterobjekt. Das Environment wird dann mit den Werten des Parameterobjekt, daraus ergebenen Werten und Standardwerten initialisiert.

Methoden

Die Methode `start()`

Die Methode `start()` beinhaltet den typischen Ablauf eines genetischen Algorithmus, wie im Listing 15.1 zu sehen.

Listing 15.1: Allgemeiner Algorithmus

```
1 initialisieren der Menge
2 loop
3   Verändern der potentiellen Lösungen in der Menge
4   Selektion von potentiellen Lösungen
5 until (terminal condition)
```

Dabei wird das ”initialisieren der Menge“ durch die Methode `init()` übernommen.

Das ”Verändern der potentiellen Lösungen in der Menge“ übernimmt die Methode `makeNewIndividual()`. Durch sie wird immer nur ein Individuum durch eine Operation erzeugt und dieses, wenn es noch nicht in der Menge ist, in die Menge eingefügt. Die ”Selektion von potentiellen Lösungen“ geschieht durch die Methode `eraseIndividual()`, die so viele Individuen löscht, wie in der Menge zuviel sind, also auch eventuell keines.

Ob die ”terminal condition“ zutrifft, wird durch die `reachTerminalCondition()` ermittelt.

Der späteren Änderung dieses Ablaufes steht allerdings nichts im Wege, z.B. wenn nicht nur ein Individuum erzeugt werden soll, sondern viele.

Initialisierungsmethoden mit der Methode `init()`

Die Methode `init()` initialisiert die Menge.

Dabei kann mit Hilfe der Init Parameter bestimmt werden, wie viele fib-Objekte für neue Individuen mit welcher Initialisierungsart initialisiert werden können.

Weitere Initialisierungsarten können hier eingefügt werden.

Der Aufbau einer Schleife für eine Initialisierungsart (n), sollte sich an dem im Listing 15.2 dargestellten Schema orientieren.

Listing 15.2: Aufbau einer Initialisierungsartsschleife

```
1 for (unsigned long i = 0;i < (ParameterV.getInitParameter(n)); i
      ++)
2 {
3         **ein neues PicturObject obj erstellen**
4         insertObject(obj);//erzeugtes Objekt einfügen
5 }
```

Dabei gibt `ParameterV.getInitParameter(n)` an, wie viele solcher Individuen erzeugt werden sollen.

Terminal condition Auswerten mit der Methode `reachTerminalCondition()`

Die Methode `terminalCondition()` wertet die Terminalbedingung mit Hilfe der Parameter aus. Sie liefert true wenn die Terminalbedingung zutrifft.

fib-Objekte als neue Individuen laden mit der methode `load(const char* file,const unsigned int num)`

Mit der Methode `load()` können eine Anzahl von n fib-Objekten aus einer Datei (file) als Individuen in die Individuenliste des Objektes geladen werden.

Damit können alte fib-Objekte, die schon generiert und ausgegeben wurden, in eine Datei gespeichert werden (nur die fib-Objekte) und später, für einen neuen Durchlauf, wieder geladen werden.

Veränderungsmethode mit der Methode `makeNewIndividual()`

Erzeugt neue Individuen mit Hilfe der Operatoren Methoden.

Die Auswahl der entsprechenden Operation erfolgt mit Hilfe von if-Blöcken, einer zufällig erzeugten Zahl und der Operatorenparameter.

Die Operatoren werden, je nach den zugehörigen Parametern, mit einer bestimmten Wahrscheinlichkeit (0 bis 1) ausgewählt werden. Bildlich kann man sich vorstellen, das diese Wahrscheinlichkeiten für einen Operator einen entsprechenden Anteil (Kuchenstück) einer Kreisscheibe belegen, z.B. bei der Wahrscheinlichkeit 0,3, einen 0,3 großen Anteil. Dann wird mit der Zufallszahl zufällig, über der ganzen Kreisscheibe ein Punkt gewählt und die entsprechende Operation ausgewählt.

Weiterhin verfügt die Methode über die Möglichkeit, die Auswahl der Methode anzeigen zu lassen. Dies geschieht wenn der Dokumentparameter 1 (`getDocumentParameter(1)`) true ist.

Für jede weitere Operation muss diese Methode nach dem im Listing 15.3 dargestellten Schema erweitert werden.

Listing 15.3: Erweiterung für die makeNewIndividual() Methode

```
1 if (rand<=0){break;} //fertig mit der Operatorenauswahl?
2 if (rand<=ParameterV.getOperationsParameter(10)){
3    if (ParameterV.getDocumentParameter(1)) //Auswahl anzeigen?
4        {cout<<''Operation0``<<endl<<flush;}
5    obj=Operation0(sobj);} //Name der auszuführenden Operation,
6                           //Operation ausführen
7 rand-=ParameterV.getOperationsParameter(10); //drehen der Scheibe
8        //so dass die nächste Operation bei 0 beginnt
```

Operatoren Methoden

Die Operatoren Methoden realisieren die genetischen Operationen auf dem übergebenen PicturObject der Individuen.

Für jede weitere Operation muss eine neue Methode nach dem im Listing 15.4 dargestellten Schema erstellt werden und ihr Kopf in den Header der Klasse Environment eingefügt werden.

Listing 15.4: Schema für neue Operationen

```
1 PicturObject* Environment::Operation(PicturObject* obj)
2                             //Operationsname
3 //Beschreibung
4 {
5 **PicturObjekt obj nach Belieben verändern und zurückgeben,**
6 **wenn das erzeugte Objekt obj nicht in die Individuenmenge**
7 **eingefügt werden soll, Objekte obj löschen und 0 zurückgeben**
8 }
```

Realisiert werden sollen zunächst die Operationen die schon in Teil III angesprochen wurden.

Die verwendeten Operatoren können in der Environment Klasse eingesehen werden.

Auswahl von Individuen mit der Methode `selectIndividual()`

Rückgabe: Zeiger auf ein Individuum aus der Individuenliste

Diese Methode wählt nach einen Auswahlmechanismus ein Individuum der Individuenliste aus und gibt einen Zeiger auf dieses zurück. Der Standardauswahlmechanismus wählt ein Individuum zufällig aus, wobei die Wahrscheinlichkeit, dass ein Individuum ausgewählt wird, proportional zu dessen Fitness/Goodness ist.

Es können hier aber auch weitere Auswahlmechanismen spezifiziert werden, die dann in Abhängigkeit vom Operationsparameter 1 ausgewählt werden, z.B. wenn der Operationsparameter 1 gleich 0 ist, wird die Standardmethode benutzt.

Löschen von Individuen mit der Methode `eraseIndividual()`

Diese Methode wählt nach einen Auswahlmechanismus ein Individuum der Individuenliste aus und löscht dieses. Dies wird solange wiederholt, bis die Individuenliste die über Parameter vorgegebene Maximalgröße wieder erreicht hat.

Der Standardauswahlmechanismus wählt ein Individuum zufällig aus, wobei die Wahrscheinlichkeit, dass ein Individuum ausgewählt wird, umgekehrt proportional zu dessen Fitness/Goodness ist.

Es können hier aber auch weitere Auswahlmechanismen spezifiziert werden, die dann in Abhängigkeit vom einem Operationsparameter ausgewählt werden, z.B. wenn der Operationsparameter 2 gleich 0 ist, wird die Standardmethode benutzt.

Wenn das Individuum gelöscht wird, müssen die abhängige Hilfsvariablen des Environment Objektes angepasst werden.

Einfügen von Individuen mit der Methode `insertIndividual(Individual* idv)`

Eingabe: das Individuum welches eingefügt werden soll
Rückgabe: true, wenn das übergebene Individuum in die Individuenliste eingefügt wurde, sonst false

Fügt ein Individuum, das noch nicht in der Individuenliste ist, in diese ein. Dafür muss überprüft werden, ob ein gleiches Individuum schon in der Individuenliste ist, wenn ja, wird das übergebene Individuum nicht eingefügt.

Wenn das Individuum eingefügt wird, müssen die abhängige Hilfsvariablen des Environment Objektes angepasst werden.

In dieser Methode werden auch, abhängig von den Parametern, das beste Individuum und andere Informationen ausgegeben.

Einfügen von Objekten als Individuen mit der Methode `insertObject( PicturObject* obj)`

Eingabe: ein PicturObject Objekt welches als neues Individuum in die Individuenliste eingefügt werden soll
Rückgabe: true, wenn das übergebene Objekt eingefügt wurde, sonst false

Erzeugt aus dem übergebenen PicturObject ein Individual Objekt und versucht dieses, unter Zuhilfenahme der `insertIndividual()` Methode, in die Individuenliste einzufügen. Schlägt das Einfügen fehl, wird das Individuum wieder gelöscht. Wenn das Individuum eingefügt wird, müssen die abhängige Hilfsvariablen des Environment Objektes, die noch nicht angepasst wurden von der Methode `insertIndividual()`, angepasst werden.

Ermitteln der Fitness eines PicturObjektes `getGoodness(PicturObject* obj)`

Eingabe: ein PicturObject für das die Goodness (Fitness) ermittelt werden soll
Rückgabe: die ermittelte Goodness des übergebenen PicturObjects

Ermittelt die Goodness eines übergebenen PicturObjects. Dabei werden, je nach den Parametern des Environmet Objektes, die einzelnen Eigenschaftswerte des übergebenen PicturObjects zusammengerechnet.

`real getRandom(long double max)`

Eingabe: die Obergrenze max für die Zufallszahl die generiert werden soll
Rückgabe: eine Zufallszahl zwischen 0 und der angegebenen Obergrenze max

Liefert eine Zufallszahl zwischen 0 und der angegebenen Obergrenze max.

`Parameter getParameter()`

Rückgabe: aktuelle Parameter des Algorithmus

Liefert eine Kopie des Parameterobjektes (ParameterV).

`bool setParameter(Parameter parm)`

Eingabe: ein Parameterobjekt, das die Werte enthält auf die das eigene Parameterobjekt gesetzt werden soll

Setzt das eigene Parameterobjekt (ParameterV) auf die Werte des übergebenen ParameterObjektes parm.

`unsigned int getNumberOfIndividuals()`

Rückgabe: die Anzahl der Individuen in der Individuenliste

Liefert die Anzahl der Individuen in der Individuenliste (IndividualList).

15.3.8 Realisierung der Klasse Individual für einzelnen Individuen

Die Klasse Individual ist eine Hilfsklasse der Environment Klasse. Sie dient dazu, die Daten die zu einem Individuum gehören zusammenzufassen und zu verwalten. Diese Daten sind das PicturObject (fib) Objekt selbst, seine Goodness (Fitness), sein Immortal Status (ob es Unsterblich ist) und sein Identifier (eine Zahl mit der das Individuum im Algorithmus identifiziert werden kann).

Die Klasse Individual enthält dafür jeweils eine Methode zum Setzen und Löschen dieser Daten. Weiterhin eine Methode/Operator um zwei Individual Objekte zu ordnen, damit Individual Listen geordnet werden können.

Alle Methoden sind wegen ihrer Einfachheit und für eine höhere Ausführungsgeschwindigkeit als inline deklariert.

Konstruktoren

Die Klasse Individual verfügt über einen Standardconstructor, einen Copyconstructor und einen Parameterconstructor.

Der Standardconstructor tut nichts.

Der Parameterconstructor erhält als Parameter ein Zeiger auf ein PicturObject (fib) Objekt, einen real Wert als Goodness und ein und einen unsigned integer Identifier, mit denen er die eigenen Variablen initialisiert. Das PicturObject, auf das der Zeiger zeigt, wird nicht kopiert.

15.3.9 Realisierung der Klasse Parameter für die Parameter des genetischen Algorithmus

Dient zur Persistenzhaltung und Verwaltung aller Parameter die vom genetischen Algorithmus verwendet werden. Dabei soll eine hohe Flexibilität gewährleistet werden.

Die Parameter werden in fünf verschiedene Arten eingeteilt: Initialisierungsparameter, Parameter für die Abbruchbedingung, Parameter für die Operationen, Namen als Parameter (z.B. Dateinamen) und Parameter mit denen einzelne Dokumentationen beeinflusst werden können (Aus- und Einschalten ob die aktuelle Operation ausgegeben werden soll usw.). Für jede dieser Arten wird ein Array angelegt (Init, TerminalCondition, Operations, Names, Document). Es gibt für jedes Array eine Abfragemethode (`getXXX()`) und eine Methode zum Setzen (`setXXX()`) eines Elementes des Arrays. Diese Einteilung ist nur eine Richtlinie, unter anderem für eine besseren Übersichtlichkeit. Parameter können, z.B. wegen ihres Datentyps, durchaus in eine Parameterart gesteckt werden, zu der sie eigentlich nicht gehören. Auch können jederzeit neue Parameterarten dazukommen, wenn sie benötigt werden. Es geht nur darum, alle Parameter die der Algorithmus benötigt, möglichst sinnvoll in der Parameterklasse zu verwalten.

Konstruktoren

Die Klasse Parameter verfügt über einen Standardconstructor, einen Copyconstructor und einen Parameterconstructor.

Der Standardconstructor initialisiert alle Parameter mit einen Anfangswert (0 oder leerer string).

Der Parameterconstructor erhält als Parameter einen Pfad zu einer Parameterdatei, aus der er die Werte lädt.

Bedeutung von Parametern in den Arrays

In den Tabellen für Init Parameter 15.1, Abbruchbedingung/TerminalCondition Parameter 15.2, Operationsparameter 15.3 und Documentationsparameter 15.5 sind die vorläufige Bedeutung der Werte an den angegebenen Positionen in den Arrays aufgelistet.

Pos.	Bedeutung
0	maximale Individuenzahl der Individuen im genetischen Algorithmus (in der Individuenliste des Enviroment Objektes)
1-9	noch frei, weitere Spezifizierung der Individuenzahl
10	wie viele Individuen als zufällige erzeugte Point Objekte initialisiert werden sollen
11	wie viele Individuen als schon ”korrektes Bild“ initialisiert werden (für jeden Punkt im Bild wird das entsprechende Point Objekt mittels conc am PicturObject angehängt)

Tabelle 15.1: Init Parameter zur Initialisierung

Pos.	Bedeutung
0	maximale Anzahl der Durchläufe
1	Goodness die erreicht werden soll
11-50	Gewichte zur Erzeugung der Goodness/Fitness
11	Gewicht für den Abstand zum Originalbild
12	Gewicht für die Größe des fib-Objektes (PicturObject)
13	Gewicht für die zur Abarbeitung des fib-Objektes (PicturObject) benötigte Zeit

Tabelle 15.2: TerminalCondition Parameter für die Abbruchbedingung

Pos.	Bedeutung
0	Anzahl (n) der "unsterblichen" Individuen, d.h. n der besten Individuen können nicht gelöscht werden
1	Nummer des Auswahlmechanismus
10-100	mit welcher Wahrscheinlichkeit die Operation (Nummer Position - 10) angewendet werden soll

Tabelle 15.3: Operations für die Ausführung der Operationen

Pos.	Bedeutung
0	Pfadname der Datei für das Originalbild
1	Pfadname der Datei in die das Bild des vorläufig besten Individuums gespeichert werden soll
2	Pfadname der Datei in die das vorläufig beste Individuums gespeichert werden soll
3	Pfadname der Datei in der PicturObject Objekte stehen, die bei der Initialisierung des Environment Objektes benutzt werden sollen

Tabelle 15.4: Names für Namen

Pos.	Bedeutung
0	aktuelles einzufügendes Objekt
1	aktuelle Operation
2	aktuelles zu löschendes Objekt
3	ermittelte Distanz das PicturObjects, für das die Goodness ermittelt werden soll, vom Originalbild
4	Nummer/Identifier des aktuellen Individuums, das in die Individuenliste eingefügt werden soll
5	PicturObject für das aktuell die Goodness berechnet wird
6	ob ein neues, bestes Individuum an die Datei mit den besten Individuen angehängt werden soll, wenn false, wird das alte beste Individuum überschrieben

Tabelle 15.5: Document für die Dokumentation (wenn true, wird die entsprechende Dokumentation ausgegeben)

15.3.10 Realisierung der Klasse PicturMatrix für die Bildmatrix

Die Klasse PicturMatrix dient zum Aufbewahren und Verarbeiten von Bilddaten. Obwohl es sicherlich schon viele realisierte Matrixklassen und Matrixtemplates gibt, wird trotzdem eine eigene Klasse geschrieben. Der Grund ist unter anderem, eine höhere Flexibilität, bei Erweiterungen und Veränderungen und ein möglichst geringer Umfang der Klasse, zwecks Schonung des Speichers. Es wäre gut, wenn die Klasse den CPU Cache nur teilweise belegt, da Bilder sehr oft verglichen werden müssen.

Viele Methoden sind wegen der Geschwindigkeit als inline deklariert.

Darstellbar sind vier dimensionale Matrizen (also maximal holographische Filme), die Einsparung, wenn nur drei oder weniger dimensionale Matrizen als Obergrenze genommen werden, ist sehr gering.

Jedes PicturMatrix Objekt verfügt über eine Hintergrundfarbe, dessen Komponentenwerte standardmäßig auf 0 gesetzt sind. Welche Bildformate aktuell einlesbar sind, ist der dem Programmsystem beiliegende”readme.txt“ Textdatei zu entnehmen.

Konstruktoren

Die Klasse PicturMatrix verfügt über einen Standardconstructor, einen Copyconstructor und zwei Parameterconstructor.

Der Standardconstruktor setzt alle Werte auf 0 (auch Matrix Größe 0, mit einem Wert 0).

Der eine Parameterconstructor ist ein load Constructor, welcher die Daten für die Matrix aus einer Datei lädt, dessen Namen ihm übergeben werden.

Dem andere Parameterconstructor werden die Werte für die Größe (`unsigned int* sz`), die Nummer der Color Componenten (`unsigned int ncc`) und die (Farb-) Tiefe dieser (`unsigned int* dep`) übergeben. Der Construktor setzt die Variablen der Klasse auf die entsprechenden Werte und erzeugt eine entsprechend große leere Matrix.

Der Copyconstructor wurde angepasst. Ihm kann ein zusätzlicher boolscher Parameter (standardmäßig true) übergeben werden. Wenn dieser true ist, wird die übergebene PicturMatrix ganz kopiert inklusive Bildmatrix, sonst wird anstatt der Bildmatrix eine leere Bildmatrix, mit gleicher Größe wie die des übergebenen Objektes, erstellt. Diese Option dient dazu, eine PicturMatrix mit leeren Bild zu erstellen, dessen anderen Werte aber die gleichen sind, wie die im übergeben PicturMatrix Objekt. So kann von einem Originalbild einfach eine leere Kopie mit gleichen Attributen erstellt werden und diese dann mit einem PicturObject gefüllt (neu bezeichnet) werden.

Methoden

`unsigned int[4] getSize()`

Rückgabe: ein unsigned int array mit 4 Komponenten, für die 4 Dimensionen der Matrix, in dessen Komponenten die Ausdehnung der jeweiligen Dimension steht

Die Methode gibt die Größe der Matrix zurück.

`unsigned int getNumberOfColorComponents()`

Rückgabe: die Anzahl der Elemente die ein Colorvector der Matrix hat, z.B. 3 für das RGB Farbschema

Die Methode gibt die Anzahl der Elemente eines Colorvectors der Matrix zurück. Dient dazu ermitteln zu können, wie viel Elemente Colorvectoren in einem entsprechendes fib-Objekt haben müssen, um alle Bilder erzeugen zu können, die mit der PicturMatrix möglich sein sollen.

`unsigned int getNumberOfPositionComponents()`

Rückgabe: die Anzahl der Elemente, die ein Positionsvector haben sollte, um alle Positionen in der Matrix repräsentieren zu können, z.B. 2 wenn nur die zwei letzten Ausdehnungen der Matrix 0 sind

Die Methode gibt die Anzahl der Elemente, die ein Positionsvector haben sollte, um alle Positionen in der Matrix repräsentieren zu können an. Es muss für jede Dimension, deren Ausdehnung größer 0 ist, eine Positionskomponente geben. Die Methode dient dazu ermitteln zu können, wie viel Elemente Positionsvectoren in einem entsprechendem fib-Objekt haben müssen, um alle Bilder erzeugen zu können, die mit der PicturMatrix möglich sein sollen.

`unsigned int getDepthOfComponent(unsigned int n)`

Rückgabe: die Anzahl der Werte die die n'te Color Komponente annehmen kann

Die Methode gibt die Anzahl der Werte zurück, welche die n'te Color Komponente annehmen kann. Dies ist Teil der Farbinformationen über das Bild, wenn es sich z.B. um ein schwarz/weiß Bild handelt, gibt es nur eine Color Komponente die die Werte 0 und 1 annehmen kann, also zwei Werte, bei einem RGB Bild sind es 3 Komponenten mit jeweils 256 möglichen verschiedenen Werten.

Diese Information dient unter andern dazu, den Abstand zu anderen PicturObject Bildern zu ermitteln.

`Color* getBackgroundColor()`

Rückgabe: die Hintergrundfarbe des Bildes

Die Methode gibt die Hintergrundfarbe des PicturObjects zurück. Die Hintergrundfarbe ist Standardmäßig ein Nullvektor.

`Color* getColorFrom(unsigned int[4] pos)`

Eingabe: unsigned int array für die Position pos des Pixels desen Farbe zurückgegeben werden soll
Rückgabe: ein Zeiger auf das Color Objekt, das an der Position pos in der Matrix steht

Die Methode gibt einen Zeiger auf das Color Objekt zurück, das an der Position pos in der Matrix der PicturMatrix steht.

`bool setColorOnPosition(unsigned int* pos,Color col)`

Eingabe: eine unsigned int array für die Position pos und ein Color Objekt, von dem eine Wertekopie (Variablen im Color Objekt werden durch die Werte ersetzt die sie beinhalten) an die Position pos in der Matrix gesetzt werden soll
Rückgabe: true wenn das Setzen erfolgreich war, sonst false

Die Methode setzt eine Wertekopie, von einem übergebenen Color Objekt, an die Position pos in der Matrix, wenn es diese Position gibt.

`void overlap(PicturMatrix matrix)`

Eingabe : eine PicturMatrix

Die Methode fügt dort in der eigenen Matrix Punkte aus der Matrix des übergebenen PicturMatrix Objektes ein, wo die eigene Matrix noch keine eigenen Color Objekte besitzt.

Es werden sozusagen die Bilder der beiden PicturObjecte übereinandergelegt, wobei das eigene Bild das Bild des übergebenen Objektes überlappt.

`bool load(string file)`

Eingabe: ein Pfadname zu einer Datei, aus der das Bild in das eigene Objekt geladen werden soll

Die Methode lädt das Bild, aus der Datei file, in das eigene Objekt, also dessen Werte und Bildmatrix.

Für die möglichen Bildformate die geladen werden können, werden in dieser Methode nach und nach Erweiterungen eingefügt. Welche Bildformate aktuell einlesbar sind, ist der dem Programmsystem beiliegende ”readme.txt“ Textdatei zu entnehmen.

`bool store(string file)`

Eingabe: ein Pfadname zu einer Datei in der das Bild des eigenen Objektes gespeichert werden soll

Die Methode speichert das Bild, welches das eigene Objekt realisiert, in der Datei file.

Für die möglichen Bildformate die gespeichert werden können, werden in dieser Methode nach und nach Erweiterungen eingefügt.

`unsigned int distanceTo(PicturMatrix matrix)`

Eingabe: ein PicturMatrik Objekt zu dem die Distanz berechnet werden soll
Ausgabe: ein Wert für die Distanze der beiden PicturMatrix Objekte zueinander

Die Methode ermittelt wie ähnlich sich dieses und die gegebene Matrix sind.

Die Distanz zweier PicturMatrix Objekte, ist die Summe der Distanzen, die sich ergeben, wenn die Distanzen aller ihrer Color Objekte an den entsprechenden/gleichen Positionen ermittelt werden. Die Hintergrundfarbe ist dabei auch ein Color Objekt, das überall dort in der Matrix steht, wo kein anderes Color Objekt steht. Die Distanz eines Color Objektes zu einem anderen, ergibt sich aus den Distanzen entsprechender Komponenten (z.B. der ersten Komponenten). Die Distanz zweier Color Komponenten ergibt sich aus deren Abstand zueinander, wobei dieser niemals größer als die Farbtiefe (depth, Anzahl der möglichen Werte der Komponente) dieser Komponente sein kann. Bei schwarz/weiß Bildern ist die depth gleich 2, daraus folgt das die größte Distanz gleich 2 ist.

`unsigned int distanceToOfArea(PicturMatrix matrix, unsigned int[4] from, unsigned int[4] to)`

Eingabe: ein PicturMatrik Objekt zu dem die Bereichsdistanz berechnet werden soll und der Bereich für den die Distanze ermittelt werden soll, angegeben durch seine obere, linke Grenze (from) und untere, rechte Grenze (to), der Bereich ist immer rechteckig
Ausgabe: ein Wert für die Distanze der beiden PicturMatrix Objekte zueinander, in dem angegebenen Bereich

Diese Methode funktioniert wie die Methode `distanceTo()`, nur das nicht die Color Objekte an allen Positionen einbezogen werden, sondern nur solche, die an Positionen innerhalb des angegebenen Bereichs liegen.

Diese Methode soll dazu dienen, dass auch Teilbilder verglichen werden können.

`void resize(unsigned int[4] nsize)`

Eingabe: das unsigned int array nsize, das die neue Größe angibt

Verändert die Größe der Matrix auf die neuen Werte. Dabei werden alle Color Objekte, die sich innerhalb der Grenze der neuen Matrix befinden, von der alten Matrix übernommen.

`void clear()`

Löscht alle vorhandenen Color Objekte die sich in der Matrix befinden.
Anmerkung: Die Hintergrundfarbe befindet sich nicht in der Matrix.

Kapitel 16

Anmerkung zur Realisierung

Da es sich bei der Realisierung um ein System zum Experimentieren handelt, soll es unter anderem dafür ausgelegt sein, ständige Veränderungen durchzumachen. Bei der Implementierung sollte darauf geachtet werden, dass nicht nur die Arbeitsweisen der Methoden und Klassen verständlich sind, sondern dass auch Möglichkeiten vorhanden sind diese zu erweitern. Stellen, die von vornherein für Erweiterungen vorgesehen sind, können gekennzeichnet sein.

Im Verzeichnis der realisierten Klassen sollte eine Datei "readme.txt" vorhanden sein, in der näher beschrieben wird, wie mit dem Programmsystem umgegangen werden sollte (Algorithmus starten, setzen von Parametern etc.).

Die Zufallszahlen die im Algorithmus benötigt werden, werden durch einen Zufallszahlengenerator (©1997, 2003 by Agner Fog; Uniform random number generators) generiert, den ich im Internet gefunden habe. Der heruntergeladene Code wurde so verändert, dass nur einer der Zufallszahlengeneratoren, die ursprünglich darin zusammengefasst waren, verwendet und eingebunden wird. Der Code des Zufallszahlengenerators wurde nicht verändert. Die Anforderungen, die die generierte Zufallszahlen erfüllen müssen, sind sehr gering. Die einzige wichtige Anforderung ist, das möglichst alle Zahlen aus dem entsprechenden Bereich generiert werden können. Selbst wenn die Periodizität der generierten Zufallszahlen sehr gering ist, ist zu erwarten, dass der Algorithmus trotzdem unperiodisch ist, da die Zufallszahlen ständig an anderen Stellen, mit anderen Bedeutungen und mit verschiednen fib-Objekten eingesetzt werden. Das heißt, ein sehr schlechter Zufallszahlengenerator ist kein wirklicher Nachteil für den Algorithmus. Laut der zum Code gehörenden Dokumentation ist der verwendete Zufallszahlengenerator wesentlich besser als der Zufallszahlengenerator, der bei den Standardbibliotheken von C++ dabei ist.

Kapitel 17

Testen des Programmsystems

Das Programmsystem kann nur mit sehr viel Aufwand vollständig getestet werden. Das betrifft vor allem den Test der Klassen für die Bildbeschreibungssprache und dadurch auch die von ihr abhängigen Klassen.

Dies liegt weniger daran das die Klassen der Bildbeschreibungssprache relativ komplex sind, als vielmehr daran dass die von ihnen erzeugten Objekte ineinander verschachtelt werden können und damit ein vollständiger Test alle Möglichkeiten der Verschachtelung berücksichtigen müsste.

Zum Glück ist ein vollständiger Test auch gar nicht nötig, da die ganze Anwendung völlig unkritisch ist. Nicht nur, dass das Programmsystem in einem Bereich (Forschung) angewendet wird, in dem Ausfälle und Abstürze nicht weiter problematisch sind, da immer mit ihnen gerechnet werden muss (da es experimentell ist). Außerdem ist es ein ”anytime“, ”anywhere“ (immer und überall) Algorithmus. Das heißt, wenn es zu Fehlern oder Abstürzen kommt, hat man im allgemeinen noch das letzte beste Ergebnis, das das Programmsystem geliefert hat und es sind nicht alle Daten verloren. Auch dürften Abstürze eines Teils des Systems, bei einer verteilten Anwendung, sich nicht sehr negativ auf andere Teile auswirken und man hätte an jedem Ort immer noch das letzte beste Ergebnis, welches dort berechnet wurde.

Deshalb wurden bei den Klassen für die Bildbeschreibungssprache und den von ihnen abhängigen Klassen nur die Grundfunktionalitäten getestet.

Zu erwähnen wäre noch, dass wenn das Programmsystem läuft, die Klassen in gewisser Weise einen ständigen Randomtest durchlaufen. Da möglichst schnell hintereinander zufällige Kombinationen erzeugt werden, so dass, wenn bei der Konfiguration (Parametern) des laufenden Programmsystems ein Fehler auftreten kann, er es wahrscheinlich auch früher oder später wird. Bei 3.000.000 Operatorenanwendungen fallen dadurch wahrscheinlich die meisten Fehler auf, auch Speicherlöcher, irgendwann ist auch der größte Speicher voll. Da das Programmsystem mittlerweile schon sehr oft und lange gelaufen ist, dürften die meisten Fehler inzwischen auch schon aufgetreten sein.

Virtuelle Klassen (GraphicObject, PicturObject, ListObject) werden natürlich implizit geprüft, über die nicht virtuellen Klassen die von ihnen erben.

Für jede Klasse gibt es eine Methode, die zum Testen dieser angelegt wurde. Jede dieser Methoden steht in einer separaten Datei. Der Name der Methode und der Datei wird am Anfang jeder Testbeschreibung genannt. Die Datei zum Testen einer Klasse X hat den Namen test_X.cpp und die Methode zum testen `testX()`.

Bei einigen Tests werden die Testdaten in eine Protokolldatei geschrieben und bei manchen von ihnen sogar gezählt wie viel Einzeltests richtig waren. Andere Tests müssen dagegen manuell überprüft werden. Elementartests, bei denen automatisch die richtigen und fehlerhaften Testfälle zusammengezählt werden, sind meist Tests bei denen Methoden konkrete Zahlenwerte ausgeben sollen, die vorher bekannt sind.

17.1 Parameter

Datei: test_Parameter.cpp
Methode: `testParameter()`

Die Klasse Parameter ist sehr einfach zu testen, da die Methoden meist nur aus wenigen Zeilen bestehen.

Es wird einfach getestet, ob die übergebenen Werte richtig zurückgegeben werden und ob sie speicherbar und wiederherstellbar sind.

17.2 Die PicturMatrix Klasse

Datei: test_Picturmatrix.cpp
Methode: `testPicturmatrix()`

Bei der Klasse PicturMatrix wird geprüft, ob Bildmatrizen aus fib-Objekten richtig erzeugt werden und ob Bilddaten, von Bilddateien bestimmter Formate, richtig eingelesen und gespeichert werden.

17.3 Klassen für die fib-Objekte

17.3.1 Die Vector Klassen

Datei: test_Vector.cpp
Methode: `testVector()`

Mit verschiedenen Vector Objekten wird geprüft, ob ihre Methoden die richtigen Werte zurückgeben und ob verändernde Methoden das Vector Objekt richtig verändern. Als Testvektorobjekt wird dabei ein Color Objekt verwendet. Da fast alle Methoden schon in der Vector Klasse realisiert werden, sind die Methoden in anderen Vektorklassen meist die gleichen. Dieser Test ist fast vollständig.

17.3.2 Die Klasse Point

Datei: test_Point.cpp
Methode: `testPoint()`

Die Klasse Point kann noch gut geprüft werden, da sie keine unendlichen Objekte enthält.

Dabei wird mit verschiedenen Point-Objekten geprüft, ob ihre Methoden die richtigen Werte zurückgeben und ob verändernde Methoden das Point-Objekt richtig verändern.

17.3.3 Die Klasse Conc

Datei: test_Conc.cpp
Methode: `testConc()`

Beim conc Objekt werden nur sehr einfache fib-Objekte, die ein conc Objekt enthalten, geprüft, da das conc Objekt keine tiefere Funktionalität hat. Die einzige kompliziertere Methode ist die `moveElement()` Methode, die aber eine eigene Testmethode hat.

17.3.4 Die ListObjekt Klassen

Datei: test_Function.cpp
Methode: `testFunction()`

Die ListObject Klassen, Function und Area, werden größtenteils gemeinsam geprüft, da die Klasse ListObjekt die meisten ihrer Methoden realisiert. Das heißt konkret, dass es nur eine Methode (`testFunction()`)/Datei (test_Function.cpp) zum testen der Function Klasse gibt, in der stichprobenhaft die Methoden von ein paar fib-Objekten mit Function Elementen getestet werden.

17.4 Die Klasse Environment

Datei: test_Environment.cpp
Methode: `testEnvironment()`

Einem Objekt der Klasse Environment werden Parameter übergeben. Diese Parameter sorgen dafür, dass alle zu testenden Methoden aufgerufen werden. Dann wird dieses Environment Objekt gestartet und führt sozusagen einen Randohmtest aus. Ob die einzelnen Methoden richtig arbeiten, kann getestet werden, indem die erzeugten (einzufügenden) Objekte manuell überprüft werden. Diese können per Parametereinstellung ausgegeben werden.

Wichtig bei der Prüfung von Klassen ist, das sie das tun was sie sollen. Das ist in diesem Fall, dass die Klasse Environment den genetischen Algorithmus realisiert

und weniger, dass die Methoden genau das tun was sie sollten. Daher wurde auch hauptsächlich geprüft, ob der Algorithmus ”gute“ Ergebnisse liefert und nicht die Methoden in allen Einzelheiten.

Ich vermute z.B. das die Hilfsmethode `selectPartObject(obj)` nicht genau das tut was sie soll (prüft eventuell zu viele Teilobjekte). Da der Fehler aber schwer zu finden ist und das Ergebnis dieser Methode anscheinend in Ordnung ist, belasse ich es dabei.

17.5 Die Klasse Individual

Die Klasse Individual wird implizit von der Klasse Environment mitgeprüft. Eine extra Testmethode wurde wegen der Einfachheit dieser Klasse nicht geschrieben.

17.6 Testen der `moveElement()` Methode

Datei: test_move.cpp
Methode: `testmove()`

Wegen der hohen Komplexität der `moveElement()` Methode und der Tatsache, dass sie nur an größeren fib-Objekten getestet werden kann, gibt es für sie eine extra Methode, mit der sie getestet wird.

Getestet wird, ob das entstehende fib-Objekt nach Aufruf der `moveElement()` Methode das richtige fib-Objekt ist, d.h. das entsprechende Element wurde verschoben (evtl. um 0), auch wenn Parameter, die eingegeben wurden, ungültig waren, z.B. `moveElement()` mit einer Verschiebedistanz über das fib-Objekt hinaus. Es dürfen beim `moveElement()` Aufruf keine Schleifen im fib-Objekt entstehen und es darf auch nicht einfach ein Element verschwinden, sonst gibt es Speicherlöcher.

17.7 Bewertung des Tests des Programmsystems

Der Test ist, wie gesagt, bei weitem nicht vollständig. Da das Programmsystem aber bisher fehlerfrei läuft und auch nicht mehr gefordert wird, halte ich die bisherigen Tests für das jetzige Programmsystem für ausreichend. Bei Erweiterungen sind, für diese Erweiterungen, weitere Tests angebracht. Problematische Fehler, die im Test aufgefallen sind, sind korrigiert worden. Andere (zurzeit ein möglicher Fehler bekannt) wurden erst einmal ignoriert, da sie sich beim Laufen des Programmsystems anscheinend nicht weiter nachteilig auswirken.

Kapitel 18

Bewertung der Realisierung

Das Programmsystem läuft stabil und realisiert die Grundfunktionalitäten. In Anbetracht des Umfangs dieser und der Zeit die zur Verfügung stand, halte ich den jetzigen Stand schon für sehr gut.

Durch die leichte Erweiterbarkeit können jederzeit mehr Funktionalitäten hinzugefügt werden. Was auch ein Schwerpunkt der Realisierung war, in Anbetracht dessen dass es sich um ein System handelt, das vorrangig zum Experimentieren gedacht ist.

Mit dem Programmsystem können interessante Ergebnisse gewonnen werden. (Siehe Teil V) Womit die Aufgabenstellung im wichtigsten Punkt erfüllt wurde.

Kapitel 19

Aussicht

Bei dem Programsystem kann noch eine ganze Menge verbessert werden.

Einige mögliche Punkte sind:

- Mit ein bisschen Optimierung wird der Vergleich eines PicturMatrix und eines fib-Objektes sicherlich um einiges schneller. Entfallen könnte z.B. das Anlegen einer neuen PicturMatrix und das Erstellen von Punkten eines fib-Objektes, die eigentlich verdeckt sind. Da der Vergleich ein Großteil der Laufzeit des Algorithmus ausmacht, ist eine solche Optimierung sinnvoll.

- Bei den Operatoren für den genetischen Algorithmus, kann noch sehr viel erweitert und verbessert werden. Realisiert wurden bisher nur sehr einfache Operatoren, die hauptsächlich mit Zufall arbeiten. Operatoren die wesentlich intelligenter und damit komplexer sind, können weitere Vorzüge bringen. Operatoren die beispielsweise das Originalbild nach bestimmten Strukturen absuchen und diese dann in ein fib-Individuum einfügen.

- Eine Möglichkeit die fib-Objekte in einem speichersparenden Format abzuspeichern, würde den möglichen Komprimierungsfaktor erhöhen, da zur Zeit fib-Objekte nur als sehr speicheraufwendige, dafür aber lesbare, Zeichenketten abgespeichert werden können. Damit verbunden sollte auch ein eigenes Bildformat entwickelt werden, inklusive Header mit allgemeinen Bildinformationen wie Größe oder Informationen über das verwendete Farbschema.

- Die Erweiterung der PicturMatrix Klasse, so dass mehr Bildformate geladen werden können, ermöglicht eine breitere Einsetzbarkeit.

Auch sind eine ganze Menge Zusatzwerkzeuge für eine bessere Verwendung der fib-Bildbeschreibungssprache und dem Algorithmus möglich:

- Eine bessere Benutzerschnittstelle könnte mit einer graphische Oberfläche realisiert werden, so dass die Parameter komfortabel gesetzt werden können und auch der aktuelle Stand (Nummer des Durchlaufs, bestes Individuum, Goodness Verlauf der besten Individuen) des Algorithmus angezeigt wird.

- Eine Möglichkeit von Zeit zu Zeit die Parameter automatisch zu ändern, würde den Algorithmus bereichern.
- Da genetische Algorithmen gut für die verteilte Verarbeitung geeignet sind, wäre ein Austauschprogramm für Individuen zwischen einzelnen Algorithmenprozessen nützlich (auch über ein Netzwerk hinweg).
- Ein Zeichenprogramm für fib-Bilder, würde die Verbreitung des fib-Formats sicherlich fördern. Wünschenswert wäre, das es über ähnliche Möglichkeiten wie andere Vektorzeichenprogramme verfügt.

Teil V

Auswertung

Kapitel 20

Aufwand zur Bildcodierung

Das Programmsystem ist leider zurzeit noch nicht für eine sinnvolle Bildverarbeitung geeignet, da es schon für relative einfache, kleine Bilder zuviel Zeit zum Umsetzen dieser Bilder benötigt wird.

Es kann dadurch aber noch keine Aussage über die allgemeine Nutzlosigkeit des Ansatzes zur Bildverarbeitung gemacht werden, da spezielle, angepasste Operatoren oder andere Optimierungen, durchaus dazu führen können das der Verarbeitungsaufwand, auch für größere Bilder oder sogar Filme, auf ein akzeptables Maß schrumpft.

Leider können durch die Zeit die der Algorithmus zur Zeit benötigt nur sehr wenige Parameterkombinationen ausprobiert werden. Das erschwert das Austesten der Idee und statistische Auswertungen sind in der zur Verfügung stehenden Zeit nicht mehr zu realisieren.

20.1 Aufwand der Bildcodierung hängt von der Komplexität des Bildes ab

Interessanterweise hängt der Aufwand der Bildcodierung weniger von der Größe des Bildes ab, sondern vielmehr von der Komplexität den Strukturen auf dem Bild.

Es wurden mit drei verschiedenen Bildern Durchläufe gemacht:

- mit einem kleinem Originalbild (sw_block.btx) mit 6 mal 6 ($= 36$) Pixel und einem "weißen" Rechteck von der Position (2;2) bis (5;5)
- mit einem großem Originalbild (bigb.btx) mit einem 32 mal 32 ($= 2^{10} = 1024$) Pixel und einem "weißen" Rechteck von der Position (3;4) bis (27;27)
- mit einem sehr großem Originalbild (sw256_block1.bmp) mit einem 256 mal 256 ($= 2^{16} = 65.536$) Pixel und einem "weißen" Rechteck von der Position (10;10) bis (238;238)

Für alle Bilder wurden, bis auf den Namen der Bilddatei, die gleichen Parameter verwendet.

Für das kleinere Bild kam der Algorithmus bei dem dritten Durchlauf und rund 76.891 Iterationen auf das optimale Ergebnis:
"for(x1,[(4,1)],for(x2,[(4,1)],p((x2,x1),(1))));goodness : 5917.16 distance to original : 0 time need : 69 greatness : 10 idv : 76891"
Zu findet im Verzeichnis: \Tests\little\
Die Bilddaten sind in Listing 20.1 zu sehen. (im *.btx Format, die ersten beiden Zahlen geben die Größe des Bildes an)

Listing 20.1: Kleines btx Testbild

```
6;6;
000000
011110
011110
011110
011110
000000
```

Für das große Bild kam der Algorithmus bei dem zweiten Durchlauf und 11.088 Iterationen auf das optimale Ergebnis:
"for(x1,[(3,26)],for(x2,[(2,26)],p((x2,x1),(1))));goodness : 396.04 distance to original : 0 time need : 2425 greatness : 10 idv : 11088"
Zu findet im Verzeichnis: \Tests\bigb\
Die Bilddaten sind in Listing 20.2 zu sehen.

Listing 20.2: Großes btx Testbild

```
32;32;
00000000000000000000000000000000
00000000000000000000000000000000
00011111111111111111111111100000
00011111111111111111111111100000
00011111111111111111111111100000
00011111111111111111111111100000
00011111111111111111111111100000
00011111111111111111111111100000
00011111111111111111111111100000
00011111111111111111111111100000
00011111111111111111111111100000
00011111111111111111111111100000
00011111111111111111111111100000
00011111111111111111111111100000
00011111111111111111111111100000
00011111111111111111111111100000
00011111111111111111111111100000
00011111111111111111111111100000
00011111111111111111111111100000
00011111111111111111111111100000
00011111111111111111111111100000
00011111111111111111111111100000
00011111111111111111111111100000
00011111111111111111111111100000
00011111111111111111111111100000
00011111111111111111111111100000
00011111111111111111111111100000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

Für das größte Bild kam der Algorithmus bei einem von zwei Durchläufen und 89.853 Iterationen auf das optimale Ergebnis:
"for(x1,[(238,10)],for(x2,[(238,10)],p((x2,x1),(1))));goodness : 4.75977 distance to original : 0 time need : 209994 greatness : 10 idv : 89853"
Zu findet im Verzeichnis: \Tests\sbigb1\
Die Bilddaten sind sehr Umfangreich und werden deshalb hier nicht angegeben. Das zugehörige Bild ist im Bild 20.1 zu sehen.

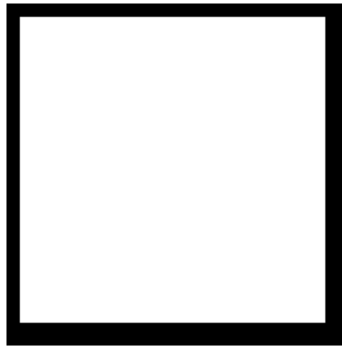

Abbildung 20.1: Allgemeiner Algorithmus

Bild	Iterationen/11.088	Bildgröße in BMP/Bildgröße des kleinsten Bildes
klein	6,9	1
groß	1	28,4
größte	8,1	1802,4

Tabelle 20.1: Operations für die Ausführung der Operationen

In Tabelle 20.1 sind die Daten für die Komprimierungsfaktoren und die Anzahl der Iterationen im Verhältnis zu den jeweils kleinsten Zahlen dargestellt.

Die drei Werte für die Anzahl der nötigen Iterationen liegen ungefähr in der gleichen Größenordnung, während die Größe dieser Bilder exponential anstieg. Andererseits benötigt die Kodierung von 8 mal 8 = 256 Pixel Bildern mit komplexeren Strukturen wesentlich mehr Iterationen, als die Kodierung des größten der oben aufgeführten Bilder.

Es ist deshalb anzunehmen das der Komprimierungsaufwand hauptsächlich durch die Komplexität der auf den Bildern dargestellten Objekten steigt und weiniger mit der Größe dieser.

Die Anzahl der Iterationen ist natürlich nicht linear zum wirklichen Zeitaufwand. Es kann davon ausgegangen werden, das der durchschnittliche Realzeitaufwand pro Iteration linear mit der Anzahl der Pixel steig. Da der Zeitaufwand in der jet-

zigen Realisierung hauptsächlich von der Vergleichsoperation verursacht wird und deren benötigte Zeit umgekehrt proportional zur der Anzahl der Pixel steigt.

Kapitel 21

Komprimierung

Es ist möglich ein Bild in fib gegenüber einem Bitmapbild sehr stark zu komprimieren.

Als ein Beispiel dafür sei das oben schon benutzte fib-Objekt mit sehr großem Originalbild (sw256_block1.bmp) mit einem 256 mal 256($= 2^{16} = 65.536$) Pixel und einem "weißen" Rechteck von der Position (10;10) bis (238;238). Die gefundene fib-Darstellung ist: "for(x1,[(238,10)],for(x2,[(238,10)],p((x2,x1),(1))))" Diese kann mit 53 Bytes kodiert werden (der oben angegebene String hat 53 Zeichen). Eine effizientere Komprimierung ist aber auf jeden Fall möglich.

Die Bitmapdarstellung in schwarz/weiß benötigt mindestens ein Bit pro Pixel, also werden 65.536 Bits oder (65.536/8) Bytes = 8.192 Bytes für das Bild in Bitmapdarstellung benötigt (das OS2 Bitmapformat (sw256_block1.bmp) benötigt 8.224 Bytes).

Selbst bei der schlechten fib-Darstellung (als Zeichenkette) ist das immer noch ein Komprimierungsfaktor von rund 154,6, das heißt die Bitmapdarstellung ist rund 154,6 mal so groß wie die oben angegebene fib-Darstellung.

An den anderen Bildern mit einem Rechteck kann auch gut gesehen werden, dass die Größe einer fib-Darstellung wenig von den Pixeln des Originalbildes abhängt, sondern von der Anzahl und Struktur der Objekte in ihm. Für alle drei Rechteckbilder in der fib-Darstellung als Zeichenkette, wurde ungefähr die gleiche Anzahl von Bytes benötigt (49 bis 53 Bytes), da alle drei Bilder nur ein Rechteck darstellen.

Bei komplizierteren Strukturen auf dem Bild steigt natürlich der Speicheraufwand. Leider steigt auch mit komplizierteren Strukturen der Kodierungsaufwand und die Wahrscheinlichkeit eine gute fib-Darstellung in akzeptabler Zeit zu finden sinkt. Dadurch wimmelt es in den bisher gefundenen fib-Darstellungen für komplexere Strukturen von überflüssigen fib-Strukturen. Damit werden diese fib-Darstellungen größer als die entsprechende Bitmapdarstellung.

Kapitel 22

Verständlichkeit

Wenn die Strukturen des Bildes einfach sind und die gefundene Lösung kurz ist, ist sie verständlich. Oder kurz: je kürzer die fib-Darstellung, um so verständlicher ist sie.

Die fib-Darstellungen für die drei oben angeführten Bilder mit einem Rechteck sind z.B. sehr einfach zu verstehen.

Beispiel: "for(x1,[(3,26)],for(x2,[(2,26)],p((x2,x1),(1))))"

Die Variablen x1 und x2 laufen beide unabhängig voneinander von 3 bis 26. Damit füllt der Punkt einen Bereich aus, der sowohl in x als auch in y Richtung von der Position 3 bis 26 reicht. Trotz ihrer Kürze kann schon diese Darstellung auf einige Arten verändert werden ohne das dargestellte Bild oder die Einfachheit der Darstellung zu verändern, z.B. können die "for" Elemente vertauscht werden oder die Grenzen in den Teilbereichsvektoren.

Bei komplizierteren fib-Objekten steigen diese Möglichkeiten stark an. Außerdem macht die Menge der fib-Elemente die Interpretation sehr viel schwieriger. Allerdings kann immer ein Ast von einem Punkt ausgehend zurückverfolgt werden und verstanden werden, wie sich dieser Punkt im erzeugten Bild auswirkt.

Ein etwas größeres fib-Objekt wird schwerer zu verstehen:

"for(x1,[(-1,-16)],fun(x2,[(x1,18,-12),(-2,x1,1)],for(x3,[(31,0)],p((x2,x3),(1)))))"
Dieses fib-Objekt stellt ein 32 mal 32 Pixel schwarz/weiß Bild dar, auf dem zwei aufeinanderfolgende Zeilen unterschiedliche Farben haben (horizontal schwarz-weiß gestrichelt), zusehen im Bild 22.1.

Das "for(x3,[(31,0)],..)" Element zieht die Striche von der linken zu rechten Seite durch.

Der "for(x1,[(-1,-16)],fun(x2,[(x1,18,-12),(-2,x1,1)],. . .)" Teil des fib-Objektes, bedeutet soviel wie: nehme die Zahlen von -1 bis -16 und multipliziere sie mit -2 (die Unterfunktion zum (x1,18,-12) Vektor berechnet sich zu rund 0), so dass damit jede zweite Zeile eine weiße Linie gezeichnet wird. Der Rest ist durch den schwarzen Hintergrund schwarz.

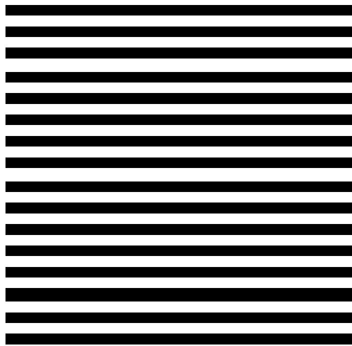

Abbildung 22.1: Bild mit horizontalen Strichen

Kapitel 23

Operatoren

Die derzeitige Realisierung lässt eine sehr große Freiheit für die Realisierung weiter Operatoren für den Algorithmus. Dies lässt auch einen großen Spielraum für die mögliche Geschwindigkeitsverbesserungen, mit der der Algorithmus zu einem guten Ergebnis kommt. In wirklich guten Operatoren kann eine ganze Menge Fachwissen stecken, das mir aber leider zur Zeit nicht zur Verfügung steht.

Möglich sind weitere Operatoren, die sich fib-Objekte vornehmen und sie auf verschiedene Weise verkürzen oder schneller in der Auswertung machen, ohne das dargestellte Bild zu beeinflussen.

Es können Operatoren geschaffen werden, die das Bild direkt nach bestimmten Strukturen durchsuchen und dann eine gefundene Struktur als ein entsprechendes optimiertes fib-Objekt in ein kopiertes Individuum einfügen.

Durch dieses "Fachwissen", das dem Algorithmus von Anfang an mitgegeben wird, hat er bessere Möglichkeiten die Bildstrukturen zu lernen.

Kapitel 24

Parameter

Es konnten leider nur sehr wenige Parameterkombinationen getestet werden, daher sind Aussagen darüber, wie bestimmte Parameter oder Parameterkombinationen sich auf den Algorithmus auswirken, nur schwer zu machen.

Im Allgemeinen kann gesagt werden, dass das Gewicht für den Abstand zum Originalbild (distance to original) immer wesentlich größer sein sollte als die Summe der Gewichte für die Abarbeitungszeit (greatness) und die benötigte Zeit zum dekodieren (timeNeeded) wenn ein ähnliches Bild gefunden werden soll. Um so größer dieser Abstand ist, um so schneller werden im allgemeinen Individuen generiert die dem Originalbild ähnlich sind, aber am so größer und langsamer sind diese Individuen. Welches Verhältnis gute Ergebnisse hervorbringt scheint unter anderem von der Farbtiefe des Originalbildes abzuhängen. Wenn der Parameter für das Gewicht für ”benötigte Zeit zum dekodieren des fib-Objekts“ (timeNeeded) im Verhältnis zu den anderen Gewichtsparametern größer wird, wird der genetische Algorithmus im Bezug auf die Zeit, welche die Iterationen durchschnittlich benötigen, schneller, da die fib-Objekte im Bezug auf ihre Auswertungszeit kleiner bleiben.

Der Parameter für die Wahrscheinlichkeit das die wertverändernde Operation angewendet wird, sollte wesentlich größer sein als die Parameter für Wahrscheinlichkeiten für andere Operationen, damit gute Individuen sich schneller entwickeln.

Kapitel 25

Bezugnahme auf die Aufgabenstellung

Das in der Aufgabenstellung dargestellte Ziel, ein System zu realisieren welches Bitmapbilder in platzsparende Vektorbilder umwandelt und eventuell in diesen Objekte mehrfach recycelt, ist nicht ganz erreicht worden. Für einzelne Beispiele (siehe ”Rechteckbilder“) funktioniert das Programmsystem schon recht gut, wenn aber die Bilder komplexere Strukturen enthalten, bringt das Programmsystem in annehmbarer Zeit keine akzeptablen Lösungen zustande, es ist wahrscheinlich einfach noch zu langsam.

Kapitel 26

Bezugnahme auf die Abschätzung aus Kapitel 10

Die Abschätzung aus Kapitel 10 wurde gemacht, damit schon vor Beginn des umfangreichsten Teils (Implementierung) des Projektes, eine Annahme darüber gemacht werden konnte, ob es überhaupt gelingen kann oder ob es lieber gleich aufgegeben werden sollte. Aus der Abschätzung ging hervor, dass das Projekt zu einem sinnvollen Ergebnis führen konnte, so war das Risiko eines Fehlschlages von vorn herein besser abschätzbar und das Projekt hatte schon ein gutes Vorzeichen. Das Programmsystem wurde nicht so modifiziert, das es die getroffenen Annahmen aus der Abschätzung in Kapitel 10 erfüllt, da das Programmsystem in der Lage war Bilder in akzeptabler Zeit zu kodieren und eine Realisierung der Annahmen der Abschätzung deshalb nur zusätzlich Zeit benötigt hätte.

Die Dauer der Umsetzung eines Bildes hängt, wie schon erwähnt, wohl eher von der Komplexität der Strukturen auf dem Bild ab, und nicht wie aus der Formel

$$o_{ges} = \sum_{n=1}^{p} (\left\lceil \frac{\log(1 - e^{1/p})}{\log(1 - \frac{n}{k}) * w} \right\rceil * (w + 1))$$

zu vermuten von der Anzahl der Pixel des Bildes, also der möglichen Punkte k (die restlichen Parameter hängen vom Algorithmus und nicht vom Bild ab). Damit hat die Abschätzung, obwohl sie Aussagen zur Realisierbarkeit gemacht hat, wahrscheinlich keine richtige Aussagen darüber getroffen, wieviel Zeit der Algorithmus für die Bildcodierung benötigt. Bei der Abschätzung wurden auch nur die Anzahl der Iterationen betrachtet und nicht die reelle Zeit die zur Bildkodierung benötigt wird, da die Zeit für eine Iteration mit der Bildgröße und fib-Objektgröße zunimmt. Die Annahmen der Abschätzung wurden auch so getroffen, dass ein Festfahren des Algorithmus unmöglich war. Der realisierte Algorithmus hat aber durchaus das Problem, dass er sich in einem lokalem Optimum festfahren kann und dann keine besseren Lösungen mehr liefert.

Daher war die Abschätzung nur dafür sinnvoll, eine Aussage über die prinzipielle Möglichkeit zu liefern, ein Bild mit der Idee zu codieren, nicht aber für

Aussagen über die Zeit, die das realisierte Programmsystem zum codieren wirklich benötigt.

Kapitel 27

Weitere Ergebnisse

Im nachfolgenden sind einige weitere Punkte aufgeführt, die bei den Testdurchläufen relativ stark aufgefallen sind und deshalb erwähnenswert sind.

27.1 Kombination von Bereichs- und Funktionsobjekten

Anscheinend ist die Wahrscheinlichkeit das Bereichs- und Funktionsobjekte sinnvoll kombiniert werden relativ gering, z.B. das beide kombiniert werden um einen schrägen Strich zu zeichnen. Dies läst meine Auswertung von mehreren generierten fib-Objekten vermuten. Fast immer werden Bereichsobjekte, die größere Bereiche definieren, ohne nachgeschaltetes Funktionsobjekt übernommen und Funktionsobjekte haben als Ausgabe nur einen Wert, selbst wenn sie Variablen, die Bereichsobjekte definieren, verwenden.

Das Problem wurde teilweise mit einer "suchenden" Operationen überwunden, die gleich eine Kombination von Bereichs- und Funktionsobjekt in das fib-Objekt einsetzen, welche eine Teilobjekt auf dem Bild zumindest teilweise richtig darstellen. Die Grundidee dieses Operators ist dabei die Erzeugung eines schrägen Striches, es werden zwei Variablen erzeugt die voneinander über eine Funktion abhängen. Beispielsweise könnten die Variablen x und y erzeugt werden, wobei y eine natürliche Zahl ist, die von 1 bis 7 läuft und x sich aus $2*y^3$ ergibt ($x = 2*y^3$).

Es konnte anhand von Testläufen festgestellt werden, dass diese Operation die Performance des Algorithmus verbessert. Es wurden einige Testläufe ohne diese Operation gemacht, die Ergebnisse befinden sich im Verzeichnis "\Tests\sline", dabei wurden nur fib-Objekte gefunden die die Bilder mit Hilfe von einzelnen Punkten codieren. Zum Vergleich wurden dann auch einige Testläufe mit dieser Operation gemacht, die Ergebnisse befinden sich im Verzeichnis "\Tests\sline2". Dabei wurden fib-Objekte mit weniger Iterationen gefunden, die die Bilder mit Hilfe eines "schrägen Striches" realisieren, der aus einer Kombination von Funktions-, Bereichsobjekt und Punkt besteht. Diese fib-Objekte sind zudem kürzer als die fib-Objektdarstellung nur mit Punkten.

27.2 Init mit korrekten Bildern

Es wird eine Initialisierung des Algorithmus mit vollständigen korrekten fib-Bildern angeboten. Dabei wird einfach für jeden Punkt im Originalbild ein Punkt im fib-Objekt an einer zufälligen Stelle eingeführt.

Die Idee dahinter ist, dass gleich von Anfang an korrekte fib-Bilder im Algorithmus vorhanden sind, die dann nur noch weiter zu optimieren sind.

Leider sind diese korrekten Bilder auch eine Art Superindividuen, die veränderte fib-Bilder unterdrücken bzw. sich gegenüber diesen sehr wahrscheinlich durchsetzen. Was dazu führt das die Entstehung von neuen besseren fib-Bildern mit Bereichs- oder Funktionselementen unwahrscheinlicher wird und damit seltener geschieht.

Ein Test dazu ist im Verzeichnis "\Tests\hsw“ zu finden, beim Durchlauf wurden dabei alle Punkte aus dem anfänglich korrekt initialisierten Bild entfernt die die Hintergrundfarbe hatten und damit überflüssig waren. Zu dem Ergebnis wurden dann keine besseren Individuen gefunden.

Vielleicht sind auch die derzeit verwendeten Operatoren und Auswahlmechanismen zum Finden besserer Individuen für korrekt initialisierte Bilder hinderlich, das kann aber leider noch nicht getestet werden. Im allgemeinen ist es wahrscheinlich besser die fib-Objekte langsam wachsen zu lassen.

27.3 Armageddon - Neustart zur Widerbelebung

Ein auffälliges Phänomen war, dass wenn sich der Algorithmus festfuhr, also über lange Zeit keine besseren Individuen mehr lieferte, durch einen Neustart des Algorithmus mit den letzten besten Individuen, oft dann nach kurzer Zeit doch bessere Individuen gefunden wurden, besonders wenn außerdem die Parameter zur Bewertung beim Neustart geändert wurden.

Ein Beispiel ist im Verzeichnis "\Tests\restart“ zu finden, dabei wurden allerdings beim Neustart die Parameter nicht verändert.

Im Nachfolgendem ist ein Ausschnitt der Iterationsanzahlen für aufeinanderfolgende beste Individuen gegeben: 113324; 130088; 131184; 143063; 144951; 149705; 193125; 274963 weiter bis 457000 gelaufen, dann ohne neue beste Individuen erhalten zu haben gestoppt.
Danach wurde ein neuer Durchlauf mit dem alten besten Individuum gestartet, die Iterationzzahlen waren: 160; 2538; 2945; 4936; 5022; 6936; 8831

Während vor dem Abbruch des Algorithmus bei rund 457.000 Iterationen die Abstände zwischen den Iterationen, bei denen neue beste Individuen gefunden wurden, im Mittel schon relativ groß waren, war nach dem Neustart der mittlere Abstand wieder deutlich geringer. Das legt die Idee nahe einen Armageddon Operator zu schaffen, der von Zeit zu Zeit, am besten wenn der Algorithmus anscheinend

festgefahren ist, alle Individuen, außer einige wenige der Besten, löscht und eventuell sogar die Parameter zur Bewertung verändert. Auch in der Natur trifft man das Phänomen an, das nach großen Katastrophen und auch großen Veränderungen die Artenvielfalt, durch den neu geschaffenen Platz, plötzlich ansteigt.

Durch die Veränderung der Parameter kann natürlich auch die Optimierung der Individuen in mehreren Schritten geschehen. Zuerst kann z.B. das Gewicht für den Abstand zum Originalbild sehr hoch gesetzt werden, so das zuerst hauptsächlich darauf optimiert wird, ein Bild zu finden, das dem Originalbild möglichst nahe kommt, und in einer zweiten Phase können die in der ersten Phase gefundenen fib-Objekt weiter optimiert werden, mit einem höheren Gewicht für die Größe der fib-Objekte, so dass die bisher gefundenen fib-Objekte hauptsächlich weiter auf Kürze optimiert werden.

27.4 Unsterbliche Individuen

In der Implementierung ist die Möglichkeit vorhanden, einige der besten Individuen "unsterblich" zu machen. Das heißt, dass die n besten Individuen nicht gelöscht werden können. Wobei $n \geq 0$ wählbar ist. Die Idee dahinter ist, dass die besten Individuen erhalten bleiben und auf sie aufgebaut werden kann. Einige unsterbliche Individuen führen anscheinend schneller zu guten Individuen als keine.
Zum Vergleich wurde ein Durchlauf mit unsterblichen Individuen, zu findet im Verzeichnis "\Tests\imort\", und ein Durchlauf ohne unsterblichen Individuen durchgeführt, zu findet im Verzeichnis "\Tests\no_imort\". Im Durchlauf mit unsterblichen Individuen wurden schneller bessere Individuen gefunden.

Allerdings ist zu befürchten, dass bei einigen unsterblichen Individuen sich der Algorithmus auch schneller festfährt und keine besseren Individuen mehr hervorbringt. Das hängt aber sicherlich noch von anderen Parametern ab.

27.5 Wachsende Individuen

Aus den verschiedenen Tests ist weiterhin zu vermuten, dass es für den Algorithmus im Allgemeinen besser ist, wenn er mit kleineren Individuen beginnt, die dann nach und nach größer werden/wachsen.

Es ist bei den Durchläufen aufgefallen, dass es anscheinend viel unwahrscheinlicher ist, dass der Algorithmus alte Teilobjekte in einzelnen Individuen durch neue bessere Teilobjekte ersetzt, als dass er bessere Teilobjekt, für noch nicht gefundene Teilbilder, ganz neu an ein Individuen anfügt. Deutlich macht das besonders der Test für die Initialisierung mit korrekten Bildern, da im allgemeinen nicht komplexere Teilobjekte eingefügt wurden, die nicht nur aus Punktobjekten und conc Objekten bestehen, und dafür dann überflüssige Punkte gelöscht wurden.

Dies läuft leider auch der Annahme aus 11.3 zuwider.

Warum das so ist und wie es umgangen werden kann, sollte genauer untersucht werden. Interessant ist auch die Frage ob dieses Phänomen auch in der Natur auftritt. Entstehen eventuell neue Arten eher aus weniger spezialisierten Arten, als aus mehr spezialisierten Arten?

Kapitel 28

Zusammenfassung

Zusammengefasst wurde ein Programmsystem geschaffen, mit dem es möglich ist, zumindest Bitmapbilder mit einfachen Strukturen in Vektorbilder einer selbstdefinierten Vektorbildbeschreibungssprache fib umzuwandeln.

Das in Aufgabenstellung beschriebene Ziel, ein Programmsystem zu schaffen, das beliebige Bitmapbilder in platzsparende Vektorbilder umwandelt und eventuell in diesen Vektorbildern Objekte mehrfach recycelt, ist (noch) nicht ganz erreicht worden. Trotzdem denke ich das die vorliegende Arbeit durchaus einen Nutzen gebracht hat. Es wurden (zumindest für mich) neuen Erkenntnissen über evolutionäre Algorithmen gewonnen.

Das Programmsystem wurde auf Erweiterbarkeit ausgelegt, so dass es in Zukunft auch weiter ausgebaut und verbessert werden kann. Damit sind auch gute Möglichkeiten zum Experimentieren mit ihm gegeben.

Kapitel 29

Ausblick/Bewertung

Die potentiellen Möglichkeiten, die in der Idee stecken, sind noch sehr schwer auszuloten. Ich vermute aber, dass es durchaus möglich ist, das Programmsystem so zu erweitern und zu verändern, dass es mit ihm durchführbar ist, gezeichnete Bilder (z.B. aus Zeichentrickfilmen) mit Abmessungen in der Größenordnung von 1.000.000 Pixeln (1.000 mal 1.000 Pixel) und eine Anzahl von Farben in der Größenordung von 256 Farben in einigen Stunden um etwa den Faktor 30, im Vergleich zu Bitmapbildern, zu komprimieren.

Durch den hohen Komprimierungsaufwand und die hohe Komprimierung, ist der Ansatz vor allem dort von Interesse, wo möglichst wenig Speicher- oder Übertragungskapazität verbraucht werden soll, aber genügend Rechenleistung zur Verfügung steht oder das Ergebnis sehr oft wiederverwendet wird. Günstig wäre das Verfahren zur Kompression von Bildern im Internetbereich, z.B. für Bilder auf Internetseiten. Für das Bild ist nur einmal eine Komprimierung nötig, aber das erzeugte Bild schont viel Bandbreite, wegen seiner geringeren Größe. Im Internetbereich werden auch viele gezeichnete Bilder verwendet (Logos, Werbebanner), ein weiterer Punkt der für fib spricht. Eventuell ist nicht einmal eine Komprimierung nötig, wenn die Bilder gleich im fib-Format erzeugt werden, aber selbst für im fib-Format erzeugte Bilder, könnte eine weitere Komprimierung mit den Algorithmus möglich sein.

Gut gelungen finde ich die Bildbeschreibungssprache fib, da sie es erlaubt, mit sehr einfachen Strukturen, Bildobjekte und Zusammenhänge zwischen diesen darzustellen, ohne dass dabei die fib-Objekte sehr groß werden. Außerdem ist sie auch gut für die Verwendung in einem genetischen Algorithmus geeignet und es ist für den Menschen leichter möglich fib-Objekte zu verstehen, als z.B. wenn ich eine Bit Repräsentation gewählt hätte. Durch die Einfachheit des fib-Format (im Gegensatz z.B. zum JPG Format), ist auch gut eine Verbreitung dieses Formates möglich.

Kapitel 30

Mithelfer

Bei der vorliegenden Arbeit haben mir einige Personen geholfen.

Mein Dank geht deshalb an Susanne Grell, für ihre Mithilfe bei der Kontrolle von Quelltext und der Dokumentation, meiner Mutter Hannelore Österholz, für die sprachliche Kontrolle (Rechtschreibung und Ausdruck) der Dokumentation, an Torsten Schaub für Anregungen und Material, an Pauline Kraak für mathematische Rückenstärkung bei der Abschätzung aus Teil III und Jens Calamé der mir anfängliche Hilfe beim Debugging des Programmsystems gab.

Kapitel 31

Eidesstattliche Erklärung

Ich versichere hiermit das die vorliegende Arbeit und die verwendeten Ideen, soweit nicht anders gekennzeichnet, von mir stammen. Ähnliche Projekte sind mir nicht bekannt und wurden damit von mir auch nicht als Teile oder Vollständig genutzt. Die Vermutung das es keine ähnlichen Projekte bisher gab, ist aber leider nur eine Vermutung, da es meines Wissens keine Möglichkeit gibt Aussagen dieser Art mit völliger Gewissheit zu treffen.

Literaturverzeichnis

[1] *Einführung in die Mikrobiologie*. http://www.biologie.hu-berlin.de/~baktgen/texte/Vorlesungen/Mikro/3_gen%_trans.pdf.

[2] *Genetic Programming for Feature Discovery and Image Discrimination Walter Alden Tackett*. http://citeseer.ist.psu.edu/tackett93genetic.html.

[3] *Genetische und Evolutionäre Algorithmen*. http://www.bwl.uni-mannheim.de/Heinzl/de/downloads/ki-kapitel-4-6.pdf, WS 2002/03.

[4] BAUER, HEINZ: *Wahrscheinlichkeitstheorie*. New York, 4 Auflage, 1991.

[5] BEHRENDS, ERHARD: *Überall Zufall: Eine Einführung in die Wahrscheinlichkeitsrechnung*. Mannheim, 1994.

[6] CANNON, W.D.: *The Wisdom of the Body*. W.W. Norton, New York, 1932.

[7] DARWIN, CHARLES: *Die Entstehung der Arten*. Nikol, 1963.

[8] FOGEL, DAVID B.: *Evolutionary Computation - The Fossil Record*. New York, 1998.

[9] FOGEL, DAVID B.: *Evolutionary computation: toward a new philosophy of machine intelligence - 2nd ed.* Institute of Electrical and Electronics Engineers, 2000.

[10] HEITKÖTTER, JÖRG und DAVID BEASLEY: *The hitch-Hiker's Guide to Evolutionary Computation*. http://surf.de.uu.net/hhg2ec/.

[11] HOLLAND, JOHN H.: *Genetic Programming*. 4 Auflage.

[12] MICHALEWICZ, ZBIGNIEW: *Genocop – Optimization via Genetic Algorithms*. http://www.cs.sunysb.edu/~algorith/implement/genocop/implement.shtml.

[13] MICHALEWICZ, ZBIGNIEW: *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, Third Revision Auflage, 1996.

[14] NAWROTZKI, KURT: *Lehrbuch der Stochastik: eine Einführung in die Wahrscheinlichkeitstheorie und die mathematische Statistik*. Harri Deutsche, 1994.

[15] PFIESTER, PAULA, NINA TRETTER, TINA KOPPENHÖFER, MATTHIAS ECKER und PROJEKTBETREUERIN ANNETT STEINER: *Projekt Bildformate*. `http://www.scheib.info/downloads/Projekt_Bildformate.pdf`, 10.2001-2.2002.

[16] POOLE, DAVID, ALAN MACKWORTH und RANDY GOEBEL: *Computational intelligence: a logical approach*. Oxford University Press, 1998.

[17] SCHMID, FRIEDRICH und MARK TREDE: *Skript zur Vorlesung: Einführung in die Stochastik der Finanzmärkte: Stochastische Prozesse, Simulation und Anwendungen*. `http://www.wiwi.uni-muenster.de/~05/hauptstudium/vorlesungen/stofin/sto%fin.pdf`, SS 2000.

[18] TELLER, ASTRO und MANUELA VELOSO: *Algorithm Evolution for Face Recognition: What Makes a Picture Difficult*. `http://www.cs.cmu.edu/~mmv/papers/tellerICEC95.pdf`.

[19] TELLER, ASTRO und MANUELA M. VELOSO: *A Controlled Experiment: Evolution for Learning Difficult Image Classification*. In: *Proceedings of the Seventh Portuguese Conference on Artificial Intelligence*, Seiten 165–176. Springer Verlag, October 1995.

[20] TURING, A.M.: *Computing Machinery and Intelligence*. Mind, 59, 1950.

[21] WERNER, BODO: *Einführung in elementare Stochastik - Wahrscheinlichkeitsrechnung und Statistik - Angewandte Mathematik für das Lehramt an Grund- und Mittelstufe sowie an Sonderschulen*. `http://www.math.uni-hamburg.de/home/werner/Stochastik.pdf`, WS 2002/2003.

[22] WIELAND, THOMAS: *C++ Entwicklung mit Linux*. dpunkt.verlag, 2001.