Endbericht

Projektgruppe 427

Evolutionäre Algorithmen zwischen experimenteller und theoretischer Analyse

Patrick Briest, Dimo Brockhoff
Bastian Degener, Matthias Englert
Christian Gunia, Oliver Heering
Michael Leifhelm, Kai Plociennik
Heiko Röglin, Andrea Schweer
Dirk Sudholt, Stefan Tannenbaum

Thomas Jansen, Ingo Wegener

16. Februar 2004

Inhaltsverzeichnis

1 Einleitung **5**

2 Zielsetzung **7**
2.1 Minimalziel. . . 7
2.2 . . . und darüber hinaus 8
2.3 Zeitplan 9
2.4 Gruppeneinteilung und Arbeitsweise 10
2.5 Organisatorisches 11

3 Erste Seminarphase **13**
3.1 Einführung, Suchräume, Fitness, Restriktionen, NFL 13
3.2 Selektionsoperatoren und multikriterielle Optimierung 15
3.3 Mutationsoperatoren und Diversitätserhaltung 18
3.4 Rekombination und Schema Theorie 22
3.5 Parametrisierung und Parametersteuerung 27
3.6 Statistische Methoden, Wahrscheinlichkeitsrechnung 28
3.7 Black-Box Komplexität 33
3.8 Methoden zur Analyse evolutionärer Algorithmen 35
3.9 $(1+\lambda)$ EA 38
3.10 Maximale Matchings 42

4 Analyse- und Designphase **45**
4.1 Analyse-Phase 45
4.2 Design-Phase 61

5 Implementierungsphase **69**
5.1 Allgemeines 69
5.2 Ablauf 72
5.3 Testen 74
5.4 Implementierungsdetails 75

6 Dokumentationsphase **85**

7 Zweite Seminarphase **87**
7.1 Minimale Spannbäume 87
7.2 Experimentbewertung 88

7.3 Das Ising-Modell auf dem Ring . . . 90
7.4 Das Ising-Modell auf dem quadratischen Torus . . . 93
7.5 Parametrisierung . . . 95
7.6 Stoppkriterien . . . 97
7.7 Pseudozufallszahlengeneratoren . . . 100
7.8 Maximale Matchings . . . 103
7.9 Multikriterielle Optimierung . . . 104
7.10 Nutzen von Crossover und echten Populationen . . . 110
7.11 Monotone Polynome . . . 112

8 Hypothesenfindung 115
8.1 Das Ising-Modell . . . 116
8.1.1 Das Ising-Modell auf Cliquen . . . 116
8.1.2 Das Ising-Modell auf Tori . . . 129
8.1.3 Das Ising-Modell auf booleschen Hypercubes . . . 132
8.2 Minimale Spannbäume . . . 136
8.2.1 Experimente auf einem asymptotischen Worst-Case-Beispiel . . . 136
8.2.2 Experimente zu Kantengewichten . . . 139
8.2.3 Experimente zum Vergleich der Algorithmen (1+1) EA und RLS . . . 140
8.2.4 Experimente zur Darstellung von Spannbäumen als Prüfernummern . 141
8.3 Maximale Matchings . . . 143
8.3.1 Matchings auf Bäumen und Pfaden . . . 143
8.3.2 Nutzen von Uniform Crossover bei Matchings auf Pfaden . . . 144
8.3.3 Matchings auf Semi-Random-Graphen . . . 145

9 Hypothesentests 147
9.1 Das Ising-Modell . . . 148
9.1.1 Das Ising-Modell auf Cliquen . . . 148
9.1.2 Das Ising-Modell auf Tori . . . 160
9.1.3 Das Ising-Modell auf booleschen Hypercubes . . . 165
9.2 Minimale Spannbäume . . . 174
9.2.1 Experimente auf einem asymptotischen Worst-Case-Beispiel . . . 174
9.2.2 Experimente zu Kantengewichten . . . 176
9.2.3 Experimente zum Vergleich der Algorithmen (1+1) EA und RLS . . . 181
9.2.4 Experimente zur Darstellung von Spannbäumen als Prüfernummern . 188
9.2.5 Vergleich zweier Fitnessfunktionen . . . 193
9.3 Maximale Matchings . . . 199
9.3.1 Matchings auf Bäumen und Pfaden . . . 199
9.3.2 Nutzen von Uniformem Crossover bei Matchings auf Pfaden . . . 207
9.3.3 Matchings auf Semi-Random-Graphen . . . 210
9.4 Kürzeste Wege in Graphen . . . 213
9.4.1 Suchraum, Fitnessfunktionen und Algorithmus . . . 213
9.4.2 Experimente . . . 214
9.4.3 Ergebnisse . . . 214

Kapitel 1

Einleitung

Dieser Endbericht der Projektgruppe 427 „Evolutionäre Algorithmen zwischen experimenteller und theoretischer Analyse“ des Lehrstuhles 2 der Universität Dortmund dient dazu, den Verlauf und die Ergebnisse der Lehrveranstaltung aus dem Sommersemester 2003 und dem Wintersemester 2003/2004 vorzustellen.

Im Wesentlichen bestand die Aufgabe des ersten Projektgruppensemesters darin, ein Softwareprodukt zu entwickeln, mit dem das Testen von Hypothesen über evolutionäre Algorithmen effizient und benutzerfreundlich ermöglicht wird. Das erstellte Softwareprodukt bekam nach langer Diskussion den Namen FrEAK, ein Akronym für „Free Evolutionary Algorithm Kit“.

Das anschließende zweite Projektgruppensemester war dann dafür vorgesehen, das im ersten Semester erstellte Programmpaket zu benutzen, um theoretisch begründete Vorhersagen experimentell zu überprüfen und durch weitergehende Experimente neue Voraussagen zu machen. In diesem Zusammenhang wurde FrEAK um einige Features erweitert.

Während der Projektgruppe galt es, ein Minimalziel zu erfüllen, das in Kapitel 2 vorgestellt wird. Die folgenden Kapitel geben die chronologische Reihenfolge der Arbeitsphasen wieder, wie sie im selbst erstellten Zeitplan in Abschnitt 2.3 beschrieben sind.

Die Projektgruppenarbeit startete mit einem Blockseminar im April 2003, zu dem jeder Teilnehmende einen Vortrag zu einem ausgewählten Thema beitrug. Eine kurze Zusammenfassung der einzelnen Vorträge, die insgesamt einen Einstieg in das Thema „Evolutionäre Algorithmen“ bieten, findet man in Kapitel 3. Die anschließenden Kapitel widmen sich den üblichen Phasen beim Softwareentwurf: In der Analyse- und Designphase (Kapitel 4) wurden die wesentlichen Konzepte von FrEAK erarbeitet. Anschließend folgte die Implementierungsphase, in der neben der eigentlichen Implementierung einige wenige grundlegende Designentscheidungen wieder verworfen wurden. Über vorgenommene Änderungen und Gründe dafür informiert Kapitel 5. Zum Ende des Semesters begann zeitgleich mit dem Ausklingen der Implementierungsphase die Test- und Dokumentationsphase, deren Beschreibung in Kapitel 6 zu finden ist.

Im zweiten Semester fand eine zweite Seminarphase statt, in der wöchentlich Referate über relevante Themen von PG-Teilnehmern gehalten wurden. Die Zusammenfassungen der Vorträge

sind in Kapitel 7 zu finden. Parallel dazu wurden Experimente zu verschiedenen Themengebieten durchgeführt, die in den Kapiteln 8 und 9 beschrieben werden.

Zusätzlich zu den Vorträgen und den Experimenten wurde FrEAK weiterentwickelt, um die Experimente zu unterstützen. In diesem Zusammenhang wurden auch viele neue Features implementiert. Die Beschreibung der wesentlichen Weiterentwicklungen im zweiten Projektgruppensemester ist in die Kapitel 4 und 5 eingebettet.

Kapitel 2

Zielsetzung

Inhalt

2.1 Minimalziel. 7
2.2 . . . und darüber hinaus 8
2.3 Zeitplan 9
2.4 Gruppeneinteilung und Arbeitsweise 10
2.5 Organisatorisches 11
2.5.1 Gruppentreffen 11
2.5.2 Programmiersprache 11
2.5.3 Softwarevertrag 11
2.5.4 Name 11
2.5.5 Tools 12

2.1 Minimalziel. . .

Folgendes Minimalziel ist im Projektgruppenantrag vorgesehen:

> Minimalziel ist die Implementierung einer Plattform, welche die Kombination einiger Operatoren zu lauffähigen evolutionären Algorithmen erlaubt. Es sind mindestens Bitstrings fester Länge und Permutationen als Suchräume und dazu passende Variationsoperatoren zu implementieren. Es sind Fitnessfunktionen für einige wichtige Probleme zu implementieren und Experimente mit den Algorithmen auf diesen Problemen müssen mit verschiedenen Parametereinstellungen möglich sein und durchgeführt werden. Notwendig ist schließlich eine Analyse der durchgeführten Experimente sowie die Überprüfung einiger Hypothesen.

Die geforderten Ziele bezüglich der Plattform wurden im ersten Semester mit der Erstellung von FrEAK erfüllt. Die Durchführung von Experimenten und deren Analyse sowie die Überprüfung von Hypothesen fand im zweiten Semester statt.

2.2 . . . und darüber hinaus

FrEAK bietet noch weit über das Minimalziel hinausgehende Funktionalität. Neben Bitstrings fester Länge und Permutationen unterstützt FrEAK auch Zyklen, Kantenauswahlen auf Graphen und Strings mit endlichem Alphabet als weitere Suchräume. Es können zudem unproblematisch weitere endliche Suchräume durch Anwender implementiert werden.

Es sind deutlich mehr als nur einige Fitnessfunktionen und Variationsoperatoren implementiert.

Der Entwurf evolutionärer Algorithmen mit FrEAK muss nicht, wie gewohnt, einem festen Schema (Initialisierung, wiederholte Anwendung von Selektion zur Reproduktion, Variation und Selektion zur Ersetzung) folgen, sondern geschieht mittels einer intuitiven graphischen Oberfläche. Die Algorithmen werden dabei durch Graphen modelliert. Knoten dieser Graphen entsprechen den Operatoren des Algorithmus wie zum Beispiel Mutation, Selektion, Rekombination. Die gerichteten Kanten symbolisieren den zeitlichen Ablauf des Algorithmus, wobei Individuen den Kanten folgen. So können beliebige evolutionäre Algorithmen einfach per Drag&Drop zusammengestellt werden. Details finden Sie in Abschnitt 4.1.5.

Ein interessantes Feature von FrEAK ist die Replayfunktion zum Reproduzieren von Simulationsläufen der erstellten Algorithmen. Wie dies genau funktioniert, steht im Abschnitt 4.1.4. Des Weiteren können mit so genannten Batches mehrere ähnliche Simulationen automatisch ausgeführt werden. Außerdem bietet FrEAK die Möglichkeit, mehrere Stoppkriterien zu kombinieren, und unterstützt auch multikriterielle Optimierung.

Die graphische Oberfläche ist darauf ausgelegt, dem Anwender einen leichten Einstieg zu ermöglichen, bietet aber dem erfahrenen Benutzer eine unkomplizierte Nutzung aller Features. Zur einfachen Auswertung der Analysedaten stellt FrEAK eine Vielzahl von Visualisierungen zur Verfügung.

Zusätzlich existiert eine Möglichkeit, FrEAK ohne die graphische Oberfläche zu starten. Dieser Modus ist nicht interaktiv und dient dazu, langwierige Simulationen über Nacht oder im Hintergrund laufen lassen zu können.

Eine wichtige Eigenschaft von FrEAK ist seine Flexibilität. Der Anwender kann relativ unproblematisch Module seiner Wahl implementieren und diese werden von FrEAK automatisch eingebunden. Insbesondere Fitnessfunktionen, Variationsoperatoren, Auswertungsmodule (so genannte Observer), Visualisierungen (im Folgenden Views genannt), Stoppkriterien und Suchräume können jederzeit ergänzt werden. Um dies zu gewährleisten, wurde eine ausführliche Dokumentation geschrieben. Des Weiteren ist ein Handbuch für die Benutzung von FrEAK verfügbar.

Zum Ende des ersten Semesters wurde FrEAK im Rahmen des Diplomandenseminars des Lehrstuhls 2 vorgestellt, um einem Großteil der potentiellen Benutzer der Software den Einstieg zu erleichtern.

2.3 Zeitplan

Folgende Tabelle zeigt den geplanten Zeitverlauf des ersten Semesters nach der Seminarphase. Die vorgegebenen Deadlines haben die Teilnehmenden sich selber gesetzt und sich bemüht, diese einzuhalten.

Deadline	Aufgabe	Zeitrahmen
16.5.2003	Analyse abgeschlossen	3,5 Wochen
30.5.2003	Design abgeschlossen	2 Wochen
20.6.2003	Implementierung abgeschlossen	3 Wochen
11.7.2003	Handbuch fertig	3 Wochen
11.7.2003	Tests abgeschlossen	–
1.8.2003	Zwischenbericht fertig	3 Wochen

Sowohl die Deadline der Analysephase als auch die der Designphase wurde eingehalten. Ohne eine Deadline hätte die Gefahr bestanden, dass die Teilnehmenden sich sowohl für die Analyse- wie auch der Designphase mehr Zeit gelassen und etliche zusätzliche Features modelliert hätten. Aus diesem Grund hätte dann womöglich nicht mehr genügend Zeit zur Verfügung gestanden, um FrEAK bis zum Semesterende mit dem damaligen Funktionsumfang zu realisieren. In dieser Hinsicht hat sich der selbst gesteckte Zeitplan als sehr nützlich erwiesen.

Ab der Implementierungsphase konnten die hoch gesteckten Ziele nicht mehr eingehalten werden. Zum vorgesehenen Ende der Implementierungsphase war FrEAK zwar bereits in einer Minimalversion benutzbar, aber noch nicht für eine tatsächliche Analyseaufgabe einsetzbar. Allerdings ist die Implementierungsphase bei FrEAK aufgrund der ausdrücklichen Erweiterbarkeit nicht endgültig abzuschließen.

Die weiteren Phasen Handbuch, Testen und Zwischenbericht gingen ineinander über und hielten sich nur noch grob an die vorgesehenen Zeiträume. Die Prioritäten der unerledigten Aufgaben wurden aber immer im Hinblick auf die Deadlines vergeben.

Im zweiten Semester wurden keine expliziten Deadlines mehr gesetzt. Durch die Termine der Vorträge der zweiten Seminarphase wurde aber implizit der zeitliche Ablauf der Experimente bestimmt.

Datum Teil 1	Datum Teil 2	Vortrag
16.10.2003	23.10.2003	Minimale Spannbäume
30.10.2003	6.11.2003	Das Ising-Modell
11.12.2003	18.12.2003	Maximale Matchings
7.1.2004	22.1.2004	Multikriterielle Optimierung

Eine Ausnahme bildeten die letzten Wochen des Semesters, als die drängende Zeit doch einige Deadlines notwendig machte. So wurden für die einzelnen Bereiche Endbericht, FrEAK-Release 0.2 und Vortrag innerhalb des Diplomandenseminars gezielt Deadlines bestimmt. Diese wurden aber alle eingehalten.

2.4 Gruppeneinteilung und Arbeitsweise

In den ersten beiden Phasen der Analyse und des Design wurde überwiegend im Plenum gearbeitet. Lediglich Teilaufgaben sind an kleinere Gruppen oder Einzelpersonen delegiert worden. Dazu gehörten unter anderem

- graphische Darstellungen von evolutionären Algorithmen in Operatorgraphen,
- Zufallszahlengeneratoren,
- Installation der Softwaretools (siehe Abschnitt 2.5.5),
- zu realisierende Module: Fitnessfunktionen, Operatoren, Observer/Views (diese Begriffe werden im Kapitel 4 genauer erläutert).

In der Implementierungsphase stellten die Teilnehmenden fest, dass sich die zu erledigenden Aufgaben gut in folgende Teilbereiche aufteilen ließen, die inhaltlich zusammen gehören:

Core: Kernfunktionalität von FrEAK (z. B. Ablaufsteuerung, Replay, ...)

GUI: Graphisches Userinterface

Observer: Erstellen von Modulen zur Berechnung und Visualisierung von Analysedaten

Rest: sonstige Module, um die FrEAK erweitert werden kann

Die Gruppengrößen orientierten sich an dem erwarteten Arbeitsaufwand. Der GUI-Gruppe wurden zunächst drei Teilnehmer fest zugeordnet, der Coregruppe vier, der Observergruppe zwei und der Restgruppe drei Personen. Tatsächlich haben sich die Gruppen bezüglich der Personen dynamisch verändert, während die Aufgaben den einzelnen Gruppen weiterhin fest zugeordnet blieben.

In der Test- und Dokumentationsphase haben sich die Arbeitsgruppen nahezu automatisch dadurch ergeben, dass viele Aufgaben durch diejenigen Personen erledigt wurden, die auch zuvor die zu testenden bzw. zu dokumentierenden Programmteile erstellt haben. Die verbliebenen Aufgaben, wie zum Beispiel der Zwischenbericht, wurden von Teilnehmern übernommen, die nicht mehr in der Fehlerbehebung eingebunden waren. Bei der Dokumentation haben sich die Teilnehmenden den Dokumentkapiteln entsprechend in Gruppen aufgeteilt.

Im zweiten Projektgruppensemester wurde vor den Vorträgen, zu denen Experimente vorgesehen waren, die Teilnehmer zufällig in drei etwa gleich große Kleingruppen aufgeteilt, die bestimmte Fragestellungen im Vorhinein bearbeitet haben. Unmittelbar vor den Vorträgen wurden die Ergebnisse der theoretischen Überlegungen verglichen. Fast immer ergaben sich dann hieraus und aus den jeweiligen Vorträgen weitergehende Fragestellungen, zu denen dann Experimente geplant wurden. Der zugehörige Experimentaufbau wurde von Kleingruppen geplant, die aus Teilnehmern bestanden, die an dem jeweiligen Experiment besonderes Interesse hatten. Meistens wurden die Experimente dann auch von den selben Teilnehmern durchgeführt und ausgewertet. Zwischen den verschiedenen Experimentphasen wurde der Fortschritt jeweils im Plenum vorgestellt und das weitere Vorgehen diskutiert. Die gewünschten Anpassungen von FrEAK wurden in diesem Zusammenhang meistens auch von den jeweiligen Teilnehmern vollzogen, die diese auch benötigten.

2.5 Organisatorisches

2.5.1 Gruppentreffen

Von jedem Treffen im Plenum wurde ein Protokoll angefertigt, um die gefassten Beschlüsse zu dokumentieren. Des Weiteren wurde die Diskussion von einem Moderator geleitet und zu Beginn einer jeden Sitzung eine Tagesordnung festgelegt. Protokollant und Moderator wurden bei den Sitzungen nach einem festen Schema gewechselt, so dass jeder Teilnehmende mehrfach protokollierte und moderierte.

In der Analyse- und Designphase fanden die Treffen zunächst dreimal wöchentlich statt. In den nachfolgenden Phasen hat sich die Hauptarbeit auf die Kleingruppen verlagert und die Plenumssitzungen fanden nur noch zweimal wöchentlich statt.

Im zweiten Projektgruppensemester fand ein vierstündiges wöchentliches Plenumtreffen statt, in dem auch jeweils ein etwa 60-minütiger Vortrag von einem Teilnehmer gehalten wurde.

2.5.2 Programmiersprache

Früh haben sich die Teilnehmenden auf Java als Sprache geeinigt, da sie die einzige Programmiersprache ist, die prinzipiell bereits jeder Teilnehmende durch sein Studium in Dortmund zu Beginn der Projektgruppe beherrschen sollte. Aus den gleichen Gründen wurde als GUI-Toolkit Java Swing gewählt.

Der relativ hohe Plattformunabhängigkeit von Java hat sich während der PG als großer Vorteil erwiesen. Leider ergaben sich jedoch mit Swing eine Reihe von schwerwiegenden Problemen, die nur mit großem Aufwand umgangen werden konnten.

2.5.3 Softwarevertrag

Alle Beteiligten und Betreuer waren sich einig, dass FrEAK frei zur Verfügung stehen sollte. Daher wird FrEAK unter der GNU General Public License (GPL) veröffentlicht. Das Copyright an FrEAK wurde durch die Unterzeichnung eines Softwarevertrages an die Gruppe abgetreten. Lizenzänderungen sind damit in Zukunft zwar möglich, können aber nur im Plenum beschlossen werden. Inzwischen gibt es auch zwei Releases von FrEAK.

2.5.4 Name

Ein wochenlanger Findungsprozess war vonnöten, um einen geeigneten Namen für das geplante Softwareprodukt zu finden, der alle zufrieden stellt. Das Ergebnis, FrEAK, ist ein Akronym für „Free Evolutionary Algorithm Kit“ und steht symbolisch für die Teilnehmenden der Projektgruppe.

2.5.5 Tools

Als erstes einigten sich die Teilnehmenden auf das gemeinsame Dokumentenformat LaTeX bzw. PDF. Dieses gestattet die einfache elektronische Übertragung von erstellten Dokumenten, sowie die Benutzung auf verschiedenen Betriebssystemen.

Um die erstellten Dokumente gemeinsam nutzen zu können, wurde das Revisionsverwaltungssystem CVS benutzt. Während der Analyse- und Designphase war Together das unterstützende Tool, mit dem die gesamten UML-Diagramme erstellt wurden.

In der Implementierungsphase kamen dann Eclipse und Netbeans als Entwicklungsumgebungen zum Einsatz. Zur Gestaltung der graphischen Oberfläche wurden allerdings Netbeans und JBuilder benutzt, da Eclipse eine komfortable Gestaltung mittels Drag&Drop nicht unterstützt. Ebenfalls wurde das Tool Javadoc eingesetzt, um ein einheitliches Dokument über die Programmcodedokumentation zu erstellen.

Bei der Durchführung der Experimente kam, wie geplant, FrEAK zum Einsatz. Sehr rechenaufwändige Experimente mussten allerdings manchmal separat implementiert werden und verwendeten nur noch Komponenten von FrEAK.

Die statistische Auswertung der Ergebnisse geschah mit SPSS, OpenOffice und Gnuplot.

Kapitel 3

Erste Seminarphase

3.1 Einführung, Suchräume, Fitness, Restriktionen, NFL

Vortragender: Bastian Degener

Literatur: Der Abschnitt 3.1.5 basiert auf den Arbeiten [15], [49], [40] und [4].

Dieser erste Vortrag der Seminars stellt eine kurze Einführung in das Thema „Evolutionäre Algorithmen“ dar, in dem einige grundlegende Begriffe erläutert werden.

3.1.1 Einführung

Evolutionäre Algorithmen sind allgemeine randomisierte Algorithmen für Optimierungsprobleme, die in immer gleichen, diskreten Runden arbeiten. Sie lehnen sich in der Arbeitsweise an die Evolution der Natur an und haben deshalb folgende Form:

1. Initialisierung
2. Selektion zur Reproduktion
3. Variation
4. Selektion zur Ersetzung
5. Falls Stoppkriterium nicht erreicht, weiter bei 2.

Die zu optimierende Funktion ist dem Algorithmus nicht direkt bekannt und Informationen über sie können nur durch Funktionsauswertungen an ausgewählten „Suchpunkten“ erlangt werden, wobei man den jeweiligen Funktionswert „Fitness“ nennt. Bei den betrachteten Suchpunkten spricht man auch von Individuen, von denen mehrere eine Population bilden, die man innerhalb eines Schleifendurchlaufes auch Generation nennt. Es wird erwartet, dass sich Individuen mit einem guten Fitnesswert tendenziell „näher“ an einem Optimum befinden als Individuen mit schlechten Fitnesswerten. Deshalb werden bei der Selektion zur Reproduktion

gute Individuen bevorzugt und in der Hoffnung, eine weitere Verbesserung zur erzielen, variiert. Anschließend werden von den so neu erzeugten Individuen eventuell einige ausgewählt, die alte Indivduen ersetzen.

3.1.2 Suchräume

Bei der praktischen Anwendung von evolutionären Algorithmen kommt es vor, dass der Suchraum, auf dem das Problem beschrieben ist, nicht dem Suchraum entspricht, auf dem der Algorithmus arbeitet. Eine geeignete Codierung zu finden ist für den möglichen Erfolg des Algorithmus unabdingbar. Insbesondere muss beachtet werden, dass in dem Suchraum vorhandene Semantik nicht verloren geht und syntaktisch eng beieinander liegende Information auch semantisch verknüpft ist.

3.1.3 Fitness

Fitnessfunktionen sollten möglichst effizient auswertbar sein. Manchmal gibt es keine absolut definierte Fitness, oder sie lässt sich nicht effizient evaluieren. Dies ist zum Beispiel bei Spielstrategien der Fall. Dann wird eine relative Fitness berechnet oder durch Experimente, wie zum Beispiel Turniere, ein Schätzwert ermittelt.

3.1.4 Restriktionen

Fitnessfunktionen können unter Umständen nur auf Teilen des Suchraumes definiert sein. Die Erzeugung unzulässiger Lösungen durch die Variationsoperatoren muss dann entweder vermieden, oder auf andere Art nachträglich behandelt werden. Das kann unter anderem dadurch geschehen, dass inkorrekte auf korrekte Lösungen abgebildet werden, die dabei möglichst „ähnlich“ sein sollen. Eine breite Streuung ist von Vorteil, um Diversität zu erhalten. Falls unzulässige Lösungen einen natürlichen Fitnesswert haben, können Restriktionen alternativ auch durch die Addition eines Strafbetrages im Fitnesswert erzwungen werden.

3.1.5 Das NFL Theorem

Das No Free Lunch Theorem ist einer der zentralen Sätze im Gebiet der evolutionäre Algorithmen und zerstörte die großen Erwartungen, die in der Pionierzeit in sie gesetzt wurden. Viele Wissenschaftler teilten anfangs die Auffassung, die von Goldberg [15] und Anderen vertreten wurde, dass evolutionäre Algorithmen Wunderwaffen für algorithmische Probleme seien.

Die Kernaussage war, dass spezielle Suchverfahren zwar auf dem jeweiligen Problem, für das sie konzipiert wurden, extrem gut seien, aber auf anderen Optimierungsproblemen versagen. Geeignete allgemeine Suchverfahren, wie zum Beispiel evolutionäre Algorithmen seien dagegen im Mittel auf allen Problemen zumindest gut, obwohl man natürlich nicht erwarten könne, dass sie die speziellen Suchverfahren auf ihrem jeweilgen Problem, wie zum Beispiel

Heapsort auf dem Sortierproblem, schlagen könnten. Das könne man aber selbstverständlich auch nicht fordern.

Goldberg [15] hat seine These äußerst vage formuliert. Wolpert und Macready [49] waren die ersten, die die Aussage „alle Probleme“ formalisiert und damit die Behauptung auch widerlegt haben. Folgendes Szenario wird No Free Lunch Szenario genannt:

Definition NFL-Szenario:

- Sei $F = \{f : S \to W\}$, wobei der Suchraum S und der Wertebereich W endlich sind.
- Wähle Algorithmus A auf F.
- $A(f)$ ist die Anzahl der verschiedenen Suchpunkte, bis ein Optimum gefunden wurde. Bei Randomisierung ist $A(f)$ der entsprechende Erwartungswert.
- $A_{S,W}$ ist der Durchschnitt aller $A(f)$, $f \in F$.

Satz (NFL Theorem): Im NFL-Szenario gilt: Für zwei beliebige Algorithmen A und A' ist $A_{S,W} = A'_{S,W}$

Der Beweis von diesem auf den ersten Blick erstaunlichen Theorem ist relativ einfach. Allerdings sind die Voraussetzungen unrealistisch, was schon dadurch ersichtlich wird, dass eine beliebige Aufzählung des Suchraumes in diesem Szenario optimal wäre, was unseren Alltagserfahrungen widerspricht.

Später haben Schumacher, Vose und Whitley [40] das Theorem folgendermaßen verschärft: Das NFL gilt genau dann, wenn die Menge der betrachteten Funktionen gegen Permutationen abgeschlossen ist. Dabei heißt eine Menge von Funktionen gegen Permutationen abgeschlossen, wenn für eine beliebige Funktion aus der Menge gilt, dass auch die Funktion, die sich durch Permutation des Suchraumes ergibt, zu der Menge gehört. Anschaulich bedeutet das, dass diese Mengen eine Struktur haben, die einem Algorithmus keinerlei Anhaltspunkte bietet, wo das Optimum liegen könnte.

Daraufhin haben Igel und Toussaint [4] über Abzählargumente gezeigt, dass es nur wenige Klassen von Funktionen gibt, die gegen Permutationen abgeschlossen sind. Diese Klassen seien außerdem semantisch nicht interessant, so ihre Einschätzung, da jeder nichttriviale Nachbarschaftsbegriff voraussetzt, dass die Funktionenklasse nicht gegen Permutationen abgeschlossen ist, Nachbarschaftsbegriffe aber bei fast allen typischerweise betrachteten Funktionen implizit vorkommen.

3.2 Selektionsoperatoren und multikriterielle Optimierung

Vortragender: Patrick Briest

Literatur: Der Vortrag basiert auf [2], [24] und [44].

Der Vortrag gliedert sich in zwei mehr oder weniger disjunkte Teile. Im ersten Teil werden das Prinzip der Selektion und eine Reihe üblicher Selektionsoperatoren vorgestellt. Der zweite Teil befasst sich mit dem Problem der multikriteriellen Optimierung. Hier liegt der Schwerpunkt beim Prinzip der Pareto-Konzepte und den Möglichkeiten der Anwendung von evolutionären Algorithmen.

3.2.1 Selektionsoperatoren

Auf dem Weg von einer Population zu ihrer Nachfolgepopulation werden Selektionsoperatoren in der Regel an zwei Stellen eingesetzt. Zunächst wird entschieden, welche Individuen Nachkommen produzieren dürfen und wie die Nachkommen auf die Gesamtpopulation verteilt werden. In einem zweiten Schritt wird entschieden, welche (durch Mutation und Rekombination veränderten) Nachkommen tatsächlich in die neue Population aufgenommen werden. Selektionsoperatoren bestimmen also wesentlich das Verhalten eines Algorithmus, da sie darüber entscheiden, wie heterogen die Population zu bestimmten Zeiten des Laufes zusammengesetzt sein darf.

Im Grunde bedeutet Selektion das Festlegen einer Wahrscheinlichkeitsverteilung auf der Population. Jedem Individuum wird eine Target-Sampling-Rate zugeordnet, die die erwartete Anzahl von Nachkommen in der nächsten Generation beschreibt. Selektion erfolgt i. A. als zuvor festgelegtes Zufallsexperiment, durch das diese Wahrscheinlichkeitsverteilung implizit festgelegt wird. Ein anderes Vorgehen beschreibt das Prinzip des Stochastic Universal Sampling. Hier wird jedem Individuum (bis auf Rundung) exakt die erwartete Anzahl von Nachkommen zugewiesen, was die Selektion im Wesentlichen zu einem deterministischen Prozess macht.

Eine entscheidende Frage bei der Bestimmung einer Nachfolgepopulation ist, ob auch Individuen aus der vorherigen Generation wieder aufgenommen werden können oder nicht. Man unterscheidet hier zunächst zwischen (μ, λ)- und $(\mu + \lambda)$-Algorithmen, die sich in der Frage unterscheiden, ob die Elterngeneration bei der Selektion zur Aufnahme in die nachfolgende Generation beteiligt wird. Einen Kompromiss zwischen beiden Verfahren stellt die Parental Gap Method dar, die das Alter der Individuen berücksichtigt und nur hinreichend jungen Individuen eine Aufnahme in die nächste Generation ermöglicht.

Zwei wichtige Kenngrößen von Selektionsoperatoren sind Takeover-Time und Selektionsintensität. Die Takeover-Time beschreibt die Anzahl von Generationen, die nötig ist, bis ein anfänglich einzelnes Individuum mit überlegener Fitness die gesamte Population einnimmt. Die Selektionsintensität ist definiert als $I = \frac{\Phi(P')-\Phi(P)}{\sigma(P)}$, wobei $\Phi(P)$ und $\Phi(P')$ die durchschnittliche Fitness in der Population P und der Nachfolgepopulation P' und $\sigma(P)$ die Varianz der Fitnesswerte in P beschreiben. Es wird also die Verbesserung der durchschnittlichen Fitness unter Berücksichtigung ihrer Varianz gemessen.

Im Folgenden werden nun einige übliche Selektionsoperatoren vorgestellt.

Bei der proportionalen Selektion (Roulette-Wheel-Sampling) wird jedem Individuum eine Selektionswahrscheinlichkeit proportional zu seiner Fitness zugeordnet. Dann wird wiederholt entsprechend dieser Wahrscheinlichkeitsverteilung ein Individuum zufällig gewählt und erhält einen Nachkommen in der Nachfolgepopulation.

Bei der Turnierselektion werden aus der Population zunächst zufällig q Individuen ausgewählt. Unter diesen wird dann ein Turniersieger bestimmt, der einen Nachkommen in der Nachfolgepopulation erhält. Dieser zweite Schritt erfolgt in der Regel deterministisch, d. h. es wird das Individuum mit der größten Fitness gewählt. Ist dieses nicht eindeutig bestimmt, so kann natürlich wieder eine zufällige Auswahl erfolgen.

Bei der rangbasierten Selektion wird anhand der Fitnesswerte zunächst für jedes Individuum dessen Rang innerhalb der Population bestimmt. Die Selektionswahrscheinlichkeit der Individuen hängt nun nur noch von deren Rang und nicht mehr von den konkreten Fitnesswerten ab. Verhält sich die Selektionswahrscheinlichkeit proportional zum Rang eines Individuums, sprechen wir von Linear Ranking, ansonsten von Nonlinear Ranking. Einen Spezialfall bildet die so genannte Threshold Selection. Eine feste Anzahl von besten Individuen wird hierbei mit gleicher Wahrscheinlichkeit selektiert, während der Rest der Population von vornherein von der Reproduktion ausgeschlossen wird.

Die Boltzmann Selektion ist eine adaptive Selektionsstrategie. Sie orientiert sich am Prinzip des Simulated Annealing. Das bedeutet, dass zu Beginn eines Laufes Individuen fast gleichverteilt zur Reproduktion ausgewählt werden, was sehr heterogene Populationen zulässt. Mit fortschreitender Zeit werden bessere Individuen dann immer stärker bevorteilt, was die Population dann sehr schnell auf die besten bisher gefundenen Individuen konzentriert.

Alle vorgestellten Selektionsstrategien sind darauf ausgelegt, die Population in Richtung des globalen Optimums der Zielfunktion zu lenken, so lange die Fitnesswerte der Individuen unmittelbar von den Zielfunktionswerten abhängen. Wenn Teile der Population sich auch auf lokale Optima konzentrieren sollen, so müssen hierzu die Fitnesswerte transformiert werden. Niching bedeutet, dass die Fitnesswerte von Individuen reduziert werden, wenn viele Individuen auf sehr engem Raum zusammen liegen. Dies verschafft Individuen, die vom Rest der Population entfernt liegen, einen Vorteil bei der Selektion und sorgt dafür, dass die Population sich auf mehrere lokale Optima verteilen kann, wobei die Anzahl der Individuen an den verschiedenen Punkten im Erwartungswert zusammen mit der absoluten Fitness dieser Punkte steigt.

3.2.2 Multikriterielle Optimierung

Ein multikriterielles Optimierungsproblem (MOP) besteht aus einer Zielfunktion $F : \Omega \to R^m$ sowie Einschränkungen $g_i(x) \leq 0, i = 1, \ldots, k$. Ziel ist es, die Funktion F zu maximieren (bzw. minimieren).

Ein naheliegender Ansatz zur Lösung eines MOP besteht darin, es in ein eindimensionales Optimierungsproblem zu transformieren, indem z. B. die Funktion $f(x) := ||F(x)||_2$ optimiert wird. Im Allgemeinen kann es aber beliebig viele Funktionswerte von F mit identischer Norm geben, ohne dass a priori entscheidbar ist, welcher der Werte zu bevorzugen wäre. Abgesehen davon können Werte existieren, die bzgl. f nicht optimal sind, aber F dennoch in einigen Komponenten optimieren.

Dies motiviert die Einführung der sogenannten Pareto-Konzepte. Wir sagen, ein Funktionswert $(u_1, \ldots, u_m)$ dominiert einen Funktionswert $(v_1, \ldots, v_m)$ genau dann, wenn gilt:

$\forall i \in \{1, \ldots, m\} : u_i \geq v_i$ und $\exists i \in \{1, \ldots, m\} : u_i > v_i$. Ein Funktionswert heißt pareto-optimal, wenn er von keinem anderen Funktionswert dominiert wird. Das Pareto-Optimal-Set P^* ist definiert als die Menge aller $x \in \Omega$, für die $F(x)$ pareto-optimal ist. Die Menge der Funktionswerte aller $x \in P^*$ wird als Pareto-Front PF^* bezeichnet.

Offenbar ist das Ziel der multikriteriellen Optimierung nun, die Pareto-Front zu bestimmen. Da diese (abhängig von der zu optimierenden Funktion) durchaus unendlich viele Vektoren enthalten kann, wird eine möglichst gute Approximation gesucht. Wenn evolutionäre Algorithmen zur Lösung eines MOP eingesetzt werden sollen, kann wie folgt vorgegangen werden.

Zunächst ergänzen wir den Algorithmus um eine zweite Population P^*. In P^* werden alle gefundenen Individuen gespeichert, deren Funktionswerte bisher von den Funktionswerten keines anderen Individuums dominiert wurden. Die Funktionswerte der Individuen in P^* stellen also die Approximation an die Pareto-Front dar. Es wird erwartet, dass diese Approximation sich im Laufe der Zeit immer weiter der tatsächlichen Pareto-Front annähert.

Ein gängiges Selektionsverfahren für multikriterielle evolutionäre Algorithmen ist das Pareto-Ranking, das der rangbasierten Selektion (siehe Abschnitt 3.2.1) ähnelt. Man definiert hier den Rang eines Individuums als die Anzahl der Individuen, von deren Funktionswerten der Funktionswert des betrachteten Individuums dominiert wird. Lässt man jetzt nur noch Individuen mit Rang 0 zur Selektion zu, entspricht dies der Threshold Selection.

Wünschenswert ist letztendlich noch ein Mechanismus, der dafür sorgt, dass die berechnete Lösungsmenge eine möglichst äquidistante Approximation an die Pareto-Front darstellt. Zu diesem Zweck eignet sich das bereits erwähnte Prinzip des Niching, wobei das Prinzip nun im Raum der Funktionswerte angewandt wird. Das führt dazu, dass die Funktionswerte dazu neigen, sich entlang der Pareto-Front zu verteilen, da eng zusammenliegende Punkte zur Reduzierung der Fitness führen. Das Verfahren ist allerdings sehr rechenzeitintensiv, da die Abstände zwischen allen Vektoren berechnet werden. Abgesehen davon ist eine gewisse Kenntnis der Struktur der Pareto-Front nötig, um entscheiden zu können, wie nah die einzelnen Vektoren sinnvollerweise beieinander liegen sollen.

Abhilfe schafft die Idee der Adaptive Grid Algorithmen. Über den gesamten Bereich, in dem pareto-optimale Funktionswerte liegen, wird hierbei ein Gitter von gleich großen mehrdimensionalen Würfeln gelegt. Die Aufteilung wird feiner, je näher die Funktionswerte beieinander liegen. Für jedes Individuum in P^* wird bestimmt, in welchem Würfel der zugehörige Funktionswert liegt. Die Fitnesswerte von Individuen werden nun verringert, wenn mehrere Funktionswerte innerhalb desselben Würfels liegen.

3.3 Mutationsoperatoren und Diversitätserhaltung

Vortragender: Michael Leifhelm

Literatur: Handbook of Evolutionary Computation[2], Kapitel C3 und C6.

3.3.1 Mutationsoperatoren

Mutationsoperatoren sind unäre Variationsoperatoren, die aus einem Individuum neue Individuen erzeugen, unabhängig von den anderen Individuen der Population. Die neu erzeugten Individuen gelten als Nachkommen, das ursprüngliche Individuum als Elter der aus ihm erzeugten Individuen. Für die lokale Suche wird es gewünscht, dass jeder Nachfahre im Durchschnitt nur sehr kleine Unterschiede zu seinem Elter aufweist und somit im Suchraum in der Nähe seines Elters liegt. Trotzdem aber müssen auch große Veränderungen mit kleiner Wahrscheinlichkeit möglich sein, damit der Nachkomme ein lokales Maximum verlassen kann.

Im Folgenden werden nun einige Mutationsoperatoren auf verschiedenen Suchräumen vorgestellt:

Bit Strings Die am häufigsten verwendete Mutation für Bitstrings, die Standardbitmutation, flippt jedes Bit mit Wahrscheinlichkeit p_m. Der Parameter p_m macht nur im Intervall [0; 0,5] Sinn, wobei der Operator mit $p_m = 0{,}5$ komplett zufällige Individuen erzeugt. Meistens wird p_m aber auf den Wert $\frac{1}{n}$ (n ist Dimension des Suchraums) gesetzt, da dann im Durchschnitt genau ein Bit flippt.

Ein weiterer Operator ist k-Bit-Mutation. Wie es der Name schon sagt, werden genau k verschiedene Bits zufällig ausgewählt und geflippt. Allerdings kann ein Individuum nie aus einem lokalen Maximum heraus springen, wenn alle Punkte mit Hammingabstand kleiner gleich k schlechtere Fitnesswerte aufweisen und der Nachkomme direkt mit seinem Elter konkurriert.

Permutationen Mutationsoperatoren für den Suchraum der Permutationen sind etwas problematisch, da die Semantik von Fitnessfunktion zu Fitnessfunktion stark variieren kann. Beim TSP stellt die Permutation eine Rundreise in Form eines Zyklus dar, beim Sortierproblem allerdings ist die Permutation eine Zuweisungsregel aus der Menge S_n. Beim Zyklus kann jede Rundreise durch n verschiedene Permutationen (z. B. {123, 231, 312}) dargestellt werden, die in S_n n verschiedene Bedeutungen haben.

Ein Operator für Zyklen ist k-Opt, der die Permutation in k Teilstücke zerschneidet und sie in zufälliger Reihenfolge wieder zusammensetzt, wobei die Teilstücke auch verdreht werden können.

Allgemeine Operatoren für Permutationen sind Jump und Exchange. Bei Jump wird ein zufällig gewähltes Element an eine andere Stelle gebracht, alle Elemente zwischen den beiden Positionen werden passend verschoben. Beim anderen Operator Exchange findet ein einfacher Austausch von zwei Elementen statt. Damit durch diese Operatoren nicht nur lokale Suche möglich ist, werden die Mutationsschritte in vielen Implementierungen poissonverteilt häufig wiederholt.

$\mathbb{R}^n$ Für reellwertige Vektoren ist die einfache Addition $x' = x + M$ (Elter x, Nachkomme x') eines Vektors $M \in \mathbb{R}^n$ für Operatoren sehr geeignet. Dabei kann M viele Formen haben, z. B. als eine Zufallsvariable $U(a, b)^n$ oder noch einfacher als $U(-b, b)^n$. Es sei dabei a die Untergrenze und b die Obergrenze der Zufallswerte. Ein Problem ergibt sich hier, da das Individuum nicht aus einem lokalen Maximum heraus kommt, das breiter

ist als die Schrittweite b. Als Lösungsausweg hilft bei der Berechnung der einzelnen Elemente des Additionsvektors die unbegrenzte Gaußverteilung. Damit es im Durchschnitt keinen Unterschied zwischen Elter und Nachkomme gibt, wird der Parameter μ für den Erwartungswert gleich 0 gesetzt. Die Varianz σ^2 wird häufig im Verlauf immer kleiner gewählt.

Syntaxbäume Mutation wird bei Syntaxbäumen sehr selten eingesetzt, statt dessen wird meistens die Populationsgröße einfach stark erhöht. Trotzdem vertreten einige Wissenschaftler die Meinung, dass Mutation auch bei Syntaxbäumen wichtig ist. Die vier Mutationsoperatoren *grow*, *shrink*, *switch* und *cycle* werden im Folgenden kurz beschrieben.

grow: Es wird zufällig ein Blatt ausgewählt und durch einen zufällig erzeugten Unterbaum ersetzt.
shrink: Ein innerer Knoten wird zufällig ausgewählt und durch ein zufällig generiertes Blatt ersetzt.
switch: Ein innerer Knoten wird zufällig ausgewählt und die Positionen zweier seiner Unterbäume werden vertauscht.
cycle: Ein zufälliger Knoten wird ersetzt durch einen zufälligen Knoten des gleichen Typs (Blatt/innerer Knoten). Der Typ des Rückgabewertes muss der Gleiche bleiben.

3.3.2 Diversitätserhaltung

Diversität steht u. a. für die Vielfalt in ökologischen Systemen, d. h. für die Artenvielfalt der beteiligten Lebensgemeinschaft. Räumliche oder strukturelle Diversität bedeutet die Vielfalt der ökologischen Nischen.

Traditionelle evolutionäre Algorithmen führen Optimierungen aus, ohne auf die Struktur des Problems einzugehen. Auf multimodalen Funktionen eignen sich Nischenmethoden (Niching) gut, da sie Diversität innerhalb der Population erhalten und somit die Wahrscheinlichkeit, das globale Optimum zu finden, meistens erhöhen.

Fitness Sharing

Fitness Sharing senkt die Fitnesswerte von Individuen, von denen sich ähnliche Individuen in der Population befinden. Es werden also Individuen bestraft, die sich in der gleichen Nische befinden. Die Nische wird definiert über eine Distanzfunktion d, z. B. den Hammingabstand und den Parameter σ_{share}. Alle Individuen, die sich innerhalb des Radius σ_{share} eines Individuums befinden, erhöhen je nach Entfernung dessen Nischenzähler, durch den der echte Fitnesswert dann geteilt wird. Auf einige von vielen ähnlichen Individuen kann eher verzichtet werden, als auf Individuen, die alleine einen Teil des Suchraums erforschen.

Nachteile von Fitness Sharing bestehen hauptsächlich in der zusätzlichen, quadratisch zur Populationsgröße benötigten Rechenzeit. Außerdem versammeln sich Individuen unter Einsatz von Fitness Sharing nicht ganz oben auf den lokalen Maxima, es muss am Ende noch Hill Climbing durchgeführt werden.

Crowding

Bei Crowding-Verfahren werden Individuen nur durch ähnliche Individuen ersetzt. Die Diversität nimmt so beim Crowding nur langsam ab und die Konvergenz zum Optimum wird verzögert.

Beim deterministischen Crowding findet paarweises Crossover statt, bei dem je zwei Nachkommen erzeugt und direkt noch mutiert werden. Jedes der beiden Elternteile wird nur dann in der Population durch einen der Nachkommen ersetzt, wenn dieser dem Elter am ähnlichsten ist und keinen schlechteren Fitnesswert aufweist.

Artenbildung

In der Natur ist die Paarung nur zwischen Individuen der gleichen Art möglich, Kreuzungen zwischen verschiedenen Arten führen zu nicht lebensfähigen Individuen. Da mit Nischenmethoden wie Fitness Sharing mehrere lokale Maxima in Form von Nischen gleichzeitig besetzt werden, kommt es bei Kreuzungen zwischen Individuen verschiedener Nischen oft zu sehr schlechten Nachfahren, die unnötige Ressourcen- und Zeitverschwendung bedeuten. Nachfahren von Individuen aus derselben Nische befinden sich allerdings auch in dieser und helfen, die Nische besser zu untersuchen. Die Beschränkung der Rekombination auf Individuen derselben Nische bzw. gleichen Art ist eine Erweiterung der Nischenmethoden.

Eine Möglichkeit, eine Artbildung zu erzwingen, ist es, nur die Rekombination von Individuen mit einem Maximalabstand σ_{mating} zu erlauben. Falls kein anderes Individuum im Umkreis von σ_{mating} gefunden wird, wird ein Individuum rein zufällig ausgewählt. Ein nahe liegender Wert für σ_{mating} ist der Parameter σ_{share} aus der Methode Fitness Sharing.

Inselmodell

Beim Inselmodell gibt es mehrere Subpopulationen (Inseln), die auf dem gleichen Suchraum unabhängig voneinander nach dem Optimum suchen. Die Evolution findet also separat statt. Nach einer festen Anzahl von Generationen, Epoche genannt, wird allerdings ein Austausch von Individuen zwischen Inseln durchgeführt. Die Inseln sind in einer Topologie (Ring, komplett vernetzt, ...) angeordnet, nur mit direkten Nachbarn in dieser Topologie darf der Individuenaustausch stattfinden.

Diffusionsmodell

Im Diffusionsmodell werden alle Individuen in einer Topologie (Gitter, Torus, ...) angeordnet und haben somit nur eine sehr begrenzte Anzahl von Nachbarn. Nur innerhalb der Nachbarschaft darf Rekombination stattfinden. Es gibt viele Arten für die Form der zur Rekombination zugelassenen Nachbarn.

Da die Rekombination sehr lokal begrenzt ist, findet nur eine langsame Diffusion der Gene durch die Struktur statt. Je größer allerdings die Nachbarschaft ist [7], desto schneller verbreiten sich auch die Gene.

3.4 Rekombination und Schema Theorie

Vortragender: Oliver Heering

Literatur: Teile des Vortrages beruhen auf [8], [22] und [23].

3.4.1 Einleitung

Evolutionäre Algorithmen machen sich Prinzipien aus der Natur zunutze, indem sie bestimmte, zufällig ausgewählte Individuen einer Population eines Suchraumes mutieren und miteinander rekombinieren. Der Vortrag *Rekombination und Schema Theorie* beschreibt im Wesentlichen das Verfahren der Rekombination zweier oder mehrerer Individuen nach unterschiedlichen Methoden in verschiedenen Suchräumen. Des Weiteren behandelt er die Schema Theorie, die auf Holland beruht und vermutet, dass sich eine gute Gesamtlösung durch die Rekombination von mehreren kleineren guten Teil-Lösungen erstellen lässt, die sogenannte *Building-Block-Hypothese*. Schließlich werden noch einige Beispiele aufgeführt, an denen nachgewiesen wurde, dass Rekombination tatsächlich zu verbesserter erwarteter Optimierungszeit führt.

3.4.2 Rekombination im $\{0,1\}^n$

Im Suchraum der Bitstrings $\{0,1\}^n$ unterscheidet man zwischen folgenden Rekombinationstypen, denen gemeinsam ist, dass jeweils ein Teil eines Eltern-Individuums einen entsprechenden Teil des Kindes ergibt. Demnach lassen sich Rekombinationen im $\{0,1\}^n$ stets als Binärmaske $m \in \{0,1\}^n$ darstellen. Es ist stets zu unterscheiden, welches der beiden möglichen Kinder x' oder $\overline{x}'$ gewählt wird, oder ob beide gewählt werden.

Einpunktkreuzung: Bei diesem Rekombinationstyp wird eine Trennstelle zufällig gemäß Gleichverteilung gewählt. Der linke Teil des ersten Elters und der rechte Teil des zweiten Elters ergeben dann das Kind.

Mehrpunktkreuzung: Funktioniert wie die Einpunktkreuzung, allerdings werden mehrere Trennstellen gewählt und abwechselnd Teile des ersten und zweiten Elternteils rekombiniert.

Gleichmäßige Kreuzung: Jedes Bit wird mit Wahrscheinlichkeit $\frac{1}{2}$ aus dem einen oder (mit gleicher Wahrscheinlichkeit) aus dem anderen Elternteil gewählt.

Poissonkreuzung: Die Poissonkreuzung ist identisch zur Mehrpunktkreuzung mit dem Unterschied, dass jede Bitposition mit einer Wahrscheinlichkeit p^* zur Trennstelle wird. Die Distanzen zwischen den Trennstellen sind somit poissonverteilt. Üblicherweise wird $p^* < \frac{1}{2}$ gewählt. $p^* = \frac{1}{2}$ entspricht der gleichmäßigen Kreuzung.

3.4.3 Rekombination im $\mathbb{R}^n$

Im $\mathbb{R}^n$ sind alle oben genannten Rekombinationstypen grundsätzlich ebenfalls anwendbar. Zusätzlich gibt es jedoch auch die Möglichkeit, einen Punkt aus dem von den Eltern x und y aufgespannten n-dimensionalen Rechteck zu wählen. Dabei kann man deterministisch vorgehen und das arithmetische oder geometrische Mittel errechnen, oder man wählt einen Punkt zufällig gemäß Gleichverteilung.

3.4.4 Rekombination bei Permutationen

Möchte man Individuen aus dem Raum der Permutationen rekombinieren, hat man bei oben genannten Verfahren das Problem, dass die so erzeugten Kinder nicht zwingend wieder Permutationen ergeben. Aus diesem Grund müssen andere Rekombinationsverfahren eingesetzt werden. Alle Permutationen seien hierbei in Zyklenschreibweise angegeben.

Order Crossover: Bei dieser Rekombination werden zwei Trennstellen gewählt und der mittlere Teil aus dem Elternteil x in das Kind übernommen. Die fehlenden Werte werden einem zyklischen Durchlauf des Elternteils y entnommen, wobei hinter der zweiten Trennstelle von y begonnen wird.

Partially Mapped Crossover: Auch hier werden zwei Trennstellen gewählt und der mittlere Teil aus dem Elternteil x übernommen. An den fehlenden Stellen wird versucht, möglichst viele Werte vom Elternteil y direkt zu übernehmen. Bei schon belegter Position i wird die nächste Position $\pi_x(i)$ getestet, bis eine freie Stelle gefunden wird.

Order Crossover 2: Hier werden k Positionen zufällig gewählt und die nicht gewählten aus Elternteil y in das Kind übernommen. Die restlichen Positionen werden aus x in der Reihenfolge übernommen, in der sie in x vorkommen.

Maximal Preservative Crossover: Die Permutationen werden als Rundreisen betrachtet. Zunächst wird eine Teiltour aus y übernommen. Diese wird dann durch Kanten ergänzt zu einer gesamten Tour. Dabei haben Kanten aus x Vorrang.

3.4.5 Die Schema Theorie

In der Informatik werden oft umfangreiche Probleme gelöst, indem sie in kleinere Probleme aufgeteilt, separat gelöst und anschließend die Lösungen zusammengefügt werden (beispielsweise bei *branch-and-bound* und *divide-and-conquer* Verfahren oder bei der dynamischen Programmierung). Daher drängt sich die Frage auf, ob diese Vorgehensweise auch auf evolutionäre Algorithmen übertragen werden kann. Die auf Holland zurückgehende Schema Theorie untersucht diese Fragestellung. Hierbei werden folgende Bezeichnungen benutzt. $S = S_1 \times \cdots \times S_n$ sei der betrachtete Suchraum.

Gen: Teil x_i eines Individuums $x = (x_1, \ldots, x_n)$.

Definierende Positionen: festgelegte Positionen eines Individuums.

Schema: Menge der möglichen verschiedenen Individuen bei beliebiger Belegung der freien Bitpositionen.

Ordnung des Schemas: Anzahl der definierenden Positionen.

Definierende Länge eines Schemas: Abstand zwischen erster und letzter definierenden Position.

Probleme bei dieser Betrachtungsweise sind unter anderem die Frage nach dem Sinn der Aufteilung in Schemata, die gegenseitige Beeinflussung einzelner Bitpositionen und die implizite Voraussetzung an die Fitnessfunktion, dass diese separabel ist.

Das Schema Theorem von Holland beschreibt als lokales Maß für klassische genetische Algorithmen die Entwicklung eines Schemas in einem Schritt. Es liefert dabei eine untere Schranke für $\frac{E(N_{P^*}(\xi))}{N_P(\xi)}$. Dabei ist ξ ein Schema, P, P^* die aktuelle bzw. nachfolgende Population, $N_P(\xi)$ die Anzahl der Individuen in P, die zu ξ gehören, und $E(N_{P^*}(\xi))$ die erwartete Anzahl Individuen in der Nachfolgepopulation P^*, die zu ξ gehören. Die von Holland aufgestellte Schranke lautet:

$$\frac{E(N_{P^*}(\xi))}{N_P(\xi)} \geq \frac{\overline{f_P}(\xi)}{\overline{f}(P)} \cdot (1 - D_c(\xi)) \cdot (1 - D_m(\xi))$$

Dabei ist $\overline{f}(P)$ die durchschnittliche Fitness aller Individuen aus P, $\overline{f_P}(\xi)$ die durchschnittliche Fitness aller Individuen aus $P \cap \xi$, $D_c(\xi)$ die Wahrscheinlichkeit, dass die Rekombination von $x \in \xi$ mit einem beliebigen Suchpunkt ein Kind $x' \notin \xi$ erzeugt und $D_m(\xi)$ die Wahrscheinlichkeit, dass die Mutation von $x \in \xi$ ein Kind $x' \notin \xi$ erzeugt.

Diese Schranke lässt sich unter gewissen Umständen (genaue Kenntnis der gewählten Mutations- und Rekombinationsoperatoren) noch präzisieren, jedoch gibt es berechtigte Kritik an der Schema Theorie. Zum einen sagt ein lokales Maß wie das Schema Theorem nichts über das Verhalten des Algorithmus in den folgenden Schritten aus, sondern nur im nächsten Schritt. Ein gutes Schema kann sich durchaus im nächsten Schritt vergrößern, aber im darauf folgenden Schritt bereits wieder ausgelöscht sein. Zum anderen sind die Schemata oft nur künstlich herbeigeführt. Der tatsächliche Nutzen der Schema Theorie ist also fraglich.

3.4.6 Nutzen von Rekombination

Im letzten Teil des Vortrags wurde der Nutzen von Rekombination für einige beispielhafte Fitnessfunktionen verdeutlicht. Sie kann bei diesen Funktionen nachweislich zu geringerer erwarteter Optimierungszeit führen.

Die Funktion H-IFF

Die Funktion H-IFF ist auf die *Building-Block-Hypothese* zugeschnitten. Sie erhält einen Bitstring der Länge 2^k als Eingabe und berechnet ihren Fitnesswert aus der Anzahl und Länge

von zusammenhängenden 0- oder 1-Blöcken an bestimmten Positionen. Dabei stellt man sich die Funktionsweise am besten als einen binären Baum mit $n = 2^k$ Blättern mit Färbung 0 oder 1 vor. Ein innerer Knoten wird gefärbt, wenn seine Kinder ebenfalls gefärbt sind und zwar in der selben Farbe. Der Funktionswert berechnet sich dann aus der Anzahl und Höhe der gleichgefärbten Teilbäume. Die Optima dieser Funktion sind offensichtlich 0^n und 1^n. Mutationsbasierte Algorithmen haben kaum eine Chance, die Anzahl oder die Länge dieser Blocks zu erhöhen, daher wird anhand eines rekombinativen *hill-climbers* gezeigt, dass rein rekombinationsbasierte Algorithmen effizient sein können. Dazu wird Einpunktkreuzung benutzt, um ein Individuum mit seinem Komplement zu kreuzen. So werden alle Bits auf der rechten Seite der Trennstelle mitgeflippt, was die Wahrscheinlichkeit erhöht, zusammenhängende 0- und 1-Blöcke zu verlängern. Bei der Analyse dieses Algorithmus unterscheidet man zwischen zwei Varianten, einer mit sogenannten *Trittbrettfahrern*, und einer ohne Trittbrettfahrer [8].

Folgende untere Schranken wurden gezeigt:

$$\begin{array}{ll} \text{Variante 1:} & n \ln n + \frac{1}{2} n \log n - O(n) \approx 1{,}193 n \log n \\ \text{Variante 2:} & n \ln n + n \log n - O(n) \approx 1{,}693 n \log n \end{array}$$

Für beide Varianten gilt die obere Schranke:

$$O(n \log n) + o(n)$$

Gegenüber einer erwarteten exponentiellen Laufzeit eines mutationsbasierten evolutionären Algorithmus lässt sich somit eine deutliche Verbesserung feststellen.

Real Royal Road Funktionen

Nach einem vergeblichen Versuch, mit den Royal Road Funktionen einen „Königsweg", also ein Beispiel mit besonderem Demonstrationscharakter für rekombinative Algorithmen zu finden, wurden die sog. *Real Royal Road* Funktionen entwickelt. Rein mutationsbasierte Algorithmen haben auch hier eine erwartete exponentielle Laufzeit. Ein *Steady State GA* angewandt auf solch eine Real Royal Road Funktion hingegen ist effizient [22].

Real Royal Road Funktion für Einpunktkreuzung

Wird beim Steady State GA Einpunktkreuzung benutzt, so zeigt folgende Fitnessfunktion, dass Rekombination zu effizienter Laufzeit führen kann:

$$\underset{m \le \lceil \frac{n}{3} \rceil}{R_{n,m}(x)} := \begin{cases} 2n^2 & \text{falls } x = 1^n \\ n \cdot |x| + b(x) & \text{falls } |x| \le n - m \\ 0 & \text{sonst} \end{cases}$$

Hierbei ist $b(x)$ die Länge des längsten zusammenhängenden Blocks aus Einsen. Für die Analyse des Algorithmus wird ein Lauf in sechs Phasen unterteilt, in denen die Individuen der

Population an den Rand der Senke (Individuen mit Funktionswert 0) geführt, alle möglichen Belegungen von 1-Blöcken der Länge $n-m$ abgedeckt und schließlich mit einem Individuum die Senke überwunden wird. Dieser Suchpunkt ist dann optimal. Der Kreuzungsoperator kommt im letzten Schritt zum Zuge und ermöglicht das Überwinden der Senke.

Insgesamt ergeben die sechs Phasen eine Laufzeit von $O(n \cdot s(n)^2 \cdot \log s(n) + n^2 \cdot s(n) \cdot m)$.

Real Royal Road Funktion für gleichmäßige Kreuzung

Wird gleichmäßige Kreuzung als Rekombination eingesetzt, empfiehlt sich eine andere Funktion als die oben beschriebene Real Royal Road Funktion. Um zu verstehen, wie diese funktioniert, sind allerdings einige Definitionen notwendig.

Individuen x werden wie folgt aufgeteilt ($m = \frac{n}{2}; k = \frac{m}{3}; k$ gerade):

$$x: \quad \overbrace{110101011010100110}^{\substack{m \text{ Bits} \\ x'}} \overbrace{\underbrace{011010}_{\substack{k \text{ Bits} \\ x''_1}} \underbrace{101001}_{\substack{k \text{ Bits} \\ x''_2}} \underbrace{100101}_{\substack{k \text{ Bits} \\ x''_3}}}^{\substack{m\text{Bits} \\ x''}}$$

Optimale Punkte sind hierbei nicht mehr nur der einzige Punkt 1^n, sondern alle Punkte in der Menge $T := \{x = (x', x'') \mid |x''_1| = |x''_2| = |x''_3| = \frac{k}{2}\}$. Des Weiteren existiert ein Zwischengebiet $C := \{x'' \mid x'' = 0^i 1^{m-i} \quad \vee \quad x'' = 1^i 0^{m-i}\}$ (Kreis der Länge $2m = n$), in das die Individuen von der Funktion zunächst gelenkt werden. Die Real Royal Road Funktion sieht somit wie folgt aus:

$$R_n^*(x', x'') := \begin{cases} n - H(x'', C) & \text{falls } x' \neq 0^m \text{ und } x'' \notin C \\ 2n - H(x', 0^m) & \text{falls } x'' \in C \\ 0 & \text{falls } x' = 0^m \text{ und } x'' \notin C \cup T \\ 3n & \text{falls } x' = 0^m \text{ und } x'' \in T \end{cases}$$

Hierbei ist $H(x'', C)$ der minimale Hammingabstand zwischen x'' und C und $H(x', 0^m)$ der Hammingabstand zwischen x' und 0^m.

Die Analyse des GAs läuft wie bei den Real Royal Road Funktionen für Einpunktkreuzung über mehrere Phasen. Zunächst landen alle Individuen im Zwischengebiet, dem Kreis C, bis dieser ganz überdeckt wird und $x' = 0^m$ ist. Danach wird für sie das Optimum, ($x' = 0^m$ und $x'' \in T$) attraktiv. Letzten Endes ist ein Suchpunkt optimal. Addiert man die Laufzeiten der einzelnen Phasen, ergibt dies $O(n^2 \cdot s(n))$.

Die Funktion $JUMP_{n,m}$

Ein Vorläufer der Real Royal Road Funktionen ist die Funktion $JUMP_{n,m}$. Die Funktionswerte der Individuen wachsen linear mit der Anzahl der Einsen, allerdings sind die letzten $n-m$ Funktionswerte vor dem Optimum 0, so dass die Individuen eine Senke überspringen müssen, um optimal zu werden [23].

$$\text{JUMP}_{n,m} := \begin{cases} n & \text{falls } x = 1^n \\ |x| & \text{falls } |x| \leq n - m \\ 0 & \text{sonst} \end{cases}$$

Im Folgenden wird ein Steady State GA analysiert und auch hier erfolgt die Analyse, indem der Algorithmus in Phasen eingeteilt wird und diese separat analysiert werden. Mutationsbasierte Algorithmen haben Probleme beim Überwinden der Senke, daher liegt ihre erwartete Optimierungszeit bei $\Omega(n^m)$. Die Analyse des Steady State GAs mit Rekombination ergibt schließlich eine erwartete Optimierungszeit von $O(n^2 \log n)$ für m konstant und $O(n \log^3 n(n \log^2 n + 2^{2m}))$ für $m = O(\log n)$.

3.5 Parametrisierung und Parametersteuerung

Vortragender: Dimo Brockhoff

Literatur: Handbook of Evolutionary Computation [2].

Der fünfte Vortrag „Parametersteuerung und Parametrisierung“ war der letzte von fünf eher einleitenden Vorträgen in der ersten Seminarphase.

Zuerst wurden einige typische bei evolutionären Algorithmen vorkommende Parameter vorgestellt und in globale und modulare Parameter unterteilt. Anschließend wurden mit der statischen, der dynamischen und der adaptiven Parametersteuerung, sowie der Selbst-Adaptation vier Parametersteuerungsarten angesprochen und mit Beispielen erklärt. Am Ende des Vortrages wurde dann der Zusammenhang zwischen diesen vier Parametersteuerungsarten erläutert.

Bei der statischen Parametersteuerung handelt es sich um die einfachste Form der Parametersteuerungsarten. Die Parameter werden vor dem Lauf eines EA festgesetzt und bleiben während des Laufs konstant. Im Vortrag wurden dazu zwei Versuche vorgestellt, wie in den 1970ern (von DeJong) und in den 1980ern (von Grefenstette [16]) durch Experimentieren eine optimale Parametereinstellung zu finden erwartet wurde. Diese Vorstellung, optimale Parametereinstellungen für eine große Klasse von Problemen angeben zu können, erwies sich aber als unmöglich, was dazu führte, andere Möglichkeiten bei der Parametereinstellung aufzuspüren.

Eine erweiterte Parametersteuerungsart ist die dynamische Parametersteuerung, bei der die Parameter während eines EA-Laufes durch zeitabhängige Funktionen beschrieben werden können. Die Zeit darf dabei sowohl in Sekunden als auch in Generationen oder erzeugten Individuen gezählt werden. Als Beispiele hierfür wurde der dynamische (1+1) EA und Simulated Annealing vorgestellt. Wichtig ist, dass bei der dynamischen Parametersteuerung keine anderen Faktoren als die Zeit selbst Einfluss auf die Parametereinstellungen haben.

Dies entspräche nämlich der adaptiven Parametersteuerung. Hierbei kann jegliche Information, die aus dem EA-Lauf gewonnen werden kann, zur Steuerung der Parameter verwendet werden. Konkret könnte dies etwa die Fitness der erzeugten Individuen sein, aber auch etwaige Zufallsentscheidungen. Hier wurde als Beispiel die $\frac{1}{5}$-Regel von Ingo Rechenberg vorgestellt.

Diese heuristische Regel versucht, den Parameter Mutationswahrscheinlichkeit bei Evolutionsstrategien zu verändern. Gesteuert wird die Mutationswahrscheinlichkeit über den Anteil an Verbesserungen gegenüber Verschlechterungen in den letzten Generationen.

Als letzte Parametersteuerungsart wurde mit der Selbst-Adaptation eine neue Möglichkeit vorgestellt, Parameter zu steuern. Informell als „Evolution der Evolution" bezeichnet, versucht man bei der Selbst-Adaptation die Parameter ebenfalls durch die Evolution zu steuern. Jedem Individuum werden dafür Parametereinstellungen mit in die Gene codiert, die ebenfalls der Evolution (also Mutation und Rekombination) unterworfen sind. Technisch gesehen wird der Suchraum um den Raum der möglichen Parametereinstellungen ergänzt. Die Parameter der Individuen haben aber keinen direkten Einfluss auf deren Fitness. Erst die Hoffnung, dass Individuen mit guten Parameterwerten höhere Überlebenschancen haben als Individuen mit schlechten Parametern, gibt der Selbst-Adaptation einen Sinn. Die übliche Vorgehensweise ist die Folgende: Wird ein Individuum zur Reproduktion selektiert, so werden zunächst mit den dem Individuum eigenen Parametereinstellungen die Codierungen dieser Parameter der Evolution unterworfen und erst dann wird mit diesen neuen Parametereinstellungen das Individuum selbst mutiert und rekombiniert. Im Vortrag gab es insgesamt drei Beispiele für selbst-adaptive Parametersteuerung. In einem Beispiel wurde die Mutationswahrscheinlichkeit, in einem anderen die Wahl der Crossoverpunkte und im dritten Beispiel die Art des angewandten Crossoveroperators mit Hilfe von Selbst-Adaptation gesteuert.

Zum Schluss des Vortrages wurde der Zusammenhang zwischen den verschiedenen Parametersteuerungsarten durch ein Diagramm hergestellt. Bis auf die Selbst-Adaptation bilden die Steuerungsarten eine Hierarchie: Jede statische Parametersteuerung ist (im Prinzip) dynamisch und jede dynamische Parametersteuerung kann auch als eine adaptive angesehen werden.

3.6 Einführung in statistische Methoden und Wahrscheinlichkeitsrechnung

Vortragende: Christian Gunia (Abschnitte 3.6.1, 3.6.2 und 3.6.4), Andrea Schweer (Abschnitt 3.6.3)

Literatur: Für den gesamten Abschnitt das Statistiklehrbuch [32]; speziell für Abschnitt 3.6.3 das Lehrbuch [3].

3.6.1 Einführung in die Wahrscheinlichkeitsrechnung

Die Wahrscheinlichkeitsrechnung arbeitet mit Ereignissen, die eintreten oder ausbleiben können. Diese werden im Allgemeinen durch Mengen dargestellt. Die Menge aller möglichen Ereignisse ist die zu Grunde liegende Obermenge und heißt Wahrscheinlichkeitsraum Ω. Alle Ereignisse bzw. Mengen stammen aus dieser.

Offensichtlich bestehen diese Ereignisse aus „kleinsten, nicht weiter zerlegbaren Teilen", den Elementarereignissen e_i (z. B. die Ereignisse 1 bis 6 beim Würfeln, oder die reellen Zahlen, bei einer Wahrscheinlichkeitsverteilung über [0,1]). Im diskreten Fall reicht es, jedem dieser

Ereignisse eine Elementarwahrscheinlichkeit p_i zuzuweisen. Dadurch ergibt sich dann kanonisch eine Wahrscheinlichkeitsverteilung auf dem gesamten Wahrscheinlichkeitsraum Ω durch: $\text{Prob}(A) = \sum_{i:e_i \in A} p_i$. Natürlich sind an solche Wahrscheinlichkeitsverteilungen Bedingungen geknüpft: $\sum_i p_i = 1$ und $\forall i \, : \, p_i \geq 0$.

Im kontinuierlichen Fall (z. B. Verteilung in den reellen Zahlen über [0,1]) reicht dies nicht aus. Daher definiert man dann eine Dichtefunktion f und eine Verteilungsfunktion F, wobei $F(x) = \int_{-\infty}^{x} f(y)\, dy$. Die Dichtefunktion übernimmt die Aufgabe der Elementarereignisse. Nun wird also die Wahrscheinlichkeit eines Ereignisses A definiert als $\text{Prob}(A) = \int_{x \in A} f(x)\, dx$ (wobei wir davon ausgehen, dass die Menge A die notwendigen Bedingungen erfüllt, um überhaupt integrierbar zu sein).

Wie erwähnt, übernimmt die Dichtefunktion die Aufgabe der Elementarereignisse. Dementsprechend übertragen sich auch die Bedingungen passend auf sie: $\int_\Omega f(x)\, dx = 1$ und $\forall x \in \Omega \, : \, f(x) \geq 0$.

Nun haben wir zwar eine Wahrscheinlichkeitsverteilung auf Ω definiert, also Wahrscheinlichkeiten für jedes mögliche Ereignis A, aber diese interessieren uns vielleicht nicht, sondern nur davon abgeleitete Größen. Nehmen wir z. B. eine Folge von Münzwürfen mit den Ausgängen Kopf (K) und Zahl (Z) an. Ein typischer Versuchsausgang lautet nun ZZZK, eine typische Fragestellung hingegen „Wie groß ist die Wahrscheinlichkeit, dass wir öfter als 4-mal eine Zahl erhalten, wenn wir 10-mal werfen?”.

Daher definieren wir Zufallsvariablen $X : \Omega \to \mathbb{R}$. Wir bilden also von unserem Ereignisraum in einen Raum ab, der uns interessiert. Dabei erhält jede Menge aus $\mathbb{R}$ als Wahrscheinlichkeit die Wahrscheinlichkeit der Ereignisse, die unter X zu dieser Menge führen.

Diese Zufallsvariablen haben einige interessante Kenngrößen. Zwei davon sind der Erwartungswert $E(X) = \mu$ und die Varianz $V(X) = \sigma^2$. Ihre Definitionen lauten

$$E(X) = \sum_x x \cdot \text{Prob}(X = x) \quad \text{und} \quad V(X) = \sum_x (x - E(X))^2 \cdot \text{Prob}(X = x)$$

im diskreten bzw.

$$E(X) = \int_x x \cdot f(x)\, dx \quad \text{und} \quad V(x) = \int_x (x - E(X))^2 f(x)\, dx$$

im kontinuierlichen Fall. Gibt man die Art der Verteilung (Gleichverteilung, Normalverteilung, Poissonverteilung, ...), den Erwartungswert und die Varianz an, so liefert das ein gutes Bild der vorliegenden Verteilung (definiert sie sogar häufig vollständig).

Das *Gesetz der großen Zahlen* stellt die Verbindung zwischen konkreten Messwerten und den zugehörigen Wahrscheinlichkeiten her. Es besagt, dass die Wahrscheinlichkeit, im Mittel von dem Erwartungswert der Zufallsvariable abzuweichen, gegen 0 konvergiert. Eine andere asymptotische Behauptung stellt der *Zentrale Grenzwertsatz* auf: Die Verteilung der (passend) skalierten Summe von beliebig identisch verteilten Zufallsvariablen X_i konvergiert gegen die Standardnormalverteilung, falls die Zufallsvariablen unabhängig sind, also

$$\sum_{i=1}^{n} \frac{X_i - E(X_i)}{\sqrt{V(X_i)}} \to N_{0,1}.$$

3.6.2 Statistik

Während man bei der Wahrscheinlichkeitsrechnung die Verteilung der Zufallsvariablen kennt und Aussagen über die zu erwartenden Ergebnisse machen möchte, ist die Situation in der Statistik genau gegenteilig. Wir haben konkrete Messwerte vorliegen und möchten nun Aussagen über die zu Grunde liegende Verteilung machen. Dabei geht man im Allgemeinen davon aus, dass die „Art der Verteilung" (also Binomial-, Normal-, Gleichverteilung, etc.) schon bekannt ist und nur noch die konkreten Parameter der Verteilung ermittelt werden müssen. Dies sind meistens der Erwartungswert und die Varianz.

Dabei gibt es prinzipiell zwei Möglichkeiten. Entweder wir geben (möglichst wahrscheinliche) Werte für die gesuchten Parameter an oder wir geben ein Intervall und eine Wahrscheinlichkeit an, dass unsere Parameter in diesem Intervall liegen. Der erste Fall heißt Punktschätzung und der zweite aus nahe liegenden Gründen Intervallschätzung.

Bei der Punktschätzung gibt es mehrere Möglichkeiten einen passenden Schätzer, also eine deterministische Funktion über den eingegebenen Messwerten, zu finden. Man kann einen beliebigen Schätzer bestimmen und dessen Erwartungswert berechnen, denn die eingegebenen Messwerte sind ja Zufallsvariablen. Die Abweichung zwischen diesem Erwartungswert und der zu messenden Größe nennt man dann Bias. Ist das Bias Null, so haben wir einen erwartungstreuen Schätzer gefunden.

Eine weitere (nahe liegende) Möglichkeit einen solchen Schätzer zu finden ist das Maximum-Likelihood-Prinzip. Bei diesem wählen wir die gesuchten Parameter so, dass die erhaltene Stichprobe die größte Wahrscheinlichkeit hat aufzutreten (unter allen zulässigen Parametereinstellungen).

Bei kontinuierlichen Zufallsvariablen hat aber eine solche Punktschätzung (außer in trivialen Fällen) die Wahrscheinlichkeit Null, den korrekten Wert zu liefern. Daher legen wir lieber ein Intervall (ein so genanntes Konfidenzintervall) für die gesuchten Parameter fest und geben dann eine Wahrscheinlichkeit an, mit der der reale Wert in diesem Intervall liegt (Intervallschätzung).

Eine offensichtliche und häufig genutzte Methode, solch ein Intervall zu finden, ist die Folgende. Man bestimmt mittels Punktschätzung Werte für die gesuchten Parameter und schlägt anschließend ein Intervall um jeden Wert. Dadurch erhalten wir einen mehrdimensionalen Würfel, in dessen Inneren die gesuchten Parameter mit einer gewissen Wahrscheinlichkeit liegen (die berechnet werden bzw. die Wahl der Intervalle an diese Wahrscheinlichkeit angepasst werden kann). Diese Wahl ist natürlich nicht eindeutig, man möchte allerdings die Größe dieses Hyperwürfels möglichst minimieren. Ein wichtiges Prinzip dazu ist die *Pivotal Quantity*, bei der man die Verteilung so modifiziert, dass sie von den gesuchten Parametern unabhängig wird und dann daraus die passenden Grenzen berechnet. Konkrete Beispiele für die Berechnung von Erwartungswert und Varianz befinden sich in dem Vortrag.

Kann man die Größe der Stichprobe (passend groß) wählen, so liefert der Zentrale Grenzwertsatz eine Möglichkeit, diese Grenzen viel einfacher zu bestimmen, da man dann die Verteilung „kennt" (*Large-Sample-Konfidenzintervalle*).

3.6.3 Hypothesentests

Während typische Fragestellungen der Wahrscheinlichkeitsrechnung ausgehend von einer bekannten Wahrscheinlichkeitsverteilung nach der Wahrscheinlichkeit suchen, dass ein bestimmtes Ereignis eintritt, wird bei Hypothesentests gerade der umgekehrte Fall betrachtet: Ein bestimmtes Ereignis ist eingetreten, und man möchte auf der Grundlage dieses Ereignisses mit möglichst großer Sicherheit Rückschlüsse auf die zugrunde liegende Wahrscheinlichkeitsverteilung ziehen können. Im Gegensatz zum Schätzen hat man allerdings schon Vermutungen über diese Verteilung und benutzt Hypothesentests, um sich für eine der Möglichkeiten entscheiden zu können.

Eine *Hypothese* ist einfach eine Aussage über die Verteilung einer oder mehrerer Zufallsvariablen. Ein *Test* einer Hypothese ist eine Vorschrift, ob die Hypothese zurückgewiesen werden soll. Das Vorgehen bei Hypothesentests ist also eher indirekt: anstatt zu testen, ob die Richtigkeit einer Hypothese angenommen werden kann, wird getestet, ob man eine Hypothese ablehnen kann, ohne dass der dabei entstehende Fehler allzu groß wird.

Oft gibt es genau zwei sich gegenseitig ausschließende Hypothesen. In diesem Fall nennt man die beiden Hypothesen die *Nullhypothese* und die *Alternativhypothese*, wobei man üblicherweise die Formulierungen so wählt, dass die Nullhypothese diejenige ist, die verworfen werden soll (im Folgenden wird immer, soweit nicht anders erwähnt, von diesem Fall ausgegangen). Die Fehlerwahrscheinlichkeiten für die sich dabei ergebenden zwei Möglichkeiten werden *Fehler 1. Art* (die Nullhypothese wird verworfen, obwohl sie wahr ist) und *Fehler 2. Art* (die Nullhypothese wird angenommen, obwohl sie falsch ist) genannt. Der Bereich, in dem die Nullhypothese abgelehnt wird, heißt auch *kritischer Bereich* und der Bereich, in dem sie angenommen wird, entsprechend *Annahmebereich*.

Finden und Bewerten von Hypothesentests

Wie findet man nun einen Hypothesentest? Eine grundsätzliche Idee ist, eine Eigenschaft der Zufallsvariablen zu finden, die sich unter den beiden Hypothesen unterschiedlich verhält. Aus diesem unterschiedlichen Verhalten lässt sich dann ein Test ableiten. Ein Beispiel für eine solche Eigenschaft bei einer normalverteilten Zufallsvariablen ist der Mittelwert.

In den meisten Fällen wird man nicht damit zufrieden sein, *irgendeinen* Test zu finden, sondern möchte verschiedene Tests bewerten und miteinander vergleichen können. Dazu werden Gütekriterien für Hypothesentests benötigt.

Eine Eigenschaft von Hypothesentests ist die *Operationscharakteristik* (OC). Für von einem Parameter abhängige Verteilungen gilt Folgendes: Die Operationscharakteristik bildet die Werte des Parameters ab auf die Wahrscheinlichkeit, die Nullhypothese zu akzeptieren. Im Idealfall nimmt die Operationscharakteristik daher im kritischen Bereich den Wert 0 und im

Annahmebereich den Wert 1 an. In der Literatur findet man oft noch den Begriff *Gütefunktion*, die analog auf die Wahrscheinlichkeit abbildet, dass die Nullhypothese angenommen wird, daher also $1 - \mathrm{OC}$ ist.

Die *Größe* eines Tests ist ein auf der Operationscharakteristik aufbauendes Gütekriterium für Hypothesentests. Sie ist definiert als 1 minus dem Infimum der Operationscharakteristik für alle zur Nullhypothese gehörenden Parameterwerte annimmt.

Im Folgenden sei nun der Fall angenommen, dass eine Stichprobe gegeben ist, für die entschieden werden soll, aus welcher von zwei bekannten Verteilungen sie stammt. Bei einer einzigen Beobachtung ist eine intuitive Entscheidungsregel, sich für diejenige Verteilung zu entscheiden, die für diese Beobachtung den größeren Wert annimmt. Diese Intuition erweist sich als brauchbar in verallgemeinerter Form.

Ein Test heißt *einfacher Likelihood-Ratio-Test*, wenn er die folgende Form hat: Betrachte das Verhältnis λ der Wahrscheinlichkeit, dass die Beobachtung aus der Verteilung der Nullhypothese stammt, zu der Wahrscheinlichkeit, dass die Beobachtung aus der Verteilung der Alternativhypothese stammt. Wähle eine Konstante. Verwirf die Nullhypothese, wenn das Verhältnis λ kleiner als die Konstante ist; akzeptiere die Nullhypothese, wenn dieses Verhältnis größer als die Konstante ist. Im Falle der Gleichheit spielt es keine Rolle, ob der Test immer verwirft, immer akzeptiert oder randomisiert entscheidet.

Optimalität von Hypothesentests

Ein einfaches Gütekriterium eines Tests ist ein niedriger Fehler 1. bzw. 2. Art. Um beide Fehler mit einzubeziehen, kann man die Summe der Fehlerwahrscheinlichkeiten minimieren oder bei fester Wahrscheinlichkeit für einen Fehler 1. Art (der üblicherweise als der Schlimmere angesehen wird) die für einen Fehler 2. Art minimieren.

Die Definition eines *Most Powerful Test* orientiert sich an der zweiten dieser Möglichkeiten: Ein Most Powerful Test einer gegebenen Größe α ist ein Test mit Größe α und einer unter allen anderen Tests mit dieser Größe maximalen Wahrscheinlichkeit, die Nullhypothese zu verwerfen, wenn sie falsch ist. Das Neyman-Pearson-Lemma gibt an, wie man einen Most Powerful Test zu einer vorgegebenen Größe konstruieren kann.

Ein anderes Gütekriterium eines Tests ist das von „geringstem Verlust“: Man definiert eine *Verlustfunktion*, die den Verlust einer Entscheidung bei gegebenem wahren Wert angibt. Üblicherweise hat eine richtige Entscheidung einen Verlust von 0, alle anderen Entscheidungen haben positiven Verlust.

Intuitiv wird man bei einem Vergleich von mehreren Tests denjenigen Test bevorzugen, der den geringsten Verlust verursacht. Das Problem dabei ist, dass es selten einen Test geben wird, der in *allen* Fällen den geringsten Verlust hat. Daher wird der „durchschnittliche Verlust“ betrachtet. Dazu definiert man die *Risikofunktion* als den erwarteten Verlust. Ein *Minimax-Test* ist dann ein Test mit minimalem größten Risiko. Auch hier lässt sich wieder eine Konstruktionsvorschrift für Tests mit dieser Eigenschaft angeben.

Alle Tests mit den bisher vorgestellten Eigenschaften (Most Powerful Test, Minimax-Test) sind Likelihood-Ratio-Tests.

Im Falle zusammengesetzter Hypothesen (d. h. die Hypothese bestimmt die Verteilung nicht vollständig) lässt sich ähnlich vorgehen, so dass eine Verallgemeinerung des Likelihood-Ratio-Prinzips zur Definition eines Uniformly Most Powerful Test führt.

3.6.4 Zusammenhang zwischen Konfidenzintervallen und Tests

Tests und Konfidenzintervalle stehen in einem engen Zusammenhang. Zum einen kann man aus einem Konfidenzintervall einen Test auf $\Theta = \Theta_0$ gegen $\Theta \neq \Theta_0$ erhalten, indem wir überprüfen, ob der Wert Θ_0 in unserem Konfidenzintervall liegt. Falls das Konfidenzintervall eine Fehlerwahrscheinlichkeit von α hatte, so erhalten wir dabei einen Test der Größe α.

Es ist aber andererseits auch möglich, aus einer Reihe von Tests der Art $\Theta = \Theta_0$ gegen $\Theta \neq \Theta_0$ ein Konfidenzintervall zu bestimmen (wobei das Θ_0 bei den Tests wählbar sein muss).

Da große Testreihen jedoch meist mit erheblichem Aufwand verbunden sind, wünscht man sich möglichst kleine, aber dennoch aussagekräftige Tests (also Tests mit kleiner Fehlerwahrscheinlichkeit). Zwischen diesen beiden Größen besteht natürlich ein Trade-Off. Aber manchmal gelangt man in die Situation, dass ein Teil der Stichprobe bereits ausreicht, um ein „richtiges" Ergebnis zu nennen. Dies ist jedoch erst bei der Auswertung der konkreten Stichprobe erkennbar, muss also während der „Laufzeit" des Tests geschehen. Dies führt zu sequenziellen Tests, die nach jeder betrachteten Stichprobe überprüfen, ob eine gewisse „Sicherheit" des Ergebnisses vorliegt und nur dann die nächste Probe betrachten, falls dies nicht erfüllt ist.

3.7 Black-Box Komplexität

Vortragender: Kai Plociennik

Literatur: Der Vortrag basiert auf der Arbeit [9].

Die Motivation für den Vortrag mit dem Thema Black-Box Komplexität war, dass im Bereich der evolutionären Algorithmen, mit denen während der Projektgruppe im Wesentlichen umgegangen werden würde, eine speziell an das Szenario dieser Algorithmen und der von ihnen behandelten Probleme angepaßte Komplexitätstheorie vonnöten ist, um Aussagen über die Performanz solcher Algorithmen angemessen beurteilen zu können.

Diese Notwendigkeit entsteht aus mehreren Gründen:

- Im Bereich evolutionärer Algorithmen wird die Rechenzeit anders beurteilt als in der klassischen algorithmischen Komplexitätstheorie. Die Rechenzeit wird über die Zahl der Funktionsauswertungen bestimmt, die ein Algorithmus zum Optimieren einer Funktion benötigt, anstelle zum Beispiel der Zahl der Rechenschritte einer Turingmaschine.

- Die Informationen, die ein solcher Algorithmus über seine Eingabe erhält, sind andere. Eingaben sind Funktionen auf einem Suchraum und die einzige Möglichkeit, die ein evolutionärer Algorithmus besitzt, um Informationen über eine solche Funktion zu

bekommen, ist die Funktion an einzelnen Punkten des Suchraumes auszuwerten. Dem gegenüber steht die kompakte Beschreibung einer Eingabe in der klassischen Komplexitätstheorie.

- Evolutionäre Algorithmen zeichnen sich häufig durch bestimmte Eigenschaften ihrer Arbeitsweise aus. Beispielsweise werden häufig in Algorithmen, die auf Bitstrings als Suchraum arbeiten, Nullen und Einsen symmetrisch behandelt und es wird keine besondere Ordnung der einzelnen Bits des Suchraums vorausgesetzt.

Alle drei Punkte müssen also bei einer fairen Bewertung der Rechenzeit evolutionärer Algorithmen berücksichtigt werden, und dies leistet die Theorie der Black-Box Komplexität.

3.7.1 Die Theorie

Die vorgestellte Theorie ist in der erwähnten Arbeit beschrieben. Das Rechenmodell welches zur Messung der Rechenzeit benutzt wird ist nichtuniform und es wird für solche Rechner und Klassen von Problemen die bestmögliche Worst-Case erwartete Laufzeit gemessen. Das bedeutet man betrachtet für einen (randomisierten) Algorithmus die schlechteste Laufzeit bezüglich aller Eingaben aus der Problemklasse und die Bildung des Erwartungswertes der Laufzeit eines Algorithmus auf einem Problem bezieht sich auf die möglichen Abläufe des Algorithmus. Über alle möglichen Algorithmen wird dann das Minimum der erzielten Laufzeiten benutzt, um die Komplexität einer Problemklasse festzulegen.

Der Vorteil dieser Theorie (das heißt der Verwendung des benutzten (nichtuniformen) Rechenmodelles und der in der Theorie benutzten Definition von Komplexität) liegt darin, dass die einzige bis heute bekannte Methode zur Bestimmung von unteren Schranken für den Ressourcenverbrauch randomisierter Algorithmen benutzt werden kann, um untere Schranken für die Black-Box-Komplexität zu zeigen. Diese Methode ist das Minimax-Prinzip von Yao [35]. In vielen Fällen ist es mit dieser Methode einfach, untere Schranken für die Komplexität eines Problems (Problemklasse) zu zeigen.

3.7.2 Vorgestellte Teile der Theorie

Es wurden die Grundlagen der Theorie zusammen mit der erwähnten Methode vorgestellt, welche an einigen Beispielen demonstriert wurde und zeigte, dass der Beweis von unteren Schranken der Komplexität in einigen Fällen tatsächlich kurz und einfach ist. Des Weiteren wurde angerissen, wie die Theorie auf Szenarien ausweitbar ist, die nicht immer vollständig mit dem eigentlichen Szenario übereinstimmen. Dies bezieht sich zum Beispiel darauf, dass die Klassen von Problemen, die behandelt werden, im Sinne der Definition endlich sein müssen. Es gibt aber die Möglichkeit, im Einzelfall auch Ergebnisse über unendlich große Problemklassen zu erhalten.

3.7.3 Ergebnisse

Unter den vorgestellten Ergebnissen waren einige, die das Verhalten von Algorithmen auf bestimmten Funktionen in ein neues Licht rücken. Beispielsweise gibt es Funktionen, die

als eine Art Beispielfunktion für schlechtes Verhalten evolutionärer Algorithmen konstruiert wurden, und es ist auch möglich, eine hohe Laufzeit von evolutionären Algorithmen auf solchen Funktionen zu beweisen. Mit den Ergebnissen der Black-Box-Komplexität erkennt man aber, dass diese hohe (exponentielle) Laufzeit nicht aufgrund des schlechten oder nicht für diese Probleme geeigneten Verhaltens der Algorithmen entsteht, sondern probleminhärent ist. Das heißt, dass diese Probleme eine exponentielle Black-Box-Komplexität besitzen und man fairerweise also gar nicht erwarten kann, dass evolutionäre Algorithmen, deren Performanz mit dieser Komplexitätstheorie relativ fair bewertet wird, sich besser verhalten.

Dem gegenüber stehen andere Ergebnisse, die das etablierte Verständnis und die Intuition über das Verhalten evolutionärer Algorithmen korrigieren. Beispielsweise ist eine oft geäußerte Meinung über Probleme (Funktionen), dass diese für evolutionäre Algorithmen nur dann schwierig sein können, wenn sie viele lokale Optima besitzen. Dies entsteht aus der Beobachtung, dass evolutionäre Algorithmen Probleme damit haben, aus lokalen Optima heraus zu gelangen, aber meistens mit Funktionen, für die es in nicht optimalen Punkten immer bessere Nachbarn gibt, keine Probleme haben.

Im Vortrag wurde erwähnt, dass man für die Klasse unimodaler Funktionen (das sind solche Funktionen, wie sie eben erwähnt wurden) eine exponentielle Black-Box-Komplexität zeigen kann, was im krassen Gegensatz zur hergebrachten Einschätzung des Verhaltens evolutionärer Algorithmen auf solchen Funktionen steht.

3.7.4 Erkenntnisse

Die Theorie der Black-Box-Komplexität zusammen mit den vorgestellten Ergebnissen rückt das Laufzeitverhalten einiger evolutionärer Algorithmen auf bestimmten Funktionen in ein neues Licht. Diese Dinge sollten helfen, ein tieferes Verständnis des Verhaltens evolutionärer Algorithmen zu entwickeln und die Laufzeit solcher Algorithmen realistischer einschätzen zu können.

3.8 Methoden zur Analyse evolutionärer Algorithmen

Vortragende: Heiko Röglin, Dirk Sudholt

Literatur: Das Referat beruhte zu großen Teilen auf [46]. Dort findet man auch weitere Literaturverweise.

3.8.1 Einleitung

In diesem Referat wurden Methoden vorgestellt, wie man das globale Verhalten evolutionärer Algorithmen analysieren kann. Von besonderem Interesse war die Zufallsvariable X_f, die angibt, wie viele Schritte der untersuchte evolutionäre Algorithmus braucht, um die Funktion f zu optimieren. Es wurden Methoden vorgestellt, um den Erwartungswert und die Verteilung von X_f zu untersuchen. Diese Methoden wurden dann auf ausgewählte einfache Funktionen angewendet, die in abstrakter Form typische Eigenschaften praktischer Probleme widerspiegeln und für die eine vollständige Analyse bereits gelungen ist.

3.8.2 Die Methode der Fitnessschichten

Die Methode der Fitnessschichten wurde vorgestellt als eine einfache, aber effektive Methode zur Berechnung oberer Schranken für die erwartete Optimierungszeit.

Der Suchraum wird partitioniert in Fitnessschichten, das sind Mengen von Suchpunkten, die sich bezüglich ihrer Fitness total ordnen lassen. Dann errechnen sich die gewünschten Schranken aus den erwarteten Zeiten, bis Fitnessschichten zu Gunsten besserer Schichten verlassen werden.

Die Methode der Fitnessschichten wurde daraufhin auf einige einfache Beispielfunktionen angewendet und es wurden Simulationen des (1+1) EAs vorgeführt.

3.8.3 Tail Inequalities

An einem Beispiel wurde aufgezeigt, dass es im Allgemeinen nicht ausreicht, nur die erwartete Optimierungszeit zu berechnen. Eine Funktion, deren erwartete Optimierungszeit exponentiell ist, kann unter Umständen durch unabhängige Multistarts in polynomieller erwarteter Zeit optimiert werden. Deswegen ist es wichtig, auch die Verteilung der Optimierungszeit zu untersuchen.

Zu diesem Zweck wurden zwei so genannte Tail Inequalities vorgestellt. Das sind Ungleichungen, mit denen man für eine Zufallsvariable abschätzen kann, wie wahrscheinlich Abweichungen vom Erwartungswert sind.

Zuerst wurde die Markoff-Ungleichung vorgestellt; diese lässt sich auf alle Zufallsvariablen anwenden, die nur nicht-negative Werte annehmen. Da die Voraussetzung für die Anwendung der Markoff-Ungleichung derart allgemein ist, kann man nicht erwarten, dass sie in speziellen Situationen gute Abschätzungen liefert.

Wir haben die Markoff-Ungleichung am Beispiel der Einsen in einem zufälligen Suchpunkt aus $\{0,1\}^n$ angewendet. Diese Anzahl ist binomialverteilt, wir haben also wesentlich mehr Informationen zur Verfügung als in der Markoff-Ungleichung ausgenutzt werden. Deswegen ist das Ergebnis auch nicht besonders überzeugend.

Um bessere Ergebnisse zu erzielen, haben wir die Chernoff-Ungleichung vorgestellt. Diese ist speziell für binomialverteilte Zufallsvariablen ausgelegt und liefert deswegen auch ein deutlich besseres Ergebnis: beim Beispiel sind Abweichungen vom Erwartungswert $\frac{n}{2}$ um einen additiven Term der Größenordnung n^ε mit $\varepsilon > \frac{1}{2}$ exponentiell unwahrscheinlich.

Als letztes Beispiel wurde die Chernoff-Ungleichung noch dazu benutzt, um zu zeigen, dass die obere Schranke $O(n^2)$ für die erwartete Optimierungzeit von LeadingOnes mit einer Wahrscheinlichkeit exponentiell nahe an 1 eingehalten wird.

3.8.4 Das Coupon Collector's Theorem

Zunächst wurde die folgende Fragestellung untersucht: Ein Sammler möchte n verschiedene Objekte (z. B. Fußball-Bilder) sammeln. Beim Kauf sieht er nicht, welches der n Objekte er

erwirbt, aber jedes ist mit gleicher Wahrscheinlichkeit in der Verpackung. Wie viele Objekte muss er dann im Erwartungswert kaufen, bis er jedes der n Objekte mindestens einmal besitzt?

Die Antwort auf diese Frage lautet: $n \cdot \ln n + O(n)$. Weiterhin wurde gezeigt, dass Abweichungen vom Erwartungswert in diesem Szenario sehr unwahrscheinlich sind.

Dieses Ergebnis kann man z. B. dafür benutzen, um eine untere Schranke für die erwartete Optimierungszeit des (1+1) EA auf Funktionen mit eindeutigem Optimum zu zeigen. Bei einer zufälligen Initialisierung sind im Erwartungswert $\frac{n}{2}$ viele Bits nicht so belegt wie im optimalen Suchpunkt. Ähnlich wie der Sammler muss nun der (1+1) EA jedes dieser falsch belegten Bits einmal „sammeln", d. h. in diesem Kontext, dass jedes falsche Bit einmal geflippt werden muss. Im Erwartungswert flippt pro Schritt ein Bit. Mit ein paar technischen Tricks kann man nun das obige Theorem anwenden und zeigen, dass der (1+1) EA im Erwartungswert $\Omega(n \log n)$ viele Schritte braucht, um das Optimum zu erreichen.

3.8.5 Typische Läufe

Eine wichtige Idee bei der Analyse evolutionärer Algorithmen ist, typische Läufe zu betrachten: Die erwartete Optimierungszeit eines typischen Laufs wird abgeschätzt und mit der Wahrscheinlichkeit verrechnet, dass ein Lauf nicht typisch ist.

Dabei lässt sich ein Lauf oft in mehrere Phasen einteilen, die unterschiedliche Ziele haben und die in gewissen Grenzen separat analysiert werden können. Der Vorteil ist, dass man bei einem fehlerfreien Lauf in einer Phase voraussetzen kann, dass das Ziel der vorigen Phase erreicht worden ist.

Diese Ideen wurde erklärt und auf ausgewählte Funktionen angewendet, um den praktischen Einsatz zu zeigen. Das Beispiel einer so genannten Pfadfunktion mit einem Pfad benachbarter Punkte in Richtung des Optimums stellte gleichzeitig eine Überleitung zum nächsten Abschnitt dar.

3.8.6 Plateaus und Gambler's Ruin

In diesem Abschnitt wurde der Frage nachgegangen, wie evolutionäre Algorithmen sich verhalten, wenn die Fitnessfunktion keinerlei Hinweise gibt, in welcher Richtung das Optimum liegt. So genannte Plateaus sind Mengen von benachbarten Suchpunkten mit gleicher Fitness.

Als Anwendungsbeispiel wurde die Pfadfunktion aus dem vorigen Abschnitt leicht modifiziert, so dass der Pfad keine ansteigende Fitness aufweist, sondern ein Plateau mit n Suchpunkten darstellt. Mit Hilfe von „Gambler's Ruin", ein Theorem aus der Wahrscheinlichkeitstheorie, und einer Simulation wurde gezeigt, dass der (1+1) EA eine Irrfahrt auf dem Plateau durchführt und bei polynomiell kleinen Plateaus die gegebene Fitnessfunktion effizient optimiert.

3.8.7 Analysen mit Potenzialfunktionen

Die Funktion BinaryValue wurde betrachtet, diese interpretiert Bitstrings aus $\{0,1\}^n$ als Binärzahlen und ordnet ihnen den Wert als Binärzahl als Fitness zu. Die Analyse des Verhaltens des (1+1) EA auf dieser Funktion gestaltet sich zunächst als schwierig. Das Problem ist nämlich, dass der Fitnesswert kein guter Indikator dafür ist, wie nah man dem Optimum ist. Das Optimum ist der String 1^n, der String 01^{n-1} hat aber eine kleinere Fitness als der String 10^{n-1}, obwohl der Hamming-Abstand des letzteren Strings vom Optimum deutlich größer ist.

Lösung des Problems ist es, eine Potenzialfunktion einzuführen. Diese Funktion ordnet den Individuen Potenzialwerte zu, die den Abstand zum Optimum besser beschreiben als die eigentliche Fitnessfunktion. In diesem Fall bietet es sich an, die Anzahl der Einsen als Potenzialfunktion zu nehmen. Aus technischen Gründen unterteilt man die Optimierung in zwei Phasen. Die erste Phase ist beendet, wenn die erste Hälfte des Strings nur noch aus Einsen besteht. Deswegen nimmt man in der ersten Phase als Potenzial die Anzahl der Einsen in der ersten Hälfte des Strings und in der zweiten Phase die Anzahl der Einsen in der zweiten Hälfte.

Das Problem ist nun aber, dass sich der (1+1) EA nicht an der Potenzialfunktion, sondern an der eigentlichen Fitnessfunktion orientiert, d. h. insbesondere, dass das Potenzial in einzelnen Schritten auch wieder sinken kann. Es wurde jedoch gezeigt, dass man im Erwartungswert nur konstant viele erfolgreiche Schritte braucht, um das Potenzial um 1 zu erhöhen. Erfolgreiche Schritte sind im Wesentlichen diejenigen, bei denen ein neuer Suchpunkt erzeugt wird, den der (1+1) EA akzeptiert.

Als Ergebnis erhalten wir, dass BinaryValue vom (1+1) EA in erwarteter Zeit $O(n \log n)$ optimiert wird. Diese Analyse lässt sich auch auf alle linearen Funktionen erweitern.

3.8.8 Analysen mit Rekombination

Zu guter Letzt wurde die Analyse evolutionärer Algorithmen angesprochen, die mit Rekombination arbeiten. Hier entstehen Abhängigkeiten zwischen Individuen der Population, die die Analyse gegenüber dem (1+1) EA stark erschweren.

Als Beispiel für eine Analyse mit Rekombination wurden Funktionen ausgewählt, die speziell für den Einsatz von Rekombination ausgelegt sind und nur mit Hilfe von Rekombination effizient zu optimieren sind.
Dies war gleichzeitig ein weiteres Beispiel für eine Analyse typischer Läufe mit einer Einteilung in Phasen. Der Optimierungsprozess wurde dabei durch eine Simulation veranschaulicht.

3.9 (1+λ) EA

Vortragender: Matthias Englert

Literatur: Der Vortrag basiert auf [20].

In der Analyse des (1+λ) EA [20] wird das asymptotische Laufzeitverhalten des (1+λ) EA in Abhängigkeit von λ und der Suchraumdimension untersucht. Hierzu wird die Laufzeit des (1+λ) EA mit der des (1+1) EA auf drei ausgewählten Funktionen verglichen.

Es zeigt sich, dass der (1+λ) EA dem (1+1) EA auf den einfachen Funktionen LeadingOnes und OneMax für kein λ überlegen ist. Wählt man λ zu groß, optimiert der (1+λ) EA die Funktionen sogar langsamer als der (1+1) EA. Den Punkt, bis zu dem man λ erhöhen kann, ohne einen signifikanten Laufzeitverlust gegenüber dem (1+1) EA zu erleiden, asymptotisch genau zu bestimmen, ist ein vorrangiges Ziel der durchgeführten Analyse. Ein weiteres wichtiges Ziel ist, nachzuweisen, dass es Funktionen gibt, die von dem (1+λ) EA deutlich schneller optimiert werden als von dem (1+1) EA. Die zu diesem Zweck konstruierte Funktion, die im folgenden als ErringLights bezeichnet wird, ist komplexer als LeadingOnes und OneMax und besitzt eine Vielzahl lokaler Optima.

3.9.1 Der (1+λ) EA

Der Algorithmus verläuft wie folgt:

1. Wähle $x = (x_1, \ldots, x_n) \in \{0,1\}^n$ zufällig gleichverteilt.
2. Erzeuge unabhängig λ Mutanten $y_1, \ldots, y_\lambda$ aus x, indem die Bits von x unabhängig voneinander mit Mutationswahrscheinlichkeit $1/n$ geflippt werden.
3. Ermittle den maximalen Funktionswert unter den Mutanten:
$m := \max\{f(y_1), \ldots, f(y_\lambda)\}$.
4. Falls $m \geq f(x)$ gilt, ersetze x durch ein Individuum, welches aus der Menge der y_i mit maximalem Funktionswert $\{y_i \mid f(y_i) = m\}$ zufällig gewählt wird.
5. Weiter bei Schritt 2.

Die Optimierungszeit wird in Funktionsauswertungen, nicht in Generationen gemessen. Daher sollte λ immer polynomiell beschränkt sein. Es gibt also eine Konstante k, so dass $\lambda \leq n^k$ gilt.

3.9.2 LeadingOnes

Für die erwartete Laufzeit des (1+λ) EA auf LeadingOnes gilt $E(X_{\text{LeadingOnes}}) = \Theta(n^2 + n \cdot \lambda)$. Die obere Schranke lässt sich leicht durch die Methode der fitnessbasierten Partitionen zeigen. Die untere Schranke erhält man, indem man zunächst davon ausgeht, dass mit wenigstens $n/2$ Nullen gestartet wird und in n^2 Generationen kein Mutant durch das Flippen von mehr als $k + 3$ Nullen gebildet wird. Entsprechend der Methode des typischen Laufes muss man anschließend noch nachweisen, dass die Wahrscheinlichkeit, dass eine dieser Annahmen verletzt wird, nicht übermäßig groß ist. Nicht übermäßig groß bedeutet hierbei, dass die Summe der Wahrscheinlichkeiten (im Folgenden Fehlerwahrscheinlichkeiten genannt) durch eine Konstante echt kleiner eins nach oben beschränkt sein sollte. Die Fehlerwahrscheinlichkeit

für eine Initialisierung mit weniger als $n/2$ Nullen ist höchstens $1/2$. Für die Fehlerwahrscheinlichkeit, dass es unter n^2 Generationen eine Generation gibt, in der mehr als $k+3$ Nullen geflippt werden, erhält man $1/n$ als obere Schranke. Die beiden Annahmen sind daher zulässig.

Die Optimierungszeit des (1+1) EA auf LeadingOnes beträgt $\Theta(n^2)$. Der kritische Punkt für λ ist also $\lambda = \Theta(n)$. Für $\lambda = \omega(n)$ ist der (1+λ) EA deutlich schlechter als der (1+1) EA. Für kleinere λ sind beide Algorithmen asymptotisch gleichwertig.

Dies lässt sich auch anschaulich begründen. Die Wahrscheinlichkeit, durch eine Mutation eine Verbesserung zu erzielen, ist während der gesamten Laufzeit etwa $1/n$. Demnach erwarten wir unter n Mutanten etwa einen mit verbessertem Funktionswert. Es sollte also keinen deutlichen Unterschied machen, ob wir erst n Mutanten erzeugen und dann den einen Besseren suchen oder ob wir immer nur einen Mutanten erzeugen und diesen gleich überprüfen.

3.9.3 OneMax

Bei OneMax verhalten sich die Dinge ähnlich wie bei LeadingOnes. Lediglich die Analyse gestaltet sich schwieriger, da sich bei OneMax die Wahrscheinlichkeit ein Kind zu erzeugen, das besser als sein Elter ist, im Laufe der Zeit stark ändert.

Als erste obere Schranke für die Optimierungszeit des (1+λ) EA auf OneMax kann man mit Hilfe der fitnessbasierten Partitionen nachweisen, dass $E(X_{\text{OneMax}})= O(n\lambda + n \log n)$ gilt.

Die weitere obere Schranke $E(X_{\text{OneMax}}) = O(n \log n)$ wird für $\lambda = O\left(\frac{\ln n \cdot \ln \ln n}{\ln \ln \ln n}\right)$ nachgewiesen. Hierzu wird der Lauf in zwei Phasen unterteilt. In der ersten Phase gilt $\text{OneMax}(x) \leq n - \frac{n}{\ln \ln n}$ während die zweite Phase durch $\text{OneMax}(x) > n - \frac{n}{\ln \ln n}$ charakterisiert wird. Die zweite Phase ist so kurz, dass mit Standardmethoden die gewünschte obere Schranke nachgewiesen werden kann. In der ersten Phase macht man sich dagegen zu Nutzen, dass die Wahrscheinlichkeit, eine Verbesserung zu erzielen, groß ist. Beide Phasen lassen sich nun unabhängig voneinander analysieren. Sowohl für die erste als auch für die zweite Phase ergibt sich eine Laufzeit von $O(n \log n)$ und somit ist auch die erwartete Gesamtlaufzeit durch $O(n \log n)$ nach oben beschränkt.

Um als erste untere Schranke $E(X_{\text{OneMax}})= \Omega(\lambda \frac{n}{\log n})$ nachzuweisen, bedient man sich wieder der Methode des typischen Laufs. Die beiden Annahmen sind diesmal:

1. Wir starten mit wenigstens $n/2$ Nullen.
2. Unter n Generationen reduziert *keine* die Distanz zum Optimum um mehr als $\log n$.

Die Fehlerwahrscheinlichkeit zu 1. ist bekanntermaßen höchstens $1/2$. Für die Fehlerwahrscheinlichkeit von 2. ergibt sich nach kurzer Rechnung $e^{-\Omega(\log(n) \log \log n)}$. Damit sind die beiden Annahmen zulässig und man erhält als erwartete Laufzeit unmittelbar $E(X_{\text{OneMax}}) = \Omega\left(\lambda \frac{n}{\log n}\right)$.

Um sicher zu sein, dass der (1+λ) EA dem (1+1) EA für kein λ überlegen ist, zeigt man die weitere untere Schranke $E(X_{\text{OneMax}})= \Omega(n \log n)$ für $\lambda = o\left(\frac{\sqrt{n}}{\log n}\right)$. Wieder geht man von

einem typischen Ereignis aus, nämlich, dass es einen Zeitpunkt gibt, an dem $\text{ONEMAX}(x) \in \{n - \lceil\sqrt{n}\rceil, \ldots, n - \lceil\sqrt{n}/2\rceil\}$ gilt. Ab diesem Zeitpunkt ist die Wahrscheinlichkeit, in $n \log n$ Generationen einen Mutanten y von x mit $\text{ONEMAX}(y) \geq \text{ONEMAX}(x) + 3$ zu erzeugen, $o(1)$ (für $\lambda = o\left(\frac{\sqrt{n}}{\log n}\right)$). Auch dieses Ereignis kann daher vernachlässigt werden und es folgt das gewünschte Ergebnis.

Man weiß nun, dass der (1+λ) EA auf ONEMAX für $\lambda = O\left(\frac{\ln n \cdot \ln \ln n}{\ln \ln \ln n}\right)$ asymptotisch die gleiche erwartete Laufzeit wie der (1+1) EA hat und dass er für $\lambda = \omega(\log^2 n)$ dem (1+1) EA unterlegen ist. Auch wurde gezeigt, dass er nie eine bessere erwartete Laufzeit besitzt als der (1+1) EA. Es zeigt sich außerdem, dass der (1+λ) EA für $\lambda = \omega\left(\frac{\ln n \cdot \ln \ln n}{\ln \ln \ln n}\right)$ tatsächlich langsamer als der (1+1) EA wird.

3.9.4 Zusammenfassung bisheriger Ergebnisse

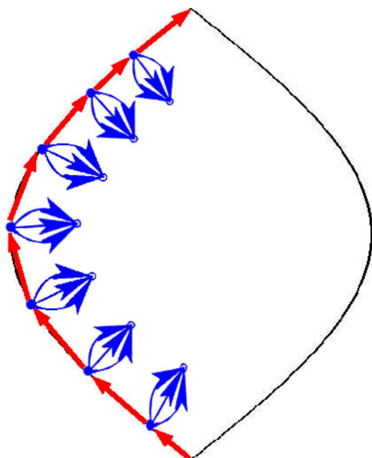
Folgende Schranken wurden für den (1+λ) EA gezeigt:

Funktion	Schranke	Bedingung
LEADINGONES	$O(n^2 + n\lambda)$	
LEADINGONES	$\Omega(n\lambda)$	
LEADINGONES	$\Omega(n^2)$	
ONEMAX	$O(n \log n + n\lambda)$	
ONEMAX	$\Omega(\lambda \frac{n}{\log n})$	
ONEMAX	$\Omega(n \log n)$	
ONEMAX	$\Theta(n \log n)$	$\lambda = O(\frac{\ln n \cdot \ln \ln n}{\ln \ln \ln n})$

3.9.5 ERRINGLIGHTS

Bisher wurden nur sehr einfache Funktionen untersucht, auf denen der (1+1) EA bereits gute Optimerungszeiten zeigt. Es war nicht zu erwarten, dass der (1+1) EA auf diesen Funktionen von dem (1+λ) EA geschlagen wird. Es ist aber möglich, eine Funktion ERRINGLIGHTS zu konstruieren, bei der, für gut gewähltes λ, eine erwartete Laufzeit von $O(n^2)$ des (1+λ) EA einer erwarteten exponentiellen Laufzeit des (1+1) EA gegenüber steht.

Zunächst wird eine Funktion entworfen, die die entscheidenen Kernelemente von ERRINGLIGHTS enthält. Für diese Kernfunktion setzt man allerdings voraus, dass nicht zufällig sondern mit 0^n initialisiert wird.



- Die Funktion hat einen Hauptpfad $0^i 1^{n-i}$.
- Auf dem Hauptpfad befinden sich etwa $\frac{\sqrt{n}}{2}$ Verzweigungspunkte.
- An den Verzweigungspunkten wird der Funktionswert auch durch das Flippen eines der $\lfloor \sqrt{n} \rfloor$ linkesten Bits erhöht.

An Verzweigungspunkten gibt es also eine ganze Reihe von Pfaden, von denen aber nur einer zum globalen Optimum führt.

Die Wahrscheinlichkeit, dass der (1+1) EA diese Funktion in polynomieller Zeit optimiert, ist höchstens $2^{-\Omega(\sqrt{n}\log n)}$. Die Wahrscheinlichkeit, dass der (1+λ) EA mit $\lambda = n\lambda'$, $\lambda' \in \mathbb{N}$ diese Funktion in $O(n^2\lambda')$ Schritten optimiert, ist mindestens $1 - \varepsilon$ für alle $\varepsilon > 0$. Der Vorteil des (1+λ) EA ist, dass er, im Gegensatz zum (1+1) EA, normalerweise keins der lokalen Optima erreicht. An Verzweigungspunkten gibt es sehr viel mehr Wege zum lokalen, als zum globalen Optimum. Es ist also sehr viel wahrscheinlicher, dass ein (1+1) EA zum lokalen Optimum läuft. Der (1+λ) EA erzeugt dagegen mit großer Wahrscheinlichkeit zumindest einen Nachfahren, der eine geringer Distanz zum globalen Optimum hat und da die Funktionswerte in Richtung des Hauptpfades stärker ansteigen, wird dieser Nachfahre bevorzugt.

3.10 Maximale Matchings

Vortragender: Stefan Tannenbaum

Literatur: Die Grundlage dieses Seminarvortrags bildet [14].

Zum Abschluss der Seminarreihe behandelten wir das Verhalten des (1+1) EA auf maximalen Matchings.

3.10.1 Zielsetzung

Das Problem, ein maximales Matching zu berechnen, ist Teil einiger kombinatorischer Fragestellungen und tritt beispielsweise bei Packet-Switching Netzwerken auf. Auf den ersten Blick ähnelt es typischen NP-harten Problemen, dennoch existieren zur Berechnung von maximalen Matchings bereits effiziente deterministische Algorithmen. Daher mag die Verwendung von EAs zur Berechnung von maximalen Matchings zuerst sinnlos erscheinen. Sie ist es auch, ebenso wie die Verwendung von EAs auf Problemen wie Sortieren, OneMax oder Trap vollkommen sinnlos ist. Unser Ziel ist nicht die Anwendung, sondern die Analyse von

evolutionären Algorithmen, um ihr Verhalten besser zu verstehen und so zu Voraussagen über das Verhalten bei in der Realität interessanten Problemen kommen zu können.

Das Problem maximale Matchings zu berechnen ist in diesem Zusammenhang besonders interessant. Es ist so komplex, dass es ein realistisches Modell für andere praktische Optimierungsprobleme liefert, dabei aber dennoch so gut verstanden, dass eine Analyse zumindest einiger wichtiger Eigenschaften gelingen konnte. Es stellt eine erste Anwendung der zuvor an abstrakten Funktionen entwickelten Analysemethoden an einem realistischen Problem dar.

3.10.2 Ergebnisse

Im Vortrag wurde zuerst gezeigt, wie das Problem maximale Matchings, das ja auf Graphen definiert ist, geeignet für einen (1+1) EA kodiert werden kann. Dazu wird jeder Kante ein Bit zugeordnet, alle Kanten deren Bits gesetzt sind, bilden dann das Matching. Die Fitness eines Matchings entspricht der Zahl der gewählten Kanten. Ungültige Matchings erhalten allerdings statt dessen eine negative Fitness, deren Betrag der Zahl sich überschneidender Kantenpaare entspricht.

Mit dieser Kodierung kann gezeigt werden, dass der (1+1) EA eine optimale Lösung in polynomieller erwarteter Zeit bis auf einen konstanten Faktor approximieren kann. Dies ist ein gutes Kriterium dafür, ob ein Problem durch einen EA sinnvoll gelöst werden kann, denn das Ziel einer Optimierung mit evolutionären Algorithmen ist im Allgemeinen, eine gute Lösung in akzeptabler Zeit zu finden.

Ferner wurde im Vortrag der (1+1) EA auf einer Linie, also einem Graph, der nur aus einem einfachen Pfad besteht, analysiert. Der (1+1) EA berechnet hier ein optimales Matching in erwarteter Zeit $O(n^4)$.

Zum Abschluss wurde noch ein Beispiel für ein Klasse von Graphen gegeben, die nur in exponentieller erwarteter Zeit optimiert werden können. Die genaue Klasse der Graphen, die in polynomieller erwarteter Zeit optimiert werden können, ist noch unbekannt. Das Beispiel für exponentielle Laufzeit ist allerdings gradbeschränkt mit Grad 3, bipartit und planar und schließt damit die genannten Klassen als Entscheidungskriterium aus.

3.10.3 Experimente

Im zweiten PG-Semester wurde eine Unterstützung für Matchings in FrEAK eingebaut. Die durchgeführten Experimente stützen die Annahme, dass der (1+1) EA Linien in erwarteter Zeit $\Theta(n^4)$ optimiert (Abschnitte 8.3.1 und 9.3.1). In einem weiteren Vortrag wurde zusätzlich eine Analyse auf Bäumen gezeigt (Abschnitt 7.8), zu der ebenfalls experimentelle Untersuchungen durchgeführt wurden. Langfristig hoffen wir, über das Thema Matchings einen Einstieg in die Analyse praktisch relevanter Probleme zu finden.

Kapitel 4

Analyse- und Designphase

Inhalt

4.1 Analyse-Phase . . . 45
4.1.1 Anforderungsdefinition . . . 45
4.1.2 Pseudozufallszahlengeneratoren . . . 47
4.1.3 Multikriterielle Optimierung . . . 47
4.1.4 Ablaufsteuerung . . . 48
4.1.5 Operatorgraphen . . . 49
4.1.6 Bedienung der graphischen Oberfläche . . . 51
4.1.7 Observer . . . 54
4.1.8 Problembereichsmodell . . . 55
4.1.9 Package-Struktur . . . 59
4.2 Design-Phase . . . 61
4.2.1 Core . . . 61
4.2.2 GUI . . . 65
4.2.3 Observer . . . 66
4.2.4 Sonstige Module . . . 66

4.1 Analyse-Phase

4.1.1 Anforderungsdefinition

Die erste Aufgabe der Analysephase war es, die Anforderungen an unser Projekt festzulegen. Dazu unterschieden wir vier verschiedene Rollen, die der Benutzer unserer Experimentierumgebung einnehmen kann. Dies können verschiedene Benutzer mit verschiedenen Intentionen sein oder ein und derselbe Benutzer, der im Laufe der Benutzung seine Rolle wechselt. Dazu erarbeiteten wir Aufgaben für die verschiedenen Rollen, die mit unserem Programm durchführbar sein sollten.

Der Programmpaketentwickler

Der Programmpaketentwickler ist ein Programmierer, der unseren Quellcode erweitern möchte, um neue Komponenten (im Folgenden *Module* genannt) zu implementieren.

Er soll die Möglichkeit haben, neue Suchräume zu erstellen, neue Fitnessfunktionen zu implementieren und neue Operatoren, Observer und Views zu entwerfen. Später kam die Implementation eigener Stoppkriterien, Populationsmodelle, Mapper und Parameter Controller hinzu. Die erwähnten Klassen werden in Abschnitt 4.1.8 zum Problembereichsmodell näher erläutert.

Unser Ziel war, dass das System von Grund auf erweiterbar sein sollte, wozu unter anderem klare Hierarchien und Strukturen und eine ausführliche und verständliche Dokumentation gehörten.

Der Algorithmendesigner

Der Algorithmendesigner möchte die graphische Oberfläche unseres Programms nutzen, um Algorithmen zu erstellen. Anders als der Programmpaketentwickler möchte er keine Quelltexte schreiben, sondern nur mit Mitteln arbeiten, die die GUI bereit stellt.

Für den Algorithmendesigner sollte unser System die Möglichkeit bieten, die verschiedenen Komponenten eines Algorithmus auszuwählen und zu konfigurieren. Dazu zählt sowohl die Auswahl von Suchräumen, Fitnessfunktionen, Operatoren und Stoppkriterien als auch die Einstellung der Parameter.

Der Optimierer

Der Optimierer sieht seine Hauptaufgabe weniger im Design der Algorithmen, sondern eher in der Steuerung des Optimierungsprozesses.

Unser Programm sollte die Möglichkeit bieten, mit vorgefertigten Algorithmen schnell zu einem laufenden Algorithmus vorstoßen zu können, damit der ungeduldige Benutzer nicht die Geduld verliert.

Danach möchte der Optimierer verschiedene Läufe des Algorithmus starten, an beliebigen Stellen anhalten und schließlich eine möglichst optimale Lösung erhalten.

Der Auswerter

Der Auswerter möchte einen laufenden Algorithmus analysieren und auswerten.

Dazu soll ein Replay-Modus implementiert werden, mit dem man jederzeit zu bereits vergangenen Läufen und Generationen zurück springen kann, um interessante Stellen aufzuspüren. Unser Programm sollte weiterhin die Auswahl und Anzeige verschiedener Daten und Maße unterstützen und eine persistente Speicherung der Läufe ermöglichen.

Grenzen

Zusätzlich zu den gewünschten Funktionen erstellten wir eine Liste von nicht erwünschten Funktionen, um uns selbst Grenzen zu setzen und keine Energie auf die Realisierung unwichtiger Funktionen zu verschwenden.

- Wir waren uns einig, dass wir kein verteiltes System realisieren wollten, da dies in Anbetracht der äußerst knappen Zeitvorgabe den Rahmen sprengen würde.
- Eine Grenze, die bereits im PG-Antrag vorgegeben war, war die Beschränkung auf endliche, diskrete Suchräume, da unendliche Suchräume diverse Probleme nach sich ziehen, mit denen wir uns nicht beschäftigen wollten.
- Der Gedanke an Suchräume, die sich aus einem kartesischen Produkt beliebiger Unterräume zusammen setzen lassen (z. B. eine Permutation mit einem Bitstring), wurde verworfen, da auch die Operatoren aus Teiloperatoren für die Unterräume zusammen gesetzt werden müssten.
- Selbstadaptation, Co-Evolution und Meta-Evolution sollten keine vorrangigen Ziele sein, da ihre hohe Komplexität den Rahmen deutlich sprengen würde.

4.1.2 Pseudozufallszahlengeneratoren

Da Zufall bei evolutionären Algorithmen eine zentrale Rolle spielt, suchten wir nach einer einfachen Möglichkeit, gute Pseudozufallszahlengeneratoren in FrEAK zu integrieren. Wir informierten uns über gängige Zufallszahlengeneratoren und suchten nach einer freien Bibliothek, die einige allgemein als gut anerkannte effiziente Generatoren enthält. Wir entschieden uns dann dafür, die Colt Bibliothek von Hoschek (siehe auch Abschnitt 5.4.1) zu verwenden, da sie unseren Anforderungen am besten entspricht.

Zusätzlich beinhaltet die Colt Bibliothek effziente Methoden um die gängigsten Verteilungen zu erzeugen. Wir haben uns in FrEAK für den Pseudozufallszahlengenerator Mersenne Twister [30] entschieden, da dieser schnell ist und relativ gute Pseudozufallszahlen liefert.

Es war angedacht, bei einem Lauf den Pseudozufallszahlengenerator mit auswählen zu können. Diese Option wäre noch leicht zu implementieren, wurde von uns aber wieder verworfen, da sie sich als nicht notwendig erwiesen hat.

4.1.3 Multikriterielle Optimierung

Im Hinblick auf den Vortrag über multikriterielle Optimierung im zweiten PG-Semester entschlossen wir uns nachträglich, die Unterstützung multikriterieller Fitnessfunktionen in FrEAK zu integrieren. Dieses Ziel geht über das Minimalziel der Projektgruppe hinaus, in Anbetracht des Nutzens und des geringen Aufwandes seiner Umsetzung erschien es uns aber wünschenswert.

4.1.4 Ablaufsteuerung

Die Ablaufsteuerung wurde bereits sehr früh ausgearbeitet, da sie sehr zentral ist und viele Teile des Programms bestimmt. Einige Anforderungen standen bereits vor der Diskussion fest und bestimmten das Design.

- FrEAK musste in der Lage sein, mehrere Läufe eines evolutionären Algorithmus mit jeweils anderen Pseudozufallszahlen zu simulieren.
- Die Läufe sollten exakt reproduzierbar sein.
- Der Benutzer sollte den Algorithmus auswerten können. Sinnvoll wäre besonders eine Anzeige von Ergebnissen bereits während des Laufs.

Wir entschieden uns, zur Erfüllung der ersten Anforderungen ein Scheduleobjekt zu verwenden. Der Algorithmus wird als Satz von Modulen, die Teil des Schedule sind, repräsentiert. Beispielsweise enthält jedes Schedule eine Fitnessfunktion und eine Menge an Stoppkriterien. Die konkrete Wahl und Konfiguration der Module bestimmen den konkreten Algorithmus. Durch die ebenfalls im Schedule enthaltenen Batches wird auch die genaue Anzahl der zu simulierenden Läufe bestimmt.

Zur Erfüllung der zweiten Anforderung fügten wir den Pseudozufallszahlengenerator als Modul in das Schedule ein. Ein bestimmtes Schedule erzeugt damit immer denselben Ablauf.

Nachdem ein Schedule zusammengestellt worden ist, kann es als Datei gespeichert werden. Dadurch ist es auch später möglich, den Ablauf zu reproduzieren. Zusätzlich kann eine Berechnung auch leicht auf entfernt stehenden Rechnern gestartet werden, indem nur das entsprechende Schedule auf den Rechner kopiert wird.

Zur Auswertung der Ergebnisse entwickelten wir das Konzept der Observer. Ein Observer ist ein Modul, das Informationen aus dem Ablauf extrahieren und darstellen kann. Um einen Ansatzpunkt zum Einfügen der Observer in den Ablauf zu haben, versenden die Module des Schedule Events, auf die sich Observer registrieren können. Observer werden in Abschnitt 4.1.7, das Eventsystem in Abschnitt 4.2.1 näher erläutert.

Zusätzlich zu diesen Mindestanforderungen entschieden wir uns, einige weitere Funktionen zu bieten.

- Die Läufe sollten wie ein Video zu steuern sein. Sie sollen jederzeit unterbrochen werden können, vergangene Abschnitte sollten jederzeit wiederholbar sein.
- Die Läufe in einem Schedule sollten sich in gewissen Aspekten unterscheiden können. Gewünscht waren beispielsweise Größe der Population und Dimension des Suchraums.

Um den ersten Punkt zu realisieren, wird in gewissen Zeitabständen eine Kopie des Schedules gespeichert. Ausgehend von diesen Kopien können alle Zustände schnell erreicht werden. Dieses Feature erschien zuerst als Spielerei, aber vor allem die Fähigkeit, kurz zurück zu springen und eine interessante Stelle nochmal zu wiederholen, erwies sich als sehr nützlich.

Kontrovers diskutiert wurde die Frage, ob zwischen den verschiedenen Läufen eines Schedule ein Unterschied im Algorithmus bestehen darf, und falls ja, wie viel Flexibilität gewünscht wird. Zur Diskussion stand, für jeden Lauf neue Instanzen der Module anzulegen, und damit eventuell auch ein Modul zu wechseln. Dieses Vorgehen ist jedoch problematisch, da die Observer, die nicht in jedem Lauf neu erzeugt werden sollen, ihre Eventquellen verlieren.

Wir entschieden uns dafür, alle Läufe mit denselben Instanzen durchzuführen, diese Instanzen aber während der Simulation umkonfigurierbar zu halten. Mittlerweile existiert mit dem Eventcontroller ein System, dass den Austausch der Module doch erlaubt. Es wird jedoch nur beim Zusammenstellen des Schedules und nicht während der Simulation genutzt.

Im Laufe der PG sind die Anforderungen an die Flexibilität der Module erheblich über das ursprünglich angepeilte Maß hinaus gestiegen. Relativ bald wurde das System der Batches eingeführt, das eine Änderung der Konfiguration eines Moduls während der Simulation gestattet.

Die gesamte Simulation wird in mehrere Batches zu je mehreren Läufen aufgeteilt. Die Konfiguration eines Moduls lässt sich dann pro Batch einstellen. Zuerst waren nur Änderungen an bestimmten Modulen erlaubt. Durch ein umfangreiches Redesign der Initialisierung der Module im zweiten PG-Semester sind mittlerweile beliebige Änderungen an beliebigen Modulen möglich.

4.1.5 Operatorgraphen

Gleich in der zweiten PG-Sitzung stellte ein PG-Teilnehmer, der nach bestehenden Paketen zu evolutionären Algorithmen recherchiert hatte, die Idee vor, Algorithmen als Operatorgraphen zu realisieren. Weitere Anregungen hierzu lieferte unter anderem ein Artikel von Rummler und Scarbata [38] und die diesem Artikel zugrunde liegende Bibliothek eaLib [37].

Die Knoten in einem solchen Graphen stellen Operatoren dar, die die eingehenden Individuen verarbeiten und weiterleiten. Die Individuen der aktuellen Generation werden zur Erzeugung einer neuen Generation auf Kanten von einem Startknoten zu einem Zielknoten geleitet, an dem die neue Generation abgelesen wird.
Optional kann ein graphischer Editor bereitgestellt werden, mit dem man Operatorgraphen zusammenstellen kann.

Diese Idee fand neben anfänglicher Bedenken zur Realisierbarkeit und zum möglichen Mehraufwand großen Anklang, da sie eine sehr flexible und leicht erweiterbare Realisierung evolutionärer Algorithmen darstellt, die weit über die Mindestanforderungen der PG hinaus geht.

Ein Komitee aus vier interessierten Freiwilligen sollte das Konzept durchdenken, ausarbeiten und die Frage der effizienten Realisierbarkeit klären.

Mächtigkeit der Operatorgraphen

Zum einen stand die Frage im Raum, ob das System mächtig genug ist, um unsere Ansprüche zu erfüllen. Dazu wurden gängige Algorithmen beispielhaft durch Operatorgraphen modelliert

und vorgestellt. Es zeigte sich, dass die betrachteten Algorithmen (u. a. der $(\mu + \lambda)$ EA und der (μ, λ) EA) relativ einfach und elegant modelliert werden konnten.

Bei exotischen Algorithmen oder starken Abhängigkeiten der Operatoren untereinander ist es zudem stets möglich, einen ganzen Algorithmus als Knoten zu modellieren. Dies ist zwar nicht Sinn und Zweck der Operatorgraphen, aber es zeigt zumindest die Existenz einer schnell umsetzbaren Lösung. Das Graph-Komitee empfindet eine solche Lösung jedoch nur als letzten Ausweg; das Ziel muss sein, hierfür elegantere Lösungen zu finden.

Parameter Controller

Als mögliche Lösung für Abhängigkeiten zwischen Operatoren kam die Idee auf, unabhängige Kontrollobjekte einzuführen, die Werte von Operatoren kontrollieren können. Diese *Parameter Controller*, wie wir sie nannten, sollen auf Ereignisse (Events) im Graphen lauschen und die Werte der kontrollierten Operatoren entsprechend anpassen.

Wir wählten diese Lösung, da so die Parameterkontrolle getrennt von den Operatoren abläuft. Dies hat mehrere Vorteile:

- Der Benutzer kann genau die gewünschten Parameter kontrollieren.
- Die Operatoren müssen nur entsprechende Schnittstellen bereit stellen, benötigen aber keinerlei Logik zur Parameterkontrolle.
- Parameter Controller sind erweiterbar, man kann ohne Probleme neue Parameter Controller implementieren.

Graphische Oberfläche

Der Plan, eine graphische Oberfläche für die Operatorgraphen bereit zu stellen, klang anfangs für die PG-Teilnehmer recht abenteuerlich. Allerdings wird die Komplexität eingeschränkt durch die Bedingungen, die an die Graphen gestellt werden: da die Graphen nur für die Erzeugung einer Generation verwendet werden, müssen die Operatorgraphen azyklisch sein und die Individuen laufen immer vom Startknoten zum Endknoten.

Andere Softwarepakete zu evolutionären Algorithmen modellieren den gesamten Ablauf vom Start des Algorithmus bis hin zur Erfüllung des Stoppkriteriums als Graphen. Dieser enthält dann meist einen Zyklus, in dem aus der alten Generation die neue Generation erzeugt wird. Wir wählten jedoch den Weg, Ablaufsteuerung und Erzeugung der neuen Generation voneinander zu trennen. Unser Graph stellt also nur einen Teil des erwähnten Zyklus' dar, der aus der alten Generation die neue Generation erzeugt. Dies hat den Vorteil, dass so für die Graphen die oben genannten Bedingungen gelten und sie daher einfacher und übersichtlicher sind als zyklische Graphen, die die Ablaufsteuerung direkt integrieren.

Angedacht war eine schichtweise Darstellung der Operatoren, bei der Operatoren entsprechend ihrer (maximalen) Entfernung zum Startknoten gezeichnet werden. Diese Idee wurde jedoch später wieder verworfen, da mit der späteren Realisierung Knoten beliebig verschoben werden konnten und eine Einschränkung nach Schichten nicht notwendig war.

Klassifikation der Operatoren

Wir unterteilten die Operatoren in folgende Klassen:

Crossover: Crossover-Operatoren haben bis auf ein paar Ausnahmen (z. B. Genpool-Crossover) mehrere Eingänge. Bei paarweisen Crossover-Operationen werden Paare von Individuen aus den beiden Eingängen miteinander rekombiniert; hierzu ist es notwendig, dass die Individuen in einer geordneten Datenstruktur vorliegen.

Mutation: alle eingehenden Individuen werden unabhängig voneinander einer Mutation unterzogen; die so erzeugten Kinder werden ausgegeben.

Selection: eine bestimmte Anzahl von Individuen wird selektiert. Zur Steigerung der Performanz werden außerdem die eingehenden Individuen zu einem Ausgang durchgeschleift, so dass sie nicht vorher dupliziert werden müssen, falls sie später noch einmal benötigt werden. Dies ist nützlich z. B. bei $(\mu + \lambda)$-Strategien, wenn wenige Individuen aus einer großen Population selektiert werden sollen und die eingehende Population noch benötigt wird; ansonsten müsste man die große Population vorher duplizieren.

Split: verteilt die eingehenden Individuen auf mehrere Ausgänge und eröffnet so mehrere Wege.

In der Version FrEAK 0.1 waren noch zwei Operatoren Merge und Duplication vorhanden, die den Fluss der Individuen steuern und Individuen von mehreren Eingängen zusammenfügen bzw. für mehrere Ausgänge duplizieren. Im zweiten PG-Semester wurde diese Funktionalität in die Ein- und Ausgänge der Operatoren verlagert, so dass diese beiden Operatoren nicht mehr nötig waren.

4.1.6 Bedienung der graphischen Oberfläche

Die Bedienung von FrEAK muss alle Möglichkeiten widerspiegeln, die zur Umsetzung der Ablaufsteuerung nötig sind. Dazu entwickelten wir am Ende der Analysephase einen groben Ablaufsplan für die graphische Oberfläche.

Der Start

Nach dem Start von FrEAK kann der Benutzer natürlich ein neues Schedule mit allen nötigen Einstellungen erzeugen. Zusätzlich sollte es möglich sein, ein schon erzeugtes Schedule oder ein Replay zu laden, also eine Art „Video“ eines schon berechneten Laufs. Um den Benutzer nicht unnötig zu verwirren, werden ein Schedule und das dazugehörige Replay immer zusammen gespeichert.

Schedule erzeugen

Als erstes muss ein Suchraum aus einer Liste ausgewählt werden, da so gut wie alle anderen Einstellungen von dem gewählten Suchraum abhängig sind. Danach gelangt man zur Auswahl und Konfiguration der Fitnessfunktion. Von hier aus sollte es dann direkt möglich sein, die Läufe bzw. „Batches“ zu editieren, den Graphen festzulegen, den Random Seed zu setzen, und zum Schluss die Observer einzurichten.

Batches

Für statistische Analysezwecke benötigt man Informationen über mehrere Läufe. Außerdem ist es wünschenswert, mehrere Läufe mit verschiedenen Parametereinstellungen zu testen. Zu diesem Zweck kann man in FrEAK mehrere jeweils verschieden konfigurierte Batches erstellen, die nacheinander abgearbeitet werden. Zusätzlich kann in jedem Batch die Anzahl der Läufe eingestellt werden.

Die geplante Möglichkeit, die Konfiguration der Batches zu laden und speichern wurde nicht umgesetzt, da dies durch das Laden und Speichern des Schedules abgedeckt wird.

Der Graph

Am Ende der Analysephase stand die genaue Struktur des Graphen noch nicht 100%ig fest. Deshalb bezogen sich die Planungen an der graphischen Oberfläche nur auf das Laden und Abspeichern des Graphs. Auch soll natürlich ein neuer Graph angelegt und ein bestehender Graph editiert werden können.

Random Seed

Ein Random Seed ist ein Wert, mit dem der Pseudozufallszahlengenerator initialisiert werden kann und der danach bestimmte und reproduzierbare Pseudozufallszahlen liefert. Es war geplant, den Random Seed direkt angeben zu können, um Läufe exakt wiederholen zu können. Da die direkte Eingabe eines Random Seeds durch den Benutzer ansonsten relativ wenig Sinn macht, wurde er in den Hintergrund verbannt und wird nun aus einem durch die Systemzeit generierten, globalen Random Seed erzeugt. Der Schedule speichert den Random Seed mit und man kann mit Hilfe der Replay-Funktion Läufe exakt wiederholen.

Observer auswählen

Zur Observer-Auswahl sollte man kommen, nachdem man die Fitnessfunktion bestimmt, den Graphen festgelegt und die Batches konfiguriert hat. Später allerdings wurden die Batches nach ganz hinten in der Konfigurationsreihenfolge verlegt. Zur Auswahl stehen sollen auf jeden Fall nur Observer, die mit der aktuellen Konfiguration kompatibel sind. Einige Observer sind abhängig von der Fitnessfunktion, weil sie z. B. die Distanz zum Optimum berechnen, aber nicht jede Fitnessfunktion ihr Optimum auch kennt. Zusätzlich benötigen viele Observer den

Graphen, da sie ihre Daten von Events erhalten, die dort erzeugt werden, aber die genaue Eventquelle erst vom Benutzer spezifiziert werden muss.

Das Observer-Setup gliedert sich in zwei Listen von Observern, eine in der alle verfügbaren Observer bzw. Views angezeigt werden und eine in der alle ausgewählten Observer mit hinzugefügten Views stehen. Jeder Observer bzw. View kann beliebig oft hinzugefügt werden. Außerdem ist es natürlich möglich, jeden Observer zu konfigurieren und dessen Eventquellen festzulegen.

Eine genauere Beschreibung der Observer befindet sich im nächsten Abschnitt (4.1.7).

Das Hauptfenster

Das zentrale Fenster während des Laufes beinhaltet alle Views aller ausgewählten Observer und ein Steuerungspanel. In diesem kann ein Lauf angehalten und fortgesetzt werden, die maximale Geschwindigkeit in Generationen pro Sekunde gesetzt werden und es werden der aktuelle Batch, der aktuelle Lauf und die aktuelle Generation angezeigt.

Ebenfalls sind vom Hauptfenster Standardaufgaben, wie das Laden, Speichern und Aufrufen der integrierten Hilfefunktion möglich.

Schedule editieren

Auch während eines angehaltenen Laufes soll es möglich sein, den Operatorgraphen zu editieren und Parameter der Operatoren zu ändern. Weiterhin sollen Observer hinzugefügt oder entfernt und zusätzliche Batches angelegt werden können. Viel später entschlossen wir uns, dass die erst in der Implementierungsphase hinzugekommenden Stoppkriterien auch während eines Laufes gewechselt werden können.

Replaymodus

Es ist in FrEAK jederzeit möglich, bereits vergangene Abschnitte einer Simulation zu betrachten (Replay). Dazu kann man schrittweise vor und zurück springen, den gewünschte Zeitindex direkt eingeben oder ihn innerhalb des aktuellen Laufs per Slider auswählen. Tatsächlich ist, anders als bei Fußballübertragungen, sogar das Anspringen zukünftiger Stellen möglich, was aber selten genutzt wird.

Headless Mode

Die ersten Versionen von FrEAK waren auf eine graphische Oberfläche angewiesen. Da dazu auf den Unix-Rechnern der Universität der pro Rechner nur einmal vorhandene X-Server über Tage blockiert werden musste, wurde eine Möglichkeit geschaffen FrEAK ohne graphische Oberfläche zu starten.

Im Headless Mode wird FrEAK beim Start eine Datei mit einem bereits fertig definierten Schedule übergeben. FrEAK berechnet dieses Schedule und kann dabei über Observer die

gewünschten statistischen Daten in Dateien schreiben. Zusätzlich wird im Headless Mode in regelmäßigen Abständen eine Sicherheitskopie des aktuellen Schedules geschrieben, so dass bei einem Absturz des Rechners nicht die gesamte Berechnung neu gestartet werden muss.

4.1.7 Observer

Zur Auswertung der Ergebnisse entwickelten wir das Konzept der Observer. Ein Observer ist ein Modul, das eine bestimmte interessante Information aus dem Ablauf extrahieren und mit Hilfe eines Views darstellen kann. Diese Information könnte beispielsweise die Liste der Fitnesswerte aller Individuen der aktuellen Generation oder der Genotyp des besten bisher bekannten Individuums sein.

Observer mit Views können ihre Informationen auf dem Bildschirm anzeigen oder zur späteren Analyse in eine Datei schreiben. Um einen Ansatzpunkt zum Einfügen der Observer in den Ablauf zu haben, versenden die Module des Schedule Events, auf die sich Observer registrieren können. Der Vorteil dieses Systems gegenüber den verbreiteten Logfiles ist, dass nur genau die Informationen aufgezeichnet werden, die auch tatsächlich benötigt werden.

Im Folgenden findet sich eine Auflistung aller Observer, die am Ende der Analysephase zum Grundumfang des Programms gehören sollten.

Die Observer

Durchschnittliche Fitness Die durchschnittliche Fitness aller Individuen der aktuellen Generation wird berechnet.

Bestes je gesehenes Individuum Das beste im Rahmen eines Laufes bisher aufgetretene Individuum wird gespeichert. Insbesondere sollten hier auch Individuen berücksichtigt werden, die durch Mutation oder Rekombination zwar kurzzeitig entstehen, aber aus Gründen der Selektion nicht in die folgende Generation aufgenommen wurden.

Aktuelle Population Dieser Observer speichert lediglich die gesamte Population. Er macht prinzipiell nichts, ist aber einer der wichtigsten Observer überhaupt.

Diversität Es wird die Diversität innerhalb der Population berechnet.

Bestes Individuum der aktuellen Generation In jeder Generation wird das beste Individuum bzw. dessen Fitness gespeichert.

Fitnessvarianz Es wird die Stichprobenvarianz aus der Liste der Fitnesswerte der aktuellen Population berechnet.

Neuer Genotyp Für jede Generation wird bestimmt, welche neu entstandenen Genotypen der Population hinzugefügt wurden.

Selektionsintensität Die Selektionsintensität bei den Generationsübergängen wird berechnet.

Sortiert nach Fitness Dieser Observer sollte durch den gesamten Lauf eine Liste der existierenden Genotypen, sortiert nach ihrem Fitnesswert führen. Zu jedem Genotyp sollte gespeichert werden, wie viele Individuen dieses Typs der aktuellen Generation angehören. Dieser Observer war in erster Linie nicht von unabhängigem Interesse, sondern sollte vor allem die effiziente Berechnung der durchschnittlichen Fitness, der Fitnessvarianz und des fittesten Individuums vereinfachen.

Aufwändige Berechnungen, die von mehreren aktiven Observern benötigt werden, sollen nur einmal zentral durchgeführt werden, indem Observer auf die Daten anderer Observer zugreifen können. Dies wurde später wieder verworfen.

Die Views

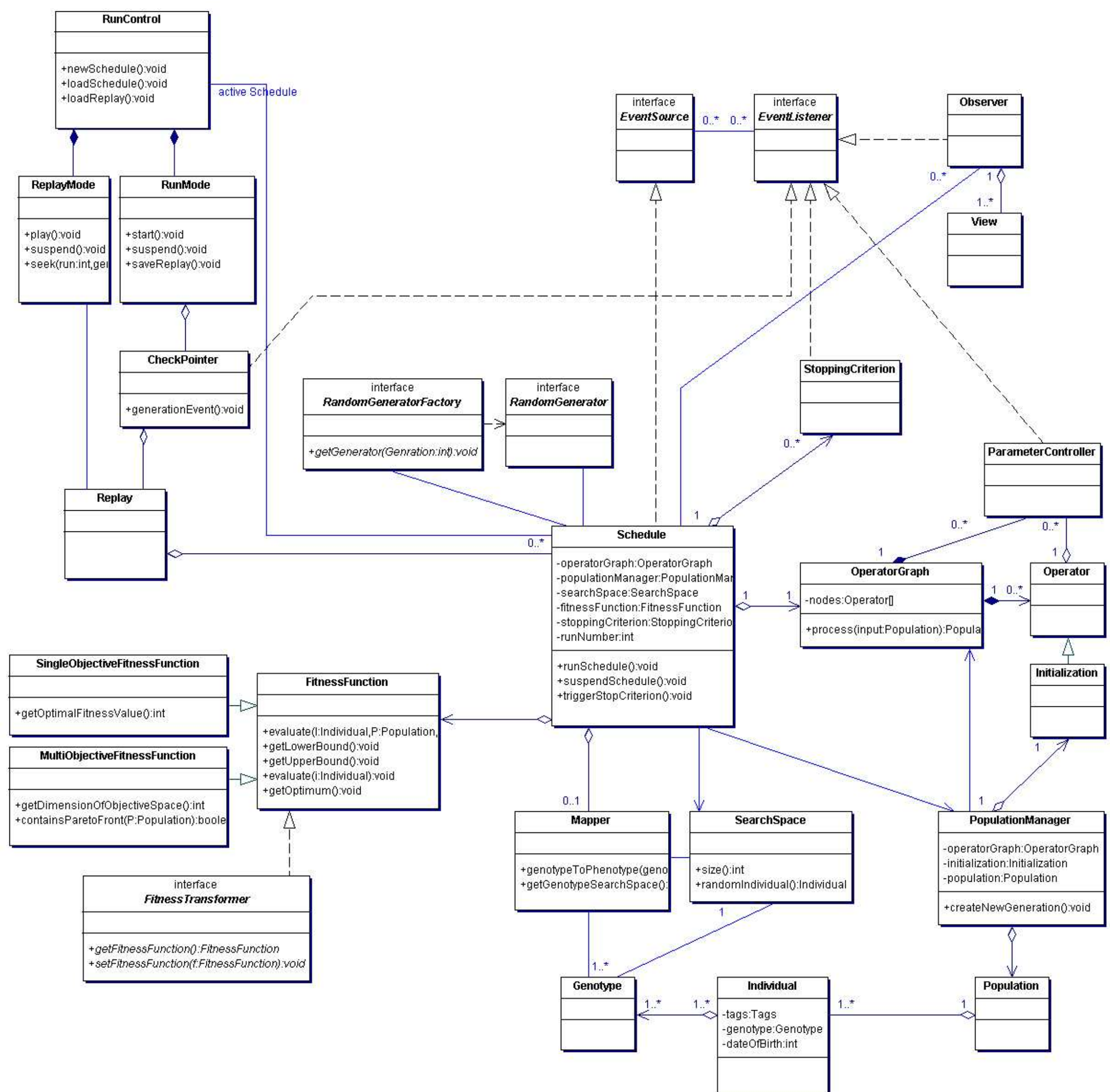
Da nicht jeder Observer mit jedem View kombiniert werden kann, soll jeder einzelne Observer Information enthalten, welche Views mit ihm funktionieren. Jeder Observer kann aber auch mit mehreren kompatiblen Views kombiniert werden.

Benötigte Events

Für die bisherige Liste von Observern genügt es in fast allen Fällen, sie einmalig am Ende eines Generationsübergangs zu informieren. Dies setzt allerdings voraus, dass aus den Eigenschaften der Individuen ersichtlich ist, ob sie beim letzten Generationsübergang entstanden sind. Eine Ausnahme bildete „Bestes je gesehenes Individuum“. Wenn nicht ausgeschlossen werden kann, dass auch beste Individuen der Selektion zum Opfer fallen können, müsste ein entsprechendes Event schon bei der Erzeugung jedes Individuums generiert werden. Dies wurde aber später wieder verworfen.

4.1.8 Problembereichsmodell

Aus den bisherigen Erkenntnissen wurde ein Problembereichsmodell erstellt, ein erstes abstraktes und stark vereinfachtes Modell der wichtigsten Klassen und Hierarchien. Modelliert wurde es als UML-Klassendiagramm.



Das Herzstück des Problembereichsmodells ist der `Schedule`, der Referenzen auf alle Komponenten des Algorithmus verwaltet und z. T. die Ablaufsteuerung des Algorithmus umsetzt. Der Schedule besitzt eine Referenz zum `OperatorGraph`, dessen Knoten `Operator`-Objekte sind. Der `OperatorGraph` erhält eine Multimenge von Individuen, führt diese durch die Operatoren und liefert Individuen der neuen Generation zurück. Daneben enthält der Graphen noch `ParameterController`, die Parameter von Operatoren steuern können.

Individuen werden durch die Klasse `Individual` repräsentiert. Ein Individuum enthält neben seinem Geburtsdatum und einem Cache für den aktuellen Fitnesswert ein `Genotype`-Objekt, das die reinen Gendaten repräsentiert. Semantisch gehören Genotypen dem Genotyp-Suchraum an, eine Instanz der Klasse `SearchSpace`. Diese liefert sowohl allgemeine Infor-

mationen zur Dimension und Größe des Suchraums, als auch die Möglichkeit, gleichverteilt zufällige Genotypen zu erzeugen. Neben dem Genotyp-Suchraum gibt es auch den Phänotyp-Suchraum, der zur Fitnessauswertung verwendet wird. Ein optionaler `Mapper` sorgt für die Abbildung von Genotypen zu Phänotypen. Falls kein Mapper vorhanden ist, entspricht der Phänotypsuchraum dem Genotypsuchraum.

Individuen werden in Populationen der Klasse `Population` gespeichert, was sowohl für die aktuelle Population des Algorithmus gilt als auch für Individuen, die durch den Graphen wandern. Zwischen den `Schedule` und den `OperatorGraph` wurde eine Klasse `PopulationManager` geschaltet. Dieser bekam die Funktion, Populationsmodelle zu realisieren und aus der aktuellen Generation die Individuen der neuen Generation zu erzeugen: der `Schedule` erteilt dem `PopulationManager` die Anweisung, die nächste Generation zu erzeugen. Dazu wählt sich der `PopulationManager` die aktuelle Population oder beliebige Teilmengen daraus und lässt diese Individuen durch den Operatorgraphen laufen. So lassen sich Populationsmodelle realisieren: beispielsweise werden parallele Multistarts einzelner Individuen dadurch umgesetzt, dass der entsprechende `PopulationManager` die Individuen nacheinander und separat durch den Operatorgraphen laufen lässt. Die initiale Population des `PopulationManager`s wird durch einen `Initialization`-Operator erstellt, so dass man hier beliebige Strategien zur Erzeugung der initialen Population einbinden kann.

Zur Fitnessauswertung entwarfen wir die folgenden Klassen: die Klasse `FitnessFunction` ist eine allgemeine Oberklasse für Fitnessfunktionen, die die Auswertungen von Individuen ermöglicht. Sie liefert optionale zusätzliche Informationen über die Fitnessfunktion wie z. B. obere und untere Schranken für den Fitnesswert und das eindeutige globale Optimum, falls existent und bekannt. Bei der Fitnessauswertung wird zusätzlich zu dem Individuum, dessen Fitness bestimmt werden soll, die zugehörige Population übergeben, so dass auch Fitnesswerte, die von den umgebenden Individuen abhängen (z. B. beim Niching), berechnet werden können. Eine Abgrenzung von statischen Fitnessfunktionen, die nur vom aktuellen Individuum abhängen, wird zu diesem Zeitpunkt abgelehnt. Erst in der Testphase wird eine solche Unterscheidung realisiert, um den Fitness-Cache der Individuen ausnutzen zu können und dadurch die Performanz zu steigern.
Von `FitnessFunction` abgeleitet sind die beiden spezielleren Klassen `SingleObjectiveFitnessFunction` und `MultiObjectiveFitnessFunction`. Diese bieten Methoden an, die weitere zusätzliche Informationen liefern. Bei monokriteriellen Fitnessfunktionen ist das der optimale Fitnesswert, falls es einen solchen gibt und er bekannt ist, und bei multikriteriellen Funktionen werden die Dimension des Bildbereiches als Information zur Verfügung gestellt und eine Methode, die für eine gegebene Population testet, ob sie Individuen enthält, die die komplette Pareto-Front repräsentieren.
Das Interface `FitnessTransformer` repräsentiert spezielle Fitnessfunktionen, die eine andere Fitnessfunktion umschließen und deren Fitnesswerte transformieren. Dies kann verwendet werden, um beispielsweise affine Transformationen anzuwenden oder Fitness-Sharing zu realisieren.

Die eigentliche Ablaufsteuerung wird von der Klasse `RunControl` übernommen, der der momentan aktive Schedule zugeordnet ist. Ursprünglich waren zwei Subklassen für die zwei Modi `RunMode` und `ReplayMode` geplant, diese wurden jedoch später in die `RunControl` direkt integriert. Dazu gibt es einen `CheckPointer`, der in bestimmten Abständen den aktuellen `Schedule` speichert und daraus Replays erzeugt. Replays werden durch die Klasse `Replay`

repräsentiert.

Die Klasse `StoppingCriterion` realisiert Stoppkriterien, die den Algorithmus beim Eintreten einer festgelegten Bedingung stoppen. Die Klasse wurde in der Designphase entfernt, da prinzipiell jedes Modul zu jeder Zeit in der Lage sein sollte, den Lauf zu stoppen. Zu diesem Zweck erhielt der Schedule eine entsprechende `triggerStop`-Methode, die einen Abbruch des Laufs nach der aktuell bearbeiteten Generation auslöst.

Gegen Ende der Implementierungsphase wurde aber ersichtlich, dass es für den Benutzer mehr als nur wünschenswert ist, Stoppkriterien in einem eigenen Panel direkt auswählen zu können. Daher wurde eine Reihe von Stoppkriterien implementiert, die sich automatisch auf ein passendes Event registrieren und dann, falls ihre Bedingung erfüllt ist, den Lauf durch Aufruf der Methode im Schedule stoppen.

Im zweiten Semester wurde zusätzlich ermöglicht, mehrere Stoppkriterien gleichzeitig zu wählen. Boolsche Funktionen auf Stoppkriterien wurden diskutiert, aber wegen des Aufwands nicht eingeführt. Außerdem machen sie nur dann Sinn, wenn alle Stoppkriterien zur selben Zeit geprüft werden.

Des Weiteren gibt es noch Pseudozufallszahlengeneratoren der Klasse `RandomGenerator`, die von einer zugehörigen `RandomGeneratorFactory` erzeugt werden. Das Konzept der Factories ist ein Entwurfsmuster, das hier eingesetzt wurde, um die Erzeugung der Pseudozufallszahlengeneratoren für die einzelnen Läufe flexibel zu gestalten. Momentan wird zwar nur ein Typ von Factory eingesetzt, der aus der Systemzeit einen globalen Random Seed generiert und damit für jeden Lauf neue Pseudozufallszahlengeneratoren mit eigenem Random Seed erzeugt. Es ist aber im Nachhinein stets möglich, mit einer eigenen Factory die Erzeugung der Pseudozufallszahlengeneratoren zu verändern. So können neue Pseudozufallszahlengeneratoren integriert werden oder die Erzeugung der Random Seeds kann verändert werden.

Das Problembereichsmodell enthält außerdem einige Klassen zum Event-System, das in Abschnitt 4.2.1 näher erläutert wird.

Die Klasse `Observer` spielt eine wichtige Rolle bei der Anzeige und Auswertung der Algorithmen. Observer empfangen Events aus anderen Teilen des Systems und berechnen daraus Daten wie z. B. bestimmte lokale oder globale Maße oder Informationen über die aktuelle Population.

Zu Anfang sollten diese Observer die Daten für den Benutzer selbst visualisieren, später wurden jedoch die Observer nach dem Model-View-Konzept in reine Datensammler (Observer) und Views unterteilt. Das hat den großen Vorteil, dass das Sammeln von Daten (Model) relativ unabhängig von der Visualisierung der Daten (Views) geschieht.
Der Benutzer hat daher die Freiheit, die Art der Visualisierung frei zu wählen und kann die gleichen Daten mit mehreren verschiedenen, parallel laufenden Views anzeigen. Instanzen einer Klasse von Views können mit verschiedenen Observern zusammen arbeiten, falls sie auf den gleichen Datentypen arbeiten; man kann also bestehende Visualisierungen wieder verwenden.

4.1.9 Package-Struktur

Die Klassen des Problembereichsmodells wurden daraufhin in Java-Pakete eingeteilt. Die Einteilung diente dazu, logische hierarchische Strukturen für die Klassen des Systems zu finden und gleichzeitig Hinweise für eine sinnvolle Gruppeneinteilung zu liefern.

Zuerst standen wir vor der Frage, welche Dimension im Hinblick auf Suchraum-abhängige Module die oberste Ebene bilden sollte: der semantische Typ des Moduls (z. B. Fitness) oder der Suchraum?

Vorschlag *A* war, dass der Suchraum an oberster Stelle stehen sollte, wie in folgender Skizze veranschaulicht:

```
[bitstring]
    [fitness]
    [operator]
    ...
[permutation]
    [fitness]
    [operator]
    ...
...
```

Der Vorteil dieser Anordnung ist, dass es leicht ist, Suchräume hinzuzufügen oder zu entfernen, da alle Klassen im Verzeichnis ihres Suchraums liegen.

Vorschlag *B* sah die Suchräume auf der untersten Ebene:

```
[fitness]
    [bitstring]
    [permutation]
    ...
[operator]
    [bitstring]
    [permutation]
    ...
...
```

Diese Konstruktion ist eleganter im Sinne der objektorientierten Programmierung, da es bei Suchraum-abhängigen Modulen gemeinsame abstrakte Oberklassen geben kann, die dann für verschiedene Suchräume implementiert werden. Man kann die Vererbungshierarchie beliebig tief ausbauen und erst zum Schluss erfolgt die Unterteilung in Suchräume.

Dadurch wird zudem die Realisierung Suchraum-unabhängiger Module erleichtert: im Falle, dass alle Module eines bestimmten Typs Suchraum-unabhängig sind (z. B. bei Selektionsoperatoren), entfällt einfach die Differenzierung auf der untersten Ebene.

Wir entschieden uns für Vorschlag *B*, da die Vorteile aus unserer Sicht überwogen. Für Module, die zwar generell Suchraum-abhängig sind, aber auf mehrere Suchräume anwendbar

sind (z. B. auf Permutationen und Zyklen), wurden Packages `common` angelegt, die auf der gleichen Ebene wie die Suchräume liegen.

Die Grobstruktur sah wie folgt aus:

```
[lib]
[src]
    [freak]
        [core]
        [gui]
        [module]
[test]
    [freak]
        [core]
        [gui]
        [module]
```

Im Verzeichnis `lib` sollten alle eingebundenen Pakete von Fremdanbietern untergebracht werden, wie z. B. Pseudozufallszahlengeneratoren. Die Quelltexte der eigentlichen Klassen liegen in `src`, das Verzeichnis `test` kopiert die Package-Struktur von `src` und enthält Testtreiber zum Test der Klassen in `src`.

Im Package `core` waren zunächst nur einige Klassen eingeplant, die für die grundlegenden Funktionen des Systems notwendig sind und die der Benutzer nicht ohne weiteres modifizieren sollte. Das Package `gui` sollte sämtliche Klassen der GUI enthalten und `module` die Module und deren konkrete Implementationen
Für die restliche Gliederung innerhalb der einzelnen Packages waren die jeweiligen Gruppen zuständig, unter Beachtung von Vorschlag B.

Weitere Entwicklungen

Im Laufe der Entwicklung veränderte sich die Package-Struktur allmählich. Die Trennung von `Core` und `Module` wurde konsequenter umgesetzt, indem sämtliche Interfaces und abstrakte Oberklassen für Module in das Package `Core` wanderten. Damit enthält das `Module` Package nur noch die konkreten Realisierungen der Module, die Schnittstellen und abstrakten Oberklassen sind in gleichnamigen Unter-Packages von `Core` zu finden.

Zu jedem Typ von Modul gibt es nun ein Interface, z. B. `FitnessFunction`, und eine abstrakte Oberklasse, hier `AbstractFitnessFunction`. Das Interface stellt die schlanke Schnittstelle zu den übrigen Teilen des Systems dar, während die abstrakte Oberklasse zur leichteren Implementierung eigener Module gedacht ist; sie stellt grundlegende Funktionen bereit, die allen Modulen dieses Typs gemein ist.

Diese Trennung von Interface und abstrakter Klasse hat den Vorteil, dass das Interface nur die Methoden enthalten muss, die zur externen Schnittstelle gehören; alle übrigen Methoden werden erst in der abstrakten Klasse spezifiziert. Zweitens ist es jederzeit möglich, eigene Realisierungen der Interfaces zu implementieren, ohne die abstrakten Oberklassen erweitern

zu müssen. Dies ist nützlich, wenn man grundlegend andere Funktionalität benötigt oder wenn eine Klasse nicht von der abstrakten Oberklasse erben kann, da sie bereits von einer anderen Klasse erbt.

4.2 Design-Phase

4.2.1 Core

Die Core-Gruppe beschäftigte sich mit der Entwicklung des Programmkerns. Dazu gehörte die Ablaufsteuerung, die Realisierung der Operatorgraphen, das Event-System und die Realisierung und Behandlung von Individuen und Populationen.

Eine zweite wichtige Aufgabe der Core-Gruppe war es, Schnittstellen für die anderen Gruppen bereit zu stellen. Dazu erstellte sie die zugehörigen Interfaces und abstrakte Oberklassen aus dem Problembereichsmodell, die dann von den anderen Gruppen verwendet wurden.

Da die anderen Gruppen auf die Resultate der Core-Gruppe angewiesen waren, begann die Core-Gruppe frühzeitig mit der Implementierung, so dass die Datenmodelle und Kontrollklassen zu Beginn der offiziellen Implementationsphase bereits fertig waren.

Ablaufsteuerung

Die erste Aufgabe im Design der Ablaufsteuerung war der Entwurf des Main Loop für die Klasse Schedule. Da der Benutzer die Simulation jederzeit unterbrechen können soll, musste der Main Loop an definierten Punkten (Points) unterbrochen und später fortgeführt werden können. Eine Unterbrechung zu volkommen beliebigen Zeitpunkten war praktisch unmöglich, da möglicherweise wichtige Informationen über den Zustand des Schedule im Callstack gelegen hätten und durch das Speichern des Scheduleobjekts alleine nicht erfasst worden wären.

Das Finden von geeigneten Points erwies sich als sehr einfach, da die Generationen von evolutionären Algorithmen den Main Loop in natürlicher Weise in Abschnitte teilen. Jeder Zeitpunkt zwischen zwei Generationen oder zwischen zwei Läufen wurde ein Point. Jedem Point wurde zur Identifikation ein Zeitindex aus Batchnummer, Laufnummer und Generationsnummer zugeordnet. Das Schedule erhielt eine Methode `step`, die die Simulation jeweils genau bis zum jeweils nächsten Point fortsetzt. Eine Unterbrechungsanfrage des Benutzers führt dann dazu, dass der aktuell laufende Step abgearbeitet, der nächste aber noch nicht begonnen wird. Weitergehende Informationen zu Unterbrechungsanfragen finden sich im Abschnitt 5.4.6.

Alle passierten Points werden in einem Replayobjekt vermerkt. Gelegentlich wird dabei zusätzlich das komplette Schedule kopiert und im Replay als so genannter Checkpoint gespeichert. Die Checkpoints dienen als Wiederaufsetzpunkte, um ein videoartig steuerbares Replay zu ermöglichen. Um einen beliebigen vergangenen Point anzusteuern, ist es nur erforderlich, das Schedule vom letzten Checkpoint vor dem gewünschten Point aus dem Replay zu laden und dann die verbleibenden Steps zum gewünschten Ziel zu berechnen.

Obwohl noch nicht vollkommen klar war, ob Benutzer ein Schedule auch nach seinem Start noch editieren können sollen, sahen wir eine Unterstützung dafür vor. Editiert der Benutzer ein Schedule, während es auf einem Point unterbrochen ist, wird ein Editpoint angelegt. Ein Editpoint ist ein speziell markierter Checkpoint. Wird ein Replay über einen Editpoint hinweg gespielt, wird der Editpoint grundsätzlich geladen und ersetzt das laufende Schedule. Dadurch kann das Replay eine Simulation mit beliebigen Benutzereingriffen reproduzieren.

Bereits in der Designphase wurde dieses System implementiert, um seine Realisierbarkeit zu testen. Dazu wurde eine einfache Benutzeroberfläche erstellt, die auch gleichzeitig die Kommunikation der endgültigen Benutzeroberfläche mit der Ablaufsteuerung testet. Das System wurde in der Implementierungsphase fertig gestellt, zwischen dem ersten und zweiten Semester aber neu entworfen, um mehr als nur simple Unterbrechungsanfragen während des Laufs behandeln zu können (siehe Abschnitt 5.4.6).

Ein geeigneter Algorithmus, der Checkpoints in geeigneten Abständen erstellt und auch bei Speicherplatzmangel wieder löschen kann, wurde auf die Implementierungsphase aufgeschoben, da er kompliziert und für den Rest der Implementierung nicht wesentlich war. Mittlerweile ist auch dieser implementiert.

Das Event-System

Geplant war ein gebräuchliches Event-System, wie es u. a. in `Java Swing` verwendet wird: *Eventquellen* versenden bei bestimmten Ereignissen *Events*, das sind Objekte, die ein Ereignis repräsentieren und meist zusätzliche Informationen zu diesem beinhalten.

So genannte *Eventlistener* können diese Events empfangen. Dazu müssen sie ein entsprechendes Interface implementieren, das sie als Listener auszeichnet und Methoden bereit stellt, die beim Eintreffen eines Events aufgerufen werden.

Ein Eventlistener kann sich nun als Listener bei einer Eventquelle registrieren. Wenn ein Ereignis auftritt, feuert die Eventquelle Events, indem sie alle registrierten Listener über das Ereignis informiert. Dies geschieht, indem die entsprechende durch das Eventlistener-Interface spezifizierte Methode aufgerufen wird; meist mit dem eingetretenen Event als Parameter.

Die Event-Klassen werden nach verschiedenen Event-Typen unterteilt, so dass es verschiedene Events gibt, zu denen jeweils spezielle Eventquellen und spezielle Eventlistener gehören.

Später musste die direkte Kommunikation zwischen Eventlistener und Eventquelle teilweise einer indirekten Kommunikation über eine zentrale Kontrollklasse weichen, da es Probleme beim Austausch von Eventquellen gab, siehe Abschnitt 5.4.4.

Operatorgraphen

Die Operatorgraphen wurden zu Beginn der Designphase innerhalb der Core-Gruppe kontrovers diskutiert. Insbesondere stellte sich die Frage, wie der Ablaufsteuerung des Graphen funktionieren sollte.

Es standen mehrere Vorschläge zur Diskussion:

1. In einem Preprocessing ermittelt der Graph eine Aufteilung der Knoten in Schichten und arbeitet sie schichtweise ab.

2. Die Knoten steuern sich untereinander: der aktive Knoten ruft seine Folgeknoten auf und übergibt ihnen seine Ausgabedaten. Diese speichern die eingehenden Daten für all ihre Eingänge und werden genau dann aktiviert, wenn alle Eingänge mit Daten gefüllt sind.

3. Verwendung einer Queue: der Graph erhält eine Queue mit den Knoten, die bearbeitet werden können. Begonnen wird mit dem Start-Knoten. Der gerade bearbeitete Knoten übergibt seine Daten seinen Folgeknoten. Wenn ein solcher Folgeknoten merkt, dass all seine Eingänge belegt sind, meldet er sich beim Graphen als fertig und wird in die Queue fertiger Knoten eingefügt.

Wir entschieden uns schließlich für den dritten Vorschlag, da die anderen beiden Vorschläge folgende Nachteile besitzen.

- Der Nachteil an Vorschlag 1 ist, dass der Graph inkonsistente Daten enthalten kann, falls zur Laufzeit Knoten entfernt werden oder neu hinzukommen. Zweitens ist es im Sinne der objektorientierten Programmierung angemessener, wenn die Knoten untereinander direkt die Individuen verarbeiten.

- Der zweite Vorschlag hat den Nachteil, dass der Call-Stack für die aufgerufenen Methoden sehr groß wird, da die Methodenaufrufe für alle Knoten ineinander geschachtelt werden. Das wäre bei Graphen realistischer Größe nicht weiter schlimm, doch werden auch die jeweiligen Individuen stets mit gespeichert. Eine vorzeitige Freigabe dieser ist nicht möglich, da sie als Parameter im Call-Stack fest integriert sind.

Der gewählte Vorschlag 3 hat die Flexibilität von Vorschlag 2 ohne die Nachteile des großen Call-Stacks.

Eine weitere Diskussion war die Frage, ob Kanten als eigene Objekte repräsentiert werden sollten oder ob die Kanten implizit durch (geordnete) Adjazenzlisten in den Knoten dargestellt werden sollen.

Wir entschieden uns für die folgende Modellierung: Knoten besitzen Inports und Outports, die eingehende und ausgehende Endpunkte der inzidenten Kanten darstellen. Eine Kante (u, v) wird somit durch einen Inport am Knoten u und einen Outport am Knoten v repräsentiert.

Mit dieser Lösung ist eine geordnete Zuordnung der Kanten gewährleistet. Zweitens können die Inports und Outports als eigenständige Objekte die Logik für das Versenden und Zwischenspeichern der Individuen übernehmen. Dies ist besonders wichtig im Hinblick auf Events, da Outports Eventquellen darstellen und man so Individuen beobachten kann, die über eine bestimmte Kante laufen.

Individuen und Populationen

Auch bei diesem Thema gab es einigen Diskussionsstoff, der im Folgenden erläutert wird.

Datenstruktur für Populationen: Als erstes musste die Frage geklärt werden, mit welcher Datenstruktur die Individuen gespeichert werden sollen. Aus theoretischer Sicht ist eine Population eine Multimenge, weshalb der erste Entwurf eine solche Datenstruktur war. Allerdings mussten wir einsehen, dass die Operatoren im Operatorgraph eine geordnete und stabile Reihenfolge der Individuen benötigten.

Der nächste Gedanke ging daher in Richtung einer linearen Liste, da die Vereinigung zweier Populationen wichtig erschien und eine solche Operation sich mit einer Liste in konstanter Zeit realisieren lässt.

Auch dieser Gedanke wurde dann verworfen, da eine lineare Liste keine effiziente Auswahl eines zufälligen Individuums erlaubt. Da auch andere Operationen außer der Vereinigung wichtig waren und effizient ausführbar sein sollten, kamen wir schließlich zur Klasse `java.util.ArrayList`, die ähnlich zu einem Array einen adressierbaren Speicherzugriff erlaubt und einige gebräuchliche Operationen in amortisierter konstanter Laufzeit realisieren kann.

Rangbasierter Zugriff: Die Population sollte außerdem Zugriffsmethoden bereit stellen, die Individuen nach ihrem Rang zu selektieren. Dabei war unklar, ob die Population die Individuen stets nach ihrer Fitness sortiert speichern oder erst bei einer Anfrage die Berechnungen starten soll.

Wir entwarfen eine Datenstruktur, die die Individuen auf zwei Arten speichert: zum einen nach einem festen und stabilen Index, zum anderen nach ihrem Rang. Allerdings entschieden wir uns schließlich gegen eine solche kombinierte Datenstruktur, da der Overhead zu groß wäre und der Rang von Individuen nicht immer benötigt wird.

Statt dessen wird nun bei der Anfrage ein Quickselect-Algorithmus gestartet, der die gewünschten Individuen in erwarteter linearer Zeit liefert.

Verwendung von Interfaces: Ein Interface namens `IndividualList` repräsentiert eine Liste von Individuen. Die `Population` implementiert dieses Interface und verwendet intern die vorhin besprochene Datenstruktur.

Die Frage, die sich uns stellte war, in welchen Teilen des Systems lediglich das Interface verwendet werden sollte und wann es nötig war, die Klasse `Population` zu verwenden. Um das System möglichst abstrakt und damit flexibel und erweiterbar zu halten, beschlossen wir, in Operatorgraphen ausschließlich das Interface zu verwenden. So bleibt die Möglichkeit, alternative Implementationen einer Population einzusetzen (z. B. eine Implementation, die die Individuen nach ihrer Fitness sortiert speichert).

Notizen: Ein weiteres erwünschtes Feature war, dass Individuen Notizen (auch als „kleine gelbe Klebezettel“ und später als *Tags* bezeichnet) in einer offenen Map speichern sollten. Damit können Individuen z. B. von ParameterControllern oder Observern markiert und auf dem Weg durch den Graphen verfolgt werden, um Abläufe sichtbar zu machen oder adaptive Strategien zu realisieren.

Heftig diskutiert wurde die Frage, ob und wie Tags bei Variationen auf Kinder vererbt werden sollen. Dagegen spricht, dass die Kinder neue Individuen sind, die mit ihren Eltern nicht viel zu tun haben müssen, so dass eventuell Informationen in vererbten Tags ungültig werden. Andererseits wird Vererbung benötigt, um die Entstehung der Individuen zurückverfolgen zu können. Die erste Realisierung sah vor, dass bei der Erzeugung eines neuen Individuums die Eltern mit angegeben werden sollen und das Kind eine Kopie ihrer Tags erhalten soll. Später wurden die Tags um ein zusätzliches Attribut erweitert, dass angibt, ob das entsprechende Tag vererbt werden soll oder nicht.

Vervielfältigung von Individuen und Populationen: Die nächste Frage war, welche Teile von Individuen und Populationen kopiert werden müssen, wenn eine Kopie angelegt werden soll. Aus Performanzgründen sollen Genotypen und Tags von Individuen nur gelesen werden können. Soll schreibend zugegriffen werden, müssen neue Individuen erzeugt werden.

Diese Lösung ist zwar nicht sehr elegant, aber sie verhindert, dass Komponenten ständig kopiert werden, obwohl sie fast nie geändert werden müssen. Dies spart in einfachen Szenarien Laufzeit und Speicherplatz: es reicht nämlich nun aus, bei der Vervielfältigung von Populationen lediglich die Datenstruktur (Liste) an sich zu duplizieren. Die Individuen bleiben gleich und sind daher u. U. in mehreren Populationen vorhanden.

Später wurde dieses Konzept auf die `IndividualLists` ausgedehnt, da die Listen von Individuen in Operatoren nicht verändert wurden. So ist es nun möglich, zur Vervielfältigung von `IndividualLists` lediglich Referenzen zu übergeben. Bei Änderungen an der Liste muss dann eine neue `IndividualList` erstellt werden.

4.2.2 GUI

Die GUI-Gruppe entwarf in der Designphase erste Prototypen für die späteren GUI-Elemente. Die Arbeit wurde in drei Teile aufgeteilt, wobei jeder der drei Mitglieder der Gruppe einen Teil bearbeitete.

Schedule Editor: Der Schedule Editor ist ein Assistent (Wizard), der zur Erstellung und späteren Editierung eines Schedule inklusive aller benötigten Einstellungen dient. Damit verbunden war auch die aufwändige Arbeit, im Hintergrund aus mehreren Komponenten, die teilweise stark voneinander abhängig sind, einen funktionsfähigen Schedule zu erzeugen.

Graph Editor: Der Graph Editor wird aus dem Schedule Editor heraus aufgerufen und dient, wie der Name schon sagt, zur Erstellung des Operatorgraphen in einer graphischen Oberfläche. Als Grundlage diente das unter der GNU Lesser General Public License veröffentlichte Graph-Paket JGraph [1].

RunFrame: Der RunFrame repräsentiert das Haupt-Fenster von FrEAK, das vom Start an angezeigt wird. Er zeigt die Resultate der aktuellen Läufe an und bietet Operationen zur Verwaltung des Schedules und zur Steuerung der Simulation.

Um ein möglichst konsistentes Aussehen der Benutzungsoberfläche von FrEAK zu erreichen, haben wir uns sehr früh dafür entschieden, uns so weit wie möglich an das Java Look&Feel, wie es in den Java Look&Feel Design Guidelines [41] beschrieben ist, zu halten. Als Grafiken für GUI-Elemente sollen vorzugsweise Bilder aus dem von Sun zu Verfügung gestellten Java Look&Feel Graphics Repository [42] verwendet werden. Die Sprache der Benutzungsoberfläche wurde auf amerikanisches Englisch festgelegt.

Die GUI-Gruppe begann die Arbeit frühzeitig und implementierte Prototypen der Fenster und Elemente der Oberfläche. Diese waren nicht auf Funktionalität des Programms angewiesen, wodurch sehr schnell ein Eindruck der Bedienung des Programms gewonnen werden konnte. Dadurch konnte getestet werden, ob die in der Analyse- und Designphase entwickelten Konzepte der Bedienung in der Praxis gut funktionieren würden.

4.2.3 Observer

Die Observer-Gruppe stieß in ihrer Planung auf unerwartete Schwierigkeiten. Ursprünglich war geplant, dass Observer auf den Daten anderer Observer aufbauen sollten. Hier war jedoch unklar, wie dies genau realisiert werden sollte. Als Abhilfe wurde ein `ObserverManager` vorgeschlagen, der alle aktiven Observer kennt und alle Events empfängt. Diese werden dann in einer sinnvollen Reihenfolge an die Observer weitergegeben.
Diese Idee wurde jedoch später fallen gelassen, da sie eine sehr umständliche Realisierung erfordert und geteilte Daten zu selten benötigt werden.

Der `ObserverManager` wurde trotzdem beibehalten und um Funktionalität erweitert, mit der er alle implementierten Observer in den entsprechenden Verzeichnissen suchen und auflisten konnte. Da auch andere Module auf die gleiche Art und Weise gefunden werden mussten, wurde in der Implementierungsphase die Funktionalität in eine neue Klasse `ModuleCollector` übertragen, die mit allen Klassen von Modulen arbeitet.

Parallel dazu suchte die Observer-Gruppe nach einem Plotter, der als Ausgabemodul eingebunden werden konnte. Das unter der GNU General Public License stehende Paket `JPlotter` wurde zu diesem Zweck ausgewählt und getestet. In den späteren Phasen stellten sich jedoch diverse Schwierigkeiten ein, so dass `JPlotter` einem selbst gebastelten, besser angepassten Plotter weichen musste.

4.2.4 Sonstige Module

Das liebevoll als Rest-Gruppe bezeichnete Arbeitsteam beschäftigte sich mit der Auswahl und Dokumentation der später zu implementierenden Suchräume, Fitnessfunktionen und Operatoren im Operatorgraphen. Die genauen Operatoren (in Abschnitt 4.1.5 beschrieben), mit denen sich die Rest-Gruppe befasste, waren die Mutations-, Selektions-, Rekombinations/Crossover- und Split-Module.

Als erstes galt es, die Suchräume auszuwählen. Da unser Programm (vorerst) nur diskrete und endliche Suchräume unterstützen sollte, wurden Suchräume wie $\mathbb{R}^n$ oder Syntaxbäume weg gelassen. Als Suchräume wurden Bitstrings $\{0,1\}^n$ und Permutationen in der Form S_n und

Cycle_n ausgewählt, wobei $\text{Cycle}_n \subset S_n$ ist und alle Zyklen auf n Knoten beschreibt. Etwas später wurde dann noch das kartesische Produkt in der Form $\{0,\ldots,l-1\}^n$ hinzugefügt.

Die Fitnessfunktionen haben wir in echte Fitnessfunktionen und Fitnesstransformer aufgeteilt, wobei Fitnesstransformer nur vor eine echte Fitnessfunktion oder andere Fitnesstransformer geschaltet werden können. Bei dem Suchraum $\{0,1\}^n$ haben wir uns für die relativ einfach zu analysierenden und bekannten Fitnessfunktionen JumpK, LeadingOnes, LongPathK, Needle, OneMax, PathToJump, PathWithTrap, Plateau und Ridge entschieden. Des Weiteren haben wir für diesen Suchraum zwei Standardtestfunktionen für multikriterielle Optimierung implementiert, nämlich Lotz und SineCosine bzw. Moco. Für den Suchraum S_n wurde das Sortierproblem ausgewählt, für Cycle_n das wohl jedem Informatiker bekannte Travelling Salesperson Problem TSP, und für $\{0,\ldots,l-1\}^n$ das Meisterschaftsproblem Champ. In der Implementierungsphase und im zweiten PG-Semester folgten dann noch weitere Fitnessfunktionen.

Als Fitnesstransformer planten wir Fitness Sharing, das nur auf Suchräumen mit einer Metrik funktioniert und einen Affine Transformer, der die Fitness eines Individuums nach der Formel $a \cdot \text{Fitness(Individuum)} + b$ transformiert. Weitere Informationen zu „Fitness Sharing” befinden sich in Abschnitt 3.3.2.

Alle Operatoren, die für die Erzeugung von neuen Individuen nur die Informationen eines Elters benötigen, sind reine Mutationsoperatoren. Für die Suchräume $\{0,1\}^n$ und $\{0,\ldots,l-1\}^n$ wurden Standardbitmutation und k-Bit-Mutation gewählt. Die beiden Suchräume S_n und Cycle_n bekamen die poissonverteilten Operatoren Jump, Exchange und einen, der Jump und Exchange kombiniert. Da S_n und Cycle_n syntaktisch identisch sind, semantisch allerdings nicht, wirken die Operatoren natürlich auch semantisch anders. Nur für Cycle_n wurde zusätzlich noch der Operator k-Opt gewählt, da dieser auf S_n semantisch keinen Sinn macht.

Alle Operatoren, die zum Erzeugen von Individuen mehrere Eltern benötigen, sind Rekombinationsoperatoren. Für die Suchräume $\{0,1\}^n$ und $\{0,\ldots,l-1\}^n$ wählten wir die Standardoperatoren uniformes Crossover, k-Punkt-Crossover, poissonsverteiltes Crossover und Genpool-Crossover. Viele Rekombinationsoperatoren machen aber auf Permutationen wenig Sinn, da durch sie viele Informationen verloren gehen. Trotzdem haben wir uns für die Rekombinationsoperatoren Order Crossover, Partially Mapped Crossover und Maximal Preservative Crossover entschieden, die in dem Referat „Rekombination” (Abschnitt 3.4) beschrieben werden. Zusätzlich wurde ein spezieller Operator für Cycle_n namens Inver-Over implementiert, da er relativ schnell ein TSP optimieren kann.

Als Selektionsoperatoren übernahmen wir aus dem gleichnamigen Referat die Operatoren fitnessproportionale Selektion, Turnier-Selektion, uniforme Selektion und Schnittselektion.

In der Designphase sahen wir nur einen Split-Operator vor, der alle eingehenden Individuen zufällig auf alle seine Ausgänge verteilt. Weitere Split-Operatoren wurden erst in der Implementierungsphase hinzugefügt.

Kapitel 5

Implementierungsphase

Inhalt

5.1 Allgemeines 69
5.1.1 Verteilung der Aufgaben 70
5.1.2 Code Konventionen 70
5.1.3 CVS und andere benutzte Tools 71
5.2 Ablauf 72
5.2.1 Konfigurierbare Objekte 72
5.2.2 Persistenz 73
5.2.3 Nichteinhaltung der Deadline 73
5.3 Testen 74
5.4 Implementierungsdetails 75
5.4.1 Verwendete Bibliotheken 76
5.4.2 Gestaltung der GUI 77
5.4.3 Erweiterungsmöglichkeiten 79
5.4.4 Verwaltung von Modulen und Events 80
5.4.5 Performance-Überlegungen 81
5.4.6 Threads 82
5.4.7 GUI-Updates 83
5.4.8 Headless Mode 84

5.1 Allgemeines

In diesem Abschnitt werden einige allgemeine Punkte beschrieben, wie sie zu Beginn der Implementierungsphase festgelegt wurden, wie Aufgabenverteilung, Konventionen zum Schreiben des Codes und verwendete Tools.

5.1.1 Verteilung der Aufgaben

Vor Beginn der Implementierung wurde eine Einteilung in vier Gruppen vorgenommen. Diese Einteilung folgte der in der Designphase festgelegten Struktur des Programmes, sowohl vom Standpunkt der Funktionalität als auch vom Standpunkt der Einteilung des Source Codes in Packages. Die Gruppen waren:

- Core
- GUI
- Observer
- Rest/Modules

Es wurden den Gruppen die in Abschnitt 2.4 beschriebenen Anzahlen von Teilnehmern zugeordnet. Ein Grund für die Zuweisung von mehr Teilnehmern zur Core-Gruppe als zu allen anderen Gruppen war der Gedanke, den Kern des Programms, also die Ablaufsteuerung, das Event-System und andere zentrale Komponenten, die unbedingt zum Betrieb des Systems notwendig sind, möglichst schnell in einen lauffähigen Zustand zu verwandeln, damit erste Erkenntnisse über die Qualität des Designs und demzufolge über die weitere Durchführbarkeit des Projektes frühzeitig vorliegen würden.

Die Einteilung in diese Gruppen erwies sich als sinnvoll. Zwar wurden die Grenzen und Zuständigkeiten der Gruppen im Verlauf der Implementierung zunehmend aufgelöst, doch zu Beginn sorgte die eindeutige Zuweisung der Zuständigkeiten zu einem relativ raschen Vorankommen bei der Implementierung. Die Notwendigkeit für eine Aufweichung der Gruppeneinteilung ergab sich einerseits aus der schnellen Fertigstellung von Teilen des Cores, wodurch Ressourcen frei wurden. Des Weiteren implementierten beispielsweise Teilnehmer Code aus dem Zuständigkeitsbereich anderer Gruppen, weil sie aufgrund ihrer Spezialisierung effizienter im Schreiben der konkreten Klassen waren.

Des Weiteren ergaben sich gerade in der späteren Implementierungsphase, in der bereits viele Bugs ausgebessert wurden, Situationen, in denen der Entdecker eines Fehlers am besten dazu geeignet war, diesen im Programm zu lokalisieren und zu beheben.

Schließlich wurde es während der Implementierungsphase notwendig, in bestimmten Situationen demjenigen eine Arbeit zuzuweisen, der zum entsprechenden Zeitpunkt Arbeitszeit frei hatte.

5.1.2 Code Konventionen

Um die Lesbarkeit des Codes für alle Teilnehmer zu erhöhen und um ein insgesamt einheitliches Layout des Codes zu erreichen, wurden einige Konventionen beschlossen, die das eben Genannte festlegen.

Im Wesentlichen wurde festgelegt, wie Code eingerückt und formatiert wird, wie javadoc-Kommentare benutzt werden (d. h. welche Tags benutzt werden und wie die Informationen

zu den einzelnen Tags formatiert werden bzw. welche Informationen aufzulisten sind), sowie einige kleinere Dinge. Richtlinie war, dass alle öffentlichen Attribute und Methoden in Klassen sowie die Funktion von Klassen dokumentiert werden sollten.

Die Festlegungen erwiesen sich im Verlauf der Implementierung als nützlich, wenn auch nicht alle Konventionen ständig eingehalten wurden. Insbesondere diejenigen, die sich auf die Kommentierung des Codes beziehen, wurden nicht sehr diszipliniert eingehalten, was dazu führte, dass die Kommentare teils zu wenig detailliert und informativ sind, als dass jeder diese sofort verstehen könnte. Dies führte nicht zu größeren Problemen, weil durch Kommunikation zwischen den einzelnen Entwicklern die meisten offenen Fragen bereits geklärt waren, bevor eine einem Teilnehmer unbekannte Klasse benutzt wurde. Des Weiteren wurden vereinzelt tutorialartige Dokumente verfasst, die recht genau beschreiben, wie bestimmte Klassen funktionieren und anzuwenden sind.

5.1.3 CVS und andere benutzte Tools

Es wurden Dokumentenstandards sowie die Verwendung einiger Tools beschlossen, um eine reibungslose Kommunikation zu ermöglichen und um einige komplexe Probleme und Aufgaben wie die Verwaltung von Bugs zu lösen.

Versionskontrolle

Aus Gründen der Wiederherstellbarkeit vergangener Versionen des Programms und der Regelung von Konflikten beim gleichzeitigen Editieren an Klassen wurde ein Versionskontrollsystem benutzt. Die Entscheidung fiel auf CVS, da einige der Teilnehmer bereits Erfahrungen mit diesem System während des Softwarepraktikums gesammelt hatten. Ein Teilnehmer übernahm die Aufgabe des Aufsetzen des Systems sowie des Anlegens neuer Projektverzeichnisse. Die Benutzung dieser Versionskontrolle erwies sich im Verlaufe der Implementierung als nützlich, da dadurch ohne größere Probleme ein gleichzeitiges Arbeiten an Klassen möglich war.

Des Weiteren erwies sich das System als nützlich, als während der Implementierung ein Plattencrash des Dateiservers eine Installation des Backups von der vorherigen Nacht nötig machte. Durch die lokal auf den Rechnern der Teilnehmer liegenden, aktuellen Versionen des Programms und die versandten Änderungs-emails, aus denen die einzelnen Änderungen und ihre Reihenfolge hervorgingen, konnten die meisten Änderungen des betreffenden Tages wiederhergestellt werden.

Dokumentenerstellung

Für die Erstellung der während des Projektes anfallenden Dokumente wie Tutorials, Handbücher und des Zwischenberichtes wurde LaTeX benutzt. Die Entscheidung für dieses System fiel unter anderem wegen der Verfügbarkeit für alle Systeme, der leichten Erlernbarkeit und der Tatsache, dass einige Teilnehmer bereits vor der Projektgruppe damit gearbeitet hatten. Des Weiteren ist der Arbeitsaufwand zur Erstellung gut aussehender Dokumente gering.

Bugtracking System

Es wurde zu Beginn der Implementierungsphase beschlossen, ein System zur Verwaltung der Bugs des Programms zu benutzen. Die Entscheidung fiel auf das System Mantis. Dieses System war von einem der Teilnehmer bereits verwendet worden. Nach der Anregung, dieses System zu benutzen, erklärte sich ein Teilnehmer bereit, es zu installieren und zu warten.

Die Entscheidung, ein solches System zu benutzen, erwies sich im Verlauf der Implementierung als sehr nützlich, da es einen einfachen Überblick über die vorhandenen Bugs ermöglicht. Es stellt alle Funktionen bereit, die man benötigt um den Status von Bugs (Bearbeitungszustand, Schwere, Reproduzierbarkeit usw.), zusätzliche benötigte Informationen und die Zuweisung von Bugs zu Bearbeitern zu verwalten. Das System wurde während der Implementierung und auch danach intensiv genutzt.

5.2 Ablauf

Die Implementierung ging zu Beginn der Implementierungsphase sehr schnell voran. Bereits nach wenigen Tagen waren große Teile des Core implementiert. Lauffähig war das Programm aber zu diesem Zeitpunkt nicht, weil große Teile der Module noch nicht implementiert waren.

Während der Implementierung ergaben sich ständig neue Anforderungen an das Programm, so dass neue Klassen und Methoden hinzugefügt werden mussten, um die benötigte interne oder externe Funktionalität bereitzustellen. Externe Funktionalität meint hier solche, die vom Benutzer in der GUI sichtbar ist, interne dagegen solche, die innerhalb des Programms benutzt wird, wie z. B. diejenige des Eventsystems.

Beispielhaft sollen einige der aufgetretenen Anforderungen zusammen mit ihren Lösungen besprochen werden.

5.2.1 Konfigurierbare Objekte

Es fiel auf, dass keine Unterstützung dafür vorhanden war, Module des Systems zu konfigurieren. Deshalb wurde ein Interface (`Configurable`) eingefügt, das eine konfigurierbare Klasse implementieren sollte. In diesem Interface existiert eine Methode, die per Konvention ein in der GUI anzeigbares Panel zurückliefert, in dem der Benutzer alle Konfigurationseinstellungen vornehmen kann. Normalerweise enthält das Panel für alle konfigurierbaren Attribute des Objektes den Namen des Attributs zusammen mit dem aktuellen Wert in einer Tabelle. Teil der Konvention ist, dass die Funktionalität, mit der Änderungen an den Einstellungen in das Objekt zurückgeschrieben werden, ebenfalls vorhanden sein muss.

Damit die Implementierung dieser Methode nicht bei jedem Modul viel Arbeit darstellt, wurde eine Klasse eingeführt, die eine Methode besitzt, die zu einem übergebenen Objekt ein solches Panel liefert, das *Standardinspector* genannt wird. Diese automatische Generierung funktioniert mit allen Attributen, die der Konvention der *Properties* folgen. Das sind solche Attribute, für die es get- und set-Methoden gibt, deren Name mit getProperty oder setProperty beginnt. Zusätzlich können Methoden implementiert werden, die dem Benutzer in der GUI Informationen darüber liefern, welche Auswirkungen das Setzen einer Property hat.

5.2.2 Persistenz

Es musste eine Möglichkeit geschaffen werden, die vom Benutzer erstellten Graphen zu speichern und wieder zu laden. Die erste Möglichkeit, die als Lösung in Betracht gezogen wurde, war, die von Java bereitgestellte Fähigkeit zur Serialisierung zu benutzen. Diese erwies sich als nicht sehr gut benutzbar, da Objekte, in deren Klassen wesentliche Veränderungen vorgenommen wurden, nicht mehr ladbar sind.

Deshalb wurde ein eigenes Speicher- und Ladeverfahren entwickelt, das nicht (bzw. nur eingeschränkt) die Java-Serialisierung benutzt und deshalb nicht so anfällig gegen Änderungen in den Klassen ist. Im Folgenden wird das Verfahren beschrieben.

Das Persistenzverfahren

Das Speichern und spätere Laden eines Objektes erfordert, alle Attribute des Objektes abzuspeichern, damit das Objekt später vollständig rekonstruiert werden kann. Solche Attribute können von primitivem Typ sein, was einfach zu handhaben ist, aber es können auch Referenzen auftreten. Insbesondere die Verkettung von Objekten im System durch diese Referenzen muss konsistent wiederherstellbar sein. Es muss also die komplette verkettete Struktur aller Objekte, die von einem zu speichernden Objekt über Referenzen erreichbar sind, abgespeichert werden.

Die Lösung dieses Problems ist eine Art Graphdurchlauf, startend bei einem zu speichernden Objekt, der alle auftretenden Referenzen verfolgt und die erreichten Objekte abspeichert. Dabei ist zu beachten, dass keine bereits gespeicherten Objekte wiederholt gespeichert werden.

Das in FrEAK angewandte Verfahren wird von einem Persistenz-Manager (Klasse `PersistenceManager`) koordiniert. Als Dateiformat wurde XML gewählt, da es gute Bibliotheken gibt, mit denen solche Dateien gelesen, geschrieben und ausgewertet werden können. Die hier benutzte Bibliothek ist jdom.

Einzelne Objekte werden als so genannte XML-Elemente gespeichert. Das bedeutet, dass schließlich eine Textdatei entsteht, in der zwischen XML-Tags Zeichen stehen, die den Zustand eines Objektes angeben. Jedes gespeicherte Objekt erhält eine Identifikationsnummer, anhand derer es eindeutig identifiziert werden kann und anhand derer auch die Referenzen zwischen Objekten wiederhergestellt werden.

Die Umwandlung eines Objektes in ein XML-Element und zurück wird von Persistenz-Handlern erledigt. Ein Problem bei der Bestimmung des für ein Objekt zuständigen Handlers besteht darin, den Typ eines Objektes zu bestimmen. Dies geschieht mit den Reflection-Fähigkeiten von Java. Schwierigkeiten treten hierbei aber dadurch auf, dass die Klasse eines Objekts möglicherweise verschiedene Interfaces implementiert und es nicht mehr einfach festzustellen ist, welcher Handler für das Objekt zuständig ist.

5.2.3 Nichteinhaltung der Deadline

Im Gegensatz zur Analyse- und zur Designphase kam es ab der Implementierungsphase zu Verzögerungen und Nichteinhaltung von Deadlines. Dies resultierte aus verschiedenen Tatsa-

chen.

Während der erst genannten Phasen ließen sich Deadlines besser einhalten, weil es kein hartes Kriterium dafür gab, wann die zu erstellenden Dinge in einem fertigen Zustand waren. Sicher hätte noch länger und detaillierter entworfen werden können, aber bei Erreichen des Endtermins waren Analyse und Design in einem von der Gruppe insgesamt akzeptierten Zustand, trotz einiger Mängel wie zu undetaillierter Beschreibungen einzelner Dinge. Beispielsweise fehlte eine genaue, formale Spezifikation des Verhaltens der Operatoren im Graphen.

Dagegen war es in der Implementierungsphase klar, ob sie als abgeschlossen gelten kann oder nicht. So lange das Programm noch in der Designphase festgelegte Eigenschaften und Fähigkeiten nicht besaß, konnte die Phase nicht als abgeschlossen bezeichnet werden.

Des Weiteren bestand in der Implementierungsphase der Zwang, bestimmte Mängel auszubessern, um Fortschritte zu machen. Sehr viel Zeit wurde mit der Beseitigung von Fehlern verbracht. Schließlich kamen erst in der Implementierungsphase Fehler im Design zu Tage, die dann in einer Erweiterung des Designs, der Ausbesserung von Fehlern oder dem Hinzufügen von weiterer Funktionalität mündeten. Aufgrund all dieser Tatsachen verzögerte sich die Fertigstellung des Programms und führte zur Nichteinhaltung von Terminen, an denen bestimmte Dinge hätten fertig sein sollen.

5.3 Testen

Das Testen des Programms gliederte sich in zwei wesentliche Phasen. Im ursprünglichen Zeitplan waren drei Wochen für die Implementierung und daran anschließend zwei Wochen für Tests und Dokumentation veranschlagt worden. Tatsächlich zog sich die Fertigstellung einiger Programmteile (z. B. des Graph Editors) dann aber doch etwa zwei Wochen über das geplante Ende der Implementierungsphase hinaus. Auf der anderen Seite wurden die Klassentests für viele Klassen aber bereits während der Implementierungsphase fertiggestellt, so dass sich Implementierung und Test anders als geplant deutlich überschnitten.

Während der ersten drei Wochen lag das Hauptziel darin, möglichst schnell einen funktionierenden Programmkern zu erstellen und diesen dann nach und nach um die weitere geplante Funktionalität zu ergänzen. Klassentests der wichtigsten Klassen wurden bereits zu dieser Zeit parallel zur Implementierung vorgenommen. So waren z. B. die meisten Klassen des Core-Paketes bereits mit JUnit getestet, bevor weitere Module hinzugefügt wurden. JUnit ist ein frei verfügbares Framework zur Erstellung von Klassentests. In einem Testtreiber wird hierbei wie üblich ein Testszenario implementiert, in dem dann die Methoden der zu testenden Klasse aufgerufen und das Verhalten des Programms überprüft wird. Hierzu werden in den Testtreibern Anforderungen an den Programmzustand nach einem Methodenaufruf bzw. die Rückgabewerte einzelner Methoden formuliert. Mit Hilfe der von JUnit bereitgestellten `assert()`-Methode wird dann überprüft, ob die gestellten Anforderungen erfüllt sind. JUnit bietet für diese Tests eine graphische Oberfläche und die Möglichkeit, aufgetretene Fehler zu protokollieren. Alle Module, die anschließend implementiert wurden, wurden ebenfalls sofort Klassentests unterzogen, soweit dies bereits möglich war. Zu diesem Zweck wurde innerhalb der FrEAK-Verzeichnisstruktur ein Test-Verzeichnis angelegt, in dem bereits existierende Tei-

le des Programms zusammen mit entsprechenden Testtreibern und Dummy-Klassen abgelegt wurden.

Am Ende der Implementierungsphase lag eine ausführbare Version des Programms vor, so dass Tests nun am eigentlichen Programm durchgeführt werden konnten. Um alle PG-Teilnehmer möglichst schnell über gefundene Bugs informieren zu können, wurde das Bug-Tracking System Mantis installiert. Die Spanne der innerhalb der zwei Wochen entdeckten Fehler reicht von fehlenden sinnvollen Features bis zu ernsthaften Programmabstürzen. Die meisten Bugs, die ernsthaftere Probleme verursachten, konnten innerhalb kurzer Zeit behoben werden. Größeren Aufwand bei der Behebung verursachten vor allem zwei Bugs. Zum einen erwies es sich als problematisch, beim Erzeugen und anschließenden Editieren eines Schedules den Überblick darüber zu behalten, wie die Module des Programms untereinander auf Events registriert waren. Abhilfe schaffte in diesem Fall nur das nachträgliche Einführen eines zentralen Event Controllers. Zum anderen führte zunächst die Benutzung von Views unweigerlich zum Programmabsturz. Die letztendliche Behebung dieser beiden Probleme ist im Abschnitt 5.4 beschrieben.

Aber auch an diversen Kleinigkeiten musste noch gefeilt werden, da diese zum Teil doch sehr ernsthafte Folgen nach sich zogen. So tauchte z. B. das Problem auf, dass einige Views sich in bestimmten Situationen während des Laufes nicht mehr konfigurieren ließen, da schon beim Öffnen des entsprechenden Dialogs das Programm mit einem OutOfMemoryError abstürzte. Wie sich heraus stellte, lag dies an der Art und Weise, wie Konfigurationen in existierende Module geschrieben werden. Im Dialog zur Konfiguration eines Moduls konnte der Benutzer zu diesem Zeitpunkt eine neue Konfiguration vollständig eingeben und diese anschließend übernehmen oder verwerfen. Um zu überprüfen, ob die Eingaben gültige Werte darstellten, mussten diese bereits während der Eingabe in das Modul geschrieben werden. Da aber nicht alle Änderungen wieder rückgängig zu machen waren, konnte diese nicht in das eigentlich zu konfigurierende Objekt geschrieben werden, weil sonst ein Klick auf den Cancel-Button nicht zwangsläufig noch das gewünschte Ergebnis erzielt hätte. Um dieses Problem zu umgehen, wurde also beim Öffnen des Konfigurationsdialogs zunächst grundsätzlich eine Kopie des zu konfigurierenden Objektes erzeugt, die beim Klick auf den OK-Button dann das ursprüngliche Objekt ersetzen sollte. Im Fall der Views bedeutete dies aber, dass auch alle enthaltenen Daten kopiert werden mussten, was bei längeren Läufen eine immense Datenmenge bedeuten kann. Aufgrund der wenigen zur Verfügung stehenden Zeit und dem Aufwand, den es bedeutet hätte, das gesamte Verfahren zum Setzen einer Konfiguration zu überarbeiten, wurde im Plenum beschlossen, das Problem durch Entfernung des Cancel-Buttons zu lösen.

5.4 Implementierungsdetails

In diesem Abschnitt wird auf einige Implementierungsdetails näher eingegangen, dabei werden nur die relevanten Details näher beleuchtet. Relevant heißt in diesem Kontext, dass zum einen die Lösungen von komplizierten Problemen beschrieben werden, die sich im Laufe der Implementierung ergaben. Zum anderen wird auf Dinge eingegangen, die für FrEAK spezifisch sind, also z. B. die Erweiterbarkeit.

5.4.1 Verwendete Bibliotheken

Bei der Implementierung von FrEAK wurden für einige Aufgaben bestehende Bibliotheken eingesetzt. Auf diejenigen, welche besonders wichtig für FrEAK sind, wird in diesem Abschnitt näher eingegangen.

Colt

Für die Simulation evolutionärer Algorithmen ist es notwendig, Zufallsentscheidungen zu treffen. Das ist mit Computern nicht möglich, da diese deterministisch arbeiten. Es gibt jedoch so genannte Pseudozufallszahlengeneratoren, das sind Algorithmen, die aus einem vorgegebenen Startwert (der z. B. mit der Systemzeit initialisiert werden kann) eine Sequenz von Bits erzeugen, die für die Simulation „hinreichend zufällig" ist. Siehe hierzu auch Abschnitt 4.1.2.

Das Entwickeln und Implementieren effizienter und guter Pseudozufallszahlengeneratoren ist eine hochgradig nicht triviale Aufgabe. Es war nicht das Ziel der PG, eigene Generatoren zu entwickeln, deswegen haben wir nach bereits verfügbaren Implementierungen bekannter Generatoren gesucht. Dabei haben wir zahlreiche Implementierungen gefunden, unter anderem die Bibliothek Colt (siehe [18]). Diese wurde am CERN entwickelt und enthält bereits Implementierungen zahlreicher guter Generatoren, insbesondere ist ein Mersenne Twister (siehe [30] und [29]) integriert.

Wir haben uns dafür entschieden, die Colt-Bibliothek zu verwenden, da sie außer den eigentlichen Generatoren auch Klassen zur Verfügung stellt, mit denen man verschiedene Verteilungen realisieren kann, z. B. (negative) Binomialverteilungen und Poisson-Verteilungen. Ferner steht Colt unter einer sehr freien Lizenz, es darf frei benutzt, kopiert, verändert, verteilt und in kommerzielle Produkte integriert werden.

In der Analysephase gab es den Vorschlag, dass der Benutzer den in FrEAK benutzen Pseudozufallszahlengenerator auswählen kann. Dieser wurde vorerst nicht umgesetzt, um den Benutzer nicht mit zu vielen Einstellungsmöglichkeiten zu konfrontieren. Die Architektur ist so gestaltet, dass es kein Problem darstellt, dieses Feature bei Bedarf noch zu ergänzen. Im Nachhinein zeigte sich jedoch, dass die Auswahl verschiedener Pseudozufallszahlengeneratoren nicht benötigt wurde.

JGraph

Ein wesentlicher Bestandteil von FrEAK ist der Operatorgraph, der einen evolutionären Algorithmus modelliert. Da der Benutzer in der GUI den Graphen editieren können soll, bestand die Notwendigkeit, diesen graphisch anzeigen zu können.

Es wurde recherchiert, ob zu diesem Zweck verwendbare Bibliotheken existieren. Die einzige gefundene Bibliothek, die für das Projekt geeignet erschien, war JGraph, da sie frei benutzbar und mächtig genug erschien. Es wurde beschlossen, JGraph zu benutzen.

Diese Entscheidung erwies sich aus mehreren Gründen als problematisch: Zum einen verhält sich die Bibliothek unter einigen Betriebssystemen merkwürdig (siehe auch Abschnitt 5.4.2).

Zum anderen erschwerte die unzureichende und oft veraltete Dokumentation die Anbindung an FrEAK. JGraph hält sich an das bekannte Model-View-Prinzip, das auch in Java Swing benutzt wird. Um einen Graphen mit der JGraph-Bibliothek anzeigen zu können, muss ein Model erstellt werden, dass das Interface `GraphModel` der Bibliothek implementiert. Dieses Interface und auch die von der Bibliothek bereitgestellte Default-Implementierung `DefaultGraphModel` enthalten allerdings sehr viele, teilweise schlecht dokumentierte Methoden.

Zunächst wurde versucht, `GraphModel` direkt vom FrEAK-Operatorgraphen zu implementieren. Zu dem Problem der unzureichenden Dokumentation kam dabei noch, dass in FrEAK die Kanten implizit, nicht aber – wie in der JGraph-Bibliothek – explizit durch Objekte dargestellt werden.

Daher wurde dieser Versuch recht bald aufgegeben und zu Gunsten der folgenden Lösung fallen gelassen: Zur JGraph-View gibt es zwei Model-Objekte, und zwar eines einer eigenen Model-Implementierung von `GraphModel` und ein FrEAK-Operatorgraph-Objekt. Die View-Komponente kennt nur das erste dieser beiden Modelle. Das FrEAK-Modell wird durch einen `GraphModelListener` (ein EventListener, der von der JGraph-Bibliothek bereitgestellt wird) über Änderungen im JGraph-Modell verständigt und mit diesem synchronisiert.

Dieser Ansatz ist anwendbar, da der Graph nur über die GUI geändert werden kann. Es kann höchstens der Fall eintreten, dass zu einem schon bestehenden FrEAK-Modell ein JGraph-Modell erzeugt werden muss. Die selbst geschriebene, `GraphModel` implementierende Klasse stellt eine Methode dafür zur Verfügung, so dass dieser Fall abgedeckt ist.

5.4.2 Gestaltung der GUI

Die GUI-Gruppe begann die Arbeit frühzeitig und implementierte Prototypen der Fenster und Elemente der Oberfläche. Diese waren nicht auf Funktionalität des Programms angewiesen, wodurch sehr schnell ein Eindruck der Bedienung des Programms gewonnen werden konnte. Dadurch konnte getestet werden, ob die in der Analyse- und Designphase entwickelten Konzepte der Bedienung in der Praxis gut funktionieren würden.

Während der Implementierungsphase wurden häufig Mängel in der Bedienbarkeit sichtbar und es stellte sich heraus, dass zusätzliche Funktionalität benötigt wurde. Die Mängel bezogen sich zum Beispiel darauf, dass bestimmte, immer wieder kehrende Operationen nur sehr umständlich ausführbar waren, oder dass immer wieder von Hand ein Layout der GUI durchgeführt werden musste. Diese Dinge mussten während der Implementierung korrigiert bzw. implementiert werden.

JGraph

JGraph als Bibliothek zur Anzeige des Graphen erwies sich abgesehen von den aufgetretenen Problemen bei der Implementierung als für unsere Zwecke ausreichend.

Ein Problem, welches sich mit der Verwendung von JGraph ergab, war, dass die GUI des Programms nicht auf allen Systemen richtig funktionierte. Probleme ergaben sich auf Mac OS,

hier konnten die Editierungsfunktionen des Graphen nicht normal benutzt werden, da die GUI extrem selten aktualisiert wurde, was zum Beispiel beim Verschieben von Knoten dazu führte, dass man nicht mehr richtig erkennen konnte, wohin man einen Knoten verschoben hatte.

Verwendete Tools

Es wurden Hilfsprogramme benutzt, um die GUI zu erzeugen. Die meisten Teilnehmer, die an der Erstellung der Oberfläche beteiligt waren, entschieden sich für die Verwendung von NetBeans als Entwicklungsumgebung, aber auch JBuilder kam zum Einsatz. Ein Grund für die Verwendung von NetBeans war, dass die Trennung von Code für den Aufbau der GUI und deren Funktionalität in NetBeans sauberer ist als in JBuilder. Dadurch konnte die von den meisten Teilnehmern verwendete Entwicklungsumgebung Eclipse für die Funktionalität der GUI verwendet werden, während NetBeans nur zum Editieren des Aussehens der GUI verwendet wurde.

Eclipse sowie NetBeans wurden unter anderem deshalb benutzt, weil diese Entwicklungsplattformen frei verfügbar sind.

Bedienbarkeit der GUI

Ein wesentliches Problem der GUI-Gestaltung war die Tatsache, dass das Programm aufgrund seiner Funktion inhärent eine gewisse Komplexität aufweist. Trotzdem sollte die Bedienung einfach und schnell erlernbar sein, auch wenn potentielle Benutzer eher der Gruppe der erfahrenen Rechneranwender zuzuordnen sind. Deshalb wurde sich schließlich für eine wizard-artige Erstellung eines Schedules entschieden, bei der der Benutzer durch eine Folge von Fenstern navigieren kann, in deren Verlauf alle benötigten Einstellungen vorgenommen werden. Dadurch ist eine korrekte und einfache Konfiguration sichergestellt.

Zunächst wurde der gesamte Wizard in einer einzigen Klasse implementiert. Dieser Ansatz führte zu zwei Problemen, zum einen zu einer unüberschaubar großen Klasse und zum anderen zu einer langen Ladezeit beim Anzeigen des Wizards. Diese Probleme wurden später dadurch gelöst, dass zunächst ein Framework erstellt wurde, in dem Panels des Wizards angezeigt werden konnten. Dann wurden die einzelnen Schritte bei der Erstellung eines Schedules auf verschiedene, einzelne Panels aufgeteilt und diese in das Framework eingefügt. Die einzelnen Klassen besaßen dann eine überschaubare Größe und konnten auch schneller geladen werden.

Ein Problem, welches sich bei der Benutzung des Programms herausstellte, war das Verlorengehen von getätigten Einstellungen beim Zurück-Navigieren durch die Dialoge. Einstellungen wurden zurückgesetzt, wenn aus ihrem Dialog zurück navigiert und dort etwas geändert wurde. Dies war anfangs auf Grund der komplexen Abhängigkeiten zwischen den Einstellungen auch so geplant, stellte sich aber beim Arbeiten mit dem Programm als enervierend heraus.

Erst im zweiten PG-Semester wurde eine umfassende Lösung dieses Problems realisiert: Die Module wurden um Methoden erweitert, die testen, ob sie in einem veränderten Schedule noch lauffähig sind. Zweitens bekam jedes Modul die Möglichkeit, sich in einer zweiten neuen Methode neu zu initialisieren, falls es beispielsweise von Einstellungen anderer Module abhängig ist und sich diese Einstellungen verändert haben.

5.4.3 Erweiterungsmöglichkeiten

Bei der Planung und Implementierung von FrEAK wurde Wert darauf gelegt, dass das Programm leicht und ohne großen Aufwand erweitert werden kann, denn Benutzer, die das Programm ernsthaft einsetzen wollen, werden in jedem Falle neue Fitnessfunktionen implementieren müssen, die die von ihnen untersuchten Probleme widerspiegeln. Ebenfalls nicht untypisch dürfte die Implementierung neuer Operatoren und anderer Module sein.

Die erweiterbaren Programmteile sind in erster Linie die so genannten Module. Das sind Suchräume, Fitnessfunktionen, Mapper, Operatoren, Parametercontroller, Stoppkriterien, Populationsmodelle, Observer und Views. Für jeden der o. g. Modultypen gibt es ein Interface und eine abstrakte Oberklasse. Für die Implementierung eines neuen Moduls reicht es aus, eine neue Klasse zu erstellen, die das entsprechende Interface implementiert. Es ist aber in der Regel deutlich unkomplizierter, einfach eine neue Klasse von der abstrakten Oberklasse erben zu lassen, denn die abstrakten Oberklassen bieten bereits Standard-Implementierungen wichtiger Methoden. Die neuen Klassen müssen nicht im Programm angemeldet werden, sondern werden automatisch vom Module Collector (siehe Abschnitt 5.4.4) verwaltet.

Die meisten Module sind konfigurierbar, d. h. sie haben Parameter, die vom Benutzer in der GUI eingestellt werden können. In Abschnitt 5.2.1 wurde beschrieben, wie die Konfigurierbarkeit von Modulen in das System integriert wurde, und dass es einen Standardinspector gibt, den man sich für ein konfigurierbares Modul konstruieren lassen kann.

Die Generierung des Standardinspectors ist optional, der Benutzer hat stets die Möglichkeit, für spezielle Module eigene Inspectors zum Editieren der Properties bereit zu stellen. Ferner gibt es noch die Möglichkeit, den Standardinspector zu erweitern. Der Standardinspector stellt bereits Möglichkeiten zur Verfügung, häufig vorkommende Typen von Properties (also z. B. Integer, Boolean, etc.) anzuzeigen und zu bearbeiten. Wird ein Modul entworfen, das eine Property hat, die einen anderen Typ hat (z. B. ein Array), so kann der Modulentwickler den Standardinspector leicht so erweitern, dass er auch mit dem neuen Datentyp klar kommt. Die Erweiterung erfolgt Java-typisch durch die Ergänzung eines Renderers und eines Editors für Tabelleneinträge, die beim Standardinspector angemeldet werden können.

Ein Problem, das sich im Laufe der Zeit herausstellte, war der Cancel-Knopf im Standardinspector. Änderungen des Wertes einer Property im Standardinspector werden sofort an das entsprechende Modul weitergereicht, da zum einen eine Bereichsprüfung durchgeführt werden soll, ob der eingegebene Wert gültig ist, und zum anderen, weil das Setzen einer Property auch andere Properties beeinflussen kann. Dieses Vorgehen führte jedoch zu Problemen. Wird bei einem Observer, der eine History einstellbarer Länge verwalten kann, die Länge dieser History verkürzt, so wird diese Verkürzung sofort durchgeführt und ein Zurücknehmen beim Klick auf Cancel ist nicht möglich. Da es prinzipiell keine Lösung gibt, die auf das Kopieren von Modul und/oder seiner Konfiguration verzichten kann, wurde dieses Problem dadurch gelöst, den Cancel-Knopf zu entfernen.

5.4.4 Verwaltung von Modulen und Events

Module

Aus der flexiblen Erweiterbarkeit von FrEAK ergibt sich die Notwendigkeit einer dynamischen Verwaltung der integrierten Module. Module, die von Benutzern des Programms hinzugefügt werden können, sind u. a. Suchräume, Fitnessfunktionen, Mapper, Operatoren, Stoppkriterien, Populationsmodelle, Observer und Views. Es wurde darauf geachtet, dass das Hinzufügen so einfach wie möglich ist, d. h. insbesondere auch, dass es nicht notwendig ist, neue Module in einer Konfigurationsdatei o. ä. anzumelden.

Statt dessen wurde ein so genannter „Module Collector" implementiert. Dieser hat die Aufgabe dynamisch zur Laufzeit die Verzeichnisstruktur nach Modulen zu durchsuchen. Implementiert man nun ein neues Modul vom Typ `xy`, so muss man lediglich eine Klasse schreiben, die das zu `xy` gehörende Interface implementiert, und diese im Package `freak.module.xy` oder in einem seiner Unterpackages ablegen.

Bei der Implementierung des Module Collectors kam die Reflection-API von Java zum Einsatz. Mit dieser ist es möglich, .class-Dateien auf vorhandene Methoden, implementierte Interfaces etc. zu überprüfen. Dadurch wurde es möglich, die Module dynamisch zu verwalten.

Dabei übernimmt der Module Collector auch die Aufgabe eines Filters. Wird eine Liste aller verfügbaren Module eines Typs erstellt, so werden die weiteren bereits vorhandenen Einstellungen beachtet, und es werden nur diejenigen Module der Liste hinzugefügt, die kompatibel mit den weiteren Einstellungen sind. Abhängigkeiten treten zum Beispiel bei der Liste der Stoppkriterien auf. Wurde eine Fitnessfunktion ausgewählt, die ihren optimalen Fitnesswert nicht kennt, so werden diverse Stoppkriterien nicht in die Liste aufgenommen.

Module mancher Typen werden in Packages gespeichert, die nach Suchräumen unterteilt sind. Das Filtern funktioniert bei solchen Modulen zweistufig. Aus der Liste aller Module eines solchen Typs werden zunächst diejenigen entfernt, die nicht kompatibel mit dem aktuellen Genotyp- bzw. Phänotypsuchraum sind. Die Kompatibilität mit einem Suchraum erkennt man entweder an dem Verzeichnis, in dem das Modul liegt, oder daran, dass es ein Interface implementiert, mit dessen Hilfe man die Kompatibilität zu einem Suchraum testen kann. Danach befinden sich nur noch Module in der Liste, die mit dem gewählten Suchraum kompatibel sind.

Um die weiteren Abhängigkeiten in den Griff zu bekommen, wird die Methode `testSchedule` des erzeugten Moduls aufgerufen. Module, die nicht kompatibel mit der bisher gewählten Konfiguration des Schedules sind, müssen dann in dieser Methode eine `UnsupportedEnvironmentException` werfen. Solche Module werden dann auch aus der Liste gelöscht.

Events

Die Kommunikation zwischen Modulen und anderen Teilen des Programms verläuft in FrEAK zu einem großen Teil über Events. Für eine ausführliche Beschreibung siehe Abschnitt 4.2.1.

Nachdem die Implementierungsphase etwas fortgeschritten war, wurde die Notwendigkeit einer globalen Verwaltung der Zuordnung von Event-Quellen und Empfängern deutlich. Es

trat nämlich u. a. folgendes Problem auf: Das gewählte Populationsmodell sendet nach jeder erzeugten Generation ein Event aus, auf das sich z. B. Observer registrieren können, die nach jeder Generation die aktuelle Population brauchen. Das Populationsmodell kann jedoch nachträglich noch geändert werden, in diesem Fall müssen auch die festgelegten Event-Zuordnungen aktualisiert werden und das alte Populationsmodell muss als Event-Quelle durch das neue ersetzt werden. Ähnliche Probleme traten auch bei anderen Modulen auf.

Deswegen wurde ein Event-Controller entworfen, der die festgelegten Event-Zuordnungen global speichert. D. h. jedes Modul, das ein Event von einer bestimmten Quelle haben möchte (z. B. dem aktuellen Populationsmodell), muss sich beim Event-Controller darauf anmelden. Wird dann das Populationsmodell geändert, so kann der Event-Controller alle Event-Zuordnungen auf den neuesten Stand bringen.

Des Weiteren bietet der Event-Controller Unterstützung bei der Verdrahtung so genannter nicht statischer Events. Das sind Event-Zuordnungen, bei denen die Event-Quelle aus Sicht des Event-Listeners variabel ist. Bestes Beispiel ist ein Observer, der alle Individuen anzeigt, die über eine bestimmte Kante des Operatorgraphen laufen. Die möglichen Event-Quellen sind dann die Outports, die zu den Kanten gehören. Die Outports melden sich bei ihrer Erzeugung beim Event-Controller als mögliche Event-Quellen an. Der Event-Controller liefert dann auf Wunsch eine Liste möglicher Event-Quellen für einen gewissen Event-Typ zurück. Diese Liste kann von der GUI angezeigt werden, und der Benutzer kann dann eine der möglichen Event-Quellen auswählen.

5.4.5 Performance-Überlegungen

In diesem Abschnitt sollen einige der Überlegungen dargelegt werden, die gemacht wurden, um die Effizienz des Programms auf ein zufrieden stellendes Niveau zu heben.

Fitnessfunktionen

Bei der Implementierung von FrEAK wurde auch Wert auf Effizienz gelegt. Die Auswertung von Fitnessfunktionen sollte so schnell wie möglich sein, damit die Simulation nicht ausgebremst wird. Das Konzept der Fitnessfunktionen ist in FrEAK sehr allgemein gehalten, d. h. die Fitness eines Individuums hängt im Allgemeinen nicht nur von seinem Genotyp ab, sondern auch von seinen Tags (siehe Abschnitt 4.2.1) und den anderen Individuen in der Population. Die Fitness darf sogar von beliebigen Informationen aus dem Schedule abhängen, z. B. von der Nummer der aktuellen Generation. Damit kann FrEAK auch eingesetzt werden, um beispielsweise Simulated Annealing zu implementieren.

Viele Fitnessfunktionen machen von diesen Möglichkeiten jedoch keinen Gebrauch und der Fitnesswert, den sie einem Individuum zuordnen, hängt ausschließlich von dem Genotyp des Individuums ab. Solche Fitnessfunktionen bezeichnen wir als statisch. Der Vorteil dieser Fitnessfunktionen ist, dass ein einmal ausgerechneter Fitnesswert für ein Individuum zwischengespeichert werden kann und nicht jedes Mal neu ausgerechnet werden muss, weil es keine Rolle spielt, ob sich die Umgebung geändert hat.

Es wurden zwei abstrakte Oberklassen eingeführt, von denen statische monokriterielle bzw. multikriterielle Fitnessfunktionen erben können. Diese enthalten bereits Methoden, die automatisch dafür sorgen, dass Fitnesswerte zwischengespeichert werden.

Graphdurchlauf

Das Erzeugen einer neuen Generation aus der aktuellen, also die Auswertung des Operator Graphen, ist die zentrale Aufgabe während der Simulation. Abgesehen von der Auswertung des Laufes durch Observer und die Darstellung dieser Auswertungen in der GUI, wird die komplette Rechenzeit während der Simulation für den Graphdurchlauf verwendet.

Aus diesem Grund wurde bei der Implementierung des Graphdurchlaufes besonders auf Effizienz geachtet. Eine Beschreibung, wie der Durchlauf organisiert ist, findet man in Abschnitt 4.2.1.

5.4.6 Threads

Auf Grund einiger besonderer Eigenschaften von FrEAK ergab sich besonderer Entwurfs- und Implementierungsaufwand bezüglich der Thread-Struktur des Programms.

Eine der grundlegenden Zielsetzungen von FrEAK war, dass während der rechenaufwändigen und lange dauernden Simulation der Algorithmen regelmäßig GUI-Updates durchgeführt werden sollten, um die Ergebnisse der Berechnungen dem Benutzer anzuzeigen. Des Weiteren sollte während einer Simulation das Programm möglichst gut bedienbar bleiben, das heißt lange Wartezeiten zwischen Eingaben in der GUI und der Reaktion des Programms auf diese Eingaben sollten vermieden werden.

Aus diesem Grund war die einfache Methode, das Programm durch nur einen Thread zu organisieren, nicht akzeptabel. Dies hätte bedeutet, das Programm komplett im Event Dispatching Thread der GUI laufen zu lassen. Dem zu Folge wäre zum Beispiel nach Betätigen des Start-Buttons die Simulation gestartet worden und bis zum Ende der Simulation wären weder GUI-Updates durchgeführt worden, noch hätte das Programm während des Laufes einer Simulation auf Benutzereingaben reagiert.

Alte Architektur

Das Programm wurde in zwei Threads aufgeteilt. Ein Thread ist der Event-Dispatching-Thread, der von Java organisiert wird. Der andere Thread ist der Simulationsthread, in dem die Simulation der Algorithmen abläuft. Dies schafft die Möglichkeit, die oben angeführten Probleme zu lösen, da die Berechnungen der Simulation die GUI nicht blockieren.

Es musste eine Kontrollmöglichkeit eingeführt werden, mit der die Simulation vom Benutzer gesteuert, das heißt zum Beispiel angehalten und gestartet werden kann. Dies führte zu den in der Multithreadprogrammierung allgegenwärtigen Synchronisationsproblemen.

In einem ersten Ansatz wurde der Simulationssthread beim Klick auf Start gestartet und dann jede Benutzerinteraktion mit Ausnahme des Unterbrechens der Simulation unterbunden.

Diese Unterbrechung wurde mit Hilfe des Interrupted-Bit des Simulationsthread realisiert. Dadurch war es leicht möglich, eine ausgelaufene Simulation von einer unterbrochenen zu unterscheiden.

Diese Architektur funktionierte zwar, erlaubte aber nicht ohne ständiges Unterbrechen der Simulation den Replaymodus bequem zu nutzen. Insbesondere folgte die Simulation nicht dem Generationsslider. Zwischen dem ersten und dem zweiten Semester wurde die Threadarchitektur daher neu entworfen.

Neue Architektur

Der Simulationsthread wird nun nicht gestartet und gestoppt, sondern ist als dauerhaft laufender Thread realisiert. Die RunControl, die diesen Thread enthält, wird von der GUI durch eine Reihe von Anfragen gesteuert. Die Anfragen sind in synchrone und asynchrone Anfragen unterteilt.

Synchrone Anfragen können nur gestellt werden, wenn die Simulation unterbrochen ist. Eine synchrone Anfrage ist beispielsweise das Laden eines neuen Schedules. Synchrone Anfragen werden innerhalb des Event-Dispatching-Thread ausgeführt.

Asynchrone Anfragen sind jederzeit möglich und werden dem Simulationsthread über eine Queue zugestellt. Es gibt dazu Start- und Stoppanfragen sowie Anfragen zum Anspringen einer bestimmten Position im Replay. Das Abschicken einer asynchronen Anfrage wechselt immer in den asynchronen Modus. Der asynchrone Modus wird vom Simulationsthread nach Ablauf des Schedules oder Empfang einer Stoppanfrage an einem definierten Punkt in der Simulation wieder verlassen. Der Event-Dispatching-Thread wird darüber in der Event-Queue informiert und kann danach wieder synchrone Anfragen stellen.

5.4.7 GUI-Updates

Durch die Trennung wird das Zeichnen der GUI asynchron zu eventuellen Änderungen ihres Inhaltes durchgeführt, weil diese beiden Aufgaben auf zwei verschiedene Threads fallen. Eine Änderung der angezeigten Daten während deren Neuzeichnung kann zu Abstürzen führen, weswegen ein Synchronisationsmechanismus benutzt werden musste.

Eines der schwer wiegendsten Probleme, die bereits die ersten Testläufe des Programms zu Tage förderten, waren die Updates der graphischen Ausgabe der Views. Die erste Implementierung dieses Programmteils hielt sich exakt an die von Sun vorgeschlagene Vorgehensweise. Die Swing-Dokumentation besagt, dass Updates von bereits „realisierten“ (d. h. sichtbaren oder anzeigbaren) GUI-Komponenten nur im Event-Dispatching-Thread ausgeführt werden dürfen. Dies stellt kein Problem dar, so lange Updates von Standardevents (wie z. B. Mausklicks innerhalb der GUI) ausgelöst werden, da diese ohnehin im Event-Dispatching-Thread behandelt werden. Im Fall der Views stammen die Events aber aus dem Simulationsthread des Programms, so dass die resultierenden GUI-Updates von Hand in den Event-Dispatching-Thread verschoben werden mussten. Java stellt zu diesem Zweck die Methode `invokeLater()` zur Verfügung, die die notwendigen Methodenaufrufe (in einem Runnable-Objekt verpackt) in die Event Queue dieses Threads schreibt.

Beim Ausführen des Programms führte dieses Vorgehen zu Abstürzen mit einem `OutOfMemoryError`. Auf Grund der großen Anzahl von zu Stande kommenden `repaint()`-Aufrufen wurde die Event Queue offensichtlich überhaupt nicht mehr abgearbeitet. In einem zweiten Versuch wurde statt der Methode `invokeLater()` die Methode `invokeAndWait()` verwendet. Diese Methode platziert analog zur vorherigen Methodenaufrufe im Event-Dispatching-Thread, blockiert den aufrufenden Thread dann allerdings so lange, bis das Repaint ausgeführt wurde. Dies hatte zur Folge, dass alle GUI-Updates nun zwar unmittelbar erfolgten, allerdings nicht, ohne den Großteil der Rechenzeit hierfür in Anspruch zu nehmen.

Es war also eine Lösung gefragt, die die Anzahl der GUI-Updates auf das unbedingt notwendige Maß reduzieren würde. Um dies zu erreichen, wurde das Programm gegen Ende des ersten Semesters um einen UpdateManager und jede View um ein entsprechendes DataModel ergänzt. Views schreiben ihre Daten nicht unmittelbar in die GUI, sondern in ihr DataModel und markieren dieses als geändert. Dieser Vorgang hat noch kein GUI-Update zur Folge. Der UpdateManager läuft als unabhängiger Thread und kontrolliert in festen Zeitabständen alle DataModels im Programm. Wird ein DataModel als verändert erkannt, so wird ein entsprechendes GUI-Update ausgelöst. Die Zugriffe der Views und des UpdateManagers auf die DataModels sind synchronisiert, so dass stets ein konsistenter Datensatz ausgegeben wird. Wird während eines Laufes in den Suspended-Modus gewechselt, so werden alle Daten aus den DataModels in die GUI geschrieben. Dies stellt sicher, dass im Suspended-Mode stets die aktuellsten Daten angezeigt werden.

Dieses Verfahren wurde anschließend auf weitere Teile der GUI ausgeweitet, um unter anderem den Generationszähler weniger rechenzeitaufwändig zu gestalten. Leider wird die Kommunikation vom Simulationssthread zum Event-Dispatching-Thread noch nicht vollständig über den UpdateManager ausgeführt. Insbesondere beim Erzeugen von neuen Panels für Views während der Simulation muss der Simulationsthread für einen Moment angehalten werden, bis die Erzeugung abgeschlossen ist, da sonst Updates von Panels deren eigentliche Erzeugung überholen könnten.

5.4.8 Headless Mode

Der nachträglich eingefügte Headless Mode konnte in die Architektur sehr einfach eingefügt werden. Im Headless Mode wird das Hautpfenster durch eine einfache Klasse ersetzt, die zu Beginn ein Schedule lädt und dann eine Startanfrage stellt. Nachdem die Simulation ausgelaufen ist, wird diese Klasse vom Simulationsthread benachrichtigt und kann das Programm beenden.

Die Klasse erzeugt, im Gegensatz zum Hauptfenster, keine Panels für Views. Dadurch werden alle Updates der DataModels wie geplant durchgeführt, auch der UpdateManager läuft, aber die DataModels schreiben ihren Daten nie in eine Swing-Komponente.

Leider wurde dabei ein interner Fehler im Event-Dispatching-Thread sichtbar. Swing benötigt nicht nur einen X-Server, wenn es Fenster darstellen soll, sondern schon, wenn ein Event in die Event Queue gestellt wird. Erstaunlicherweise läuft das Programm, wenn die auftretende Exception schlicht ignoriert wird, jedoch trotzdem wie gewünscht.

Kapitel 6

Dokumentationsphase

Da man FrEAK sowohl nur zur Simulation bestehender EAs benutzen als es auch um weitere Module (Operatoren, Suchräume, Observer, etc.) erweitern kann, haben wir uns dazu entschlossen, dies auch in den Dokumentationen deutlich werden zu lassen.

Zum einen gibt es das Benutzerhandbuch (User's Guide). Dieses beschreibt die Benutzung des Programms und geht dabei teilweise auch etwas darüber hinaus, um grundlegende Prinzipien einzuführen und zu erläutern. Die Einleitung gibt eine Einführung in FrEAK, die Realisierung von Algorithmen mittels Operatorgraphen und die Bedienung des Programms.

In Kapitel 2 wird das Anlegen eines Schedules ausführlich besprochen, einschließlich der genauen Beschreibung jedes einzelnen Schrittes. Es werden ebenfalls die Bedeutungen der einzelnen Module (Graph, Observer, Views, etc.) erläutert.

Kapitel 3 widmet sich dann ganz der Bedienung von FrEAK während eines Simulationslaufes. Hierbei kommt es besonders auf die Replay-Funktion und die Navigation durch die bereits simulierten Läufe an, da diese ein wichtiges Feature von FrEAK darstellt.

In Kapitel 4 wird das Prinzip der Operator-Graphen und deren Erstellung erklärt. Da es sich hierbei um eine wenig verbreitete und nicht völlig kanonische Darstellung von EAs handelt, bedarf es hier auch einiger Erläuterungen.

Die Entwicklung eigener Module wird im Entwicklerhandbuch (Module Developer's Guide) erklärt. In ihm wird der Entwickler an die Hand genommen und ihm Schritt für Schritt erklärt, wie man die einzelnen Module implementieren kann und worauf dabei zu achten ist. Nach einer kurzen Einleitung erklärt Kapitel 2 die Grundkonzepte von FrEAK, mit deren Hilfe man die folgenden Kapitel leichter verstehen kann.

Die weiteren Kapitel 3–11 erläutern dem Leser dann die Erzeugung von eigenen Suchräumen, Fitnessfunktionen, Mappern, Operatoren, Parametersteuerungen, Populationsmodelle, Stoppkriterien, Observer und Views. Wir haben uns bewusst für diese Reihenfolge entschieden, da dies den logischen Aufbau des Programm widerspiegelt. Allem zu Grunde liegen die Suchräume, auf denen dann die Fitnessfunktionen basieren, etc. Außerdem ist das die Reihenfolge, die der Entwickler schon von der Benutzung des Programms durch den Schedule-Editor gewohnt ist.

In Kapitel 12 werden dann weitergehende Konzepte erklärt und besprochen, die für den fortgeschrittenen Entwickler interessant sind, die jedoch nicht für das Verständnis der vorigen Kapitel notwendig sind.

Die beiden oben genannten Dokumente sind natürlich statisch, d. h. als Anleitung gedacht, also in gedruckter Form zu lesen. Da Benutzer aber auch während der Ausführung des Programms spezifische Informationen gewinnen können sollen, haben wir uns dazu entschieden, eine zusätzliche HTML-Hilfe bereit zu stellen. Prinzipiell handelt es sich dabei momentan nur um die umgewandelte Version des Benutzerhandbuchs, allerdings ist es mit Hilfe der Hyperlinks wesentlich schneller möglich, die gesuchten Informationen zu finden. Auch das Aussehen ist dem typischen Layout eines „Online-Textes" angepasst und lässt sich somit besser am Bildschirm lesen.

Die bereits in der Implementierungsphase entstandenen „How-To"s (z. B. über die Implementierung der Parametersteuerung) sind in die Dokumente eingeflossen bzw. wurden teilweise übernommen.

Kapitel 7

Zweite Seminarphase

7.1 Minimale Spannbäume

Vortragender: Kai Plociennik

Literatur: [33]

Es wurde ein Vortrag gehalten, der das Verhalten randomisierter Suchheuristiken bzw. evolutionärer Algorithmen auf dem Problem der Berechnung minimaler Spannbäume zum Thema hatte.

7.1.1 Einleitung/Motivation

Es wurde das Verhalten zweier typischer randomisierter Suchheuristiken (RSH), randomisierte lokale Suche (RLS) und des (1+1) EA, auf dem bekannten, effizient lösbaren und gut verstandenen Optimierungsproblem der Berechnung von minimalen Spannbäumen (MST, minimal spanning trees) untersucht. Die Motivation für diesen Ansatz war, etwas über das Verhalten und die Analyse von RSH zu lernen.

Interessant waren hier beispielsweise die folgenden beiden Punkte. Das MST Problem lässt sich mit Greedy-Algorithmen effizient lösen, d. h. in gewissem Sinne „lokal“. Lokalität ist hier aber eher zeitlich zu verstehen, denn die (Greedy-) Algorithmen von Kruskal und Prim benutzen Informationen über viele Knoten und Kanten, die im ganzen Graphen verteilt sind. Es ist auch nur anhand von einer Kante und der Nachbarschaft der Endknoten nicht entscheidbar, ob die Kante zu einem MST gehört. Unsere Algorithmen dagegen arbeiten eher lokal im eigentlichen Sinne: RLS mutiert nur ein oder zwei Bits pro Schritt (der Algorithmus musste für dieses Problem aufgrund des Festhängens in lokalen Optima bei ausschließlicher Benutzung von 1-Bit-Mutationen angepasst werden) und auch beim (1+1) EA sind „große“ Mutationen eher selten. Interessant war also, wie gut diese „lokalen“ Algorithmen mit dem Problem „fertig werden“.

Des Weiteren ist es so, dass viele effizient lösbare Optimierungsprobleme (so auch das MST) NP-harte Verallgemeinerungen haben. Beim MST ist das z. B. die Berechnung von gradbeschränkten Spannbäumen oder eine multikriterielle Version, in der eine Kante mehrere

Gewichte hat und nach einer pareto-optimalen Lösung gefragt wird. Bei solchen Problemen können evolutionäre Algorithmen konkurrenzfähig sein. Um die Optimierungszeit auf solchen Problemen aber abschätzen zu können, ist es nötig (zumindest hilfreich), das Verhalten auf den einfachen Varianten verstanden zu haben.

7.1.2 Ergebnisse

Um die Laufzeiten der RSH besser einschätzen zu können, wurde wiederholt, dass der Algorithmus von Kruskal eine Laufzeit von $O((m+n)\log n)$ besitzt.

Das MST Problem wurde für das Szenario RSH so kodiert, dass Punkte des Suchraums Bitstrings sind und Kantenauswahlen eines Graphen beschreiben. Es wurden zwei verschiedene Fitnessfunktionen analysiert. Die eine bewertet Suchpunkte nach den folgenden drei Kriterien: Zahl der entstehenden Zusammenhangskomponenten, Zahl der ausgewählten Kanten und schließlich Gewicht ausgewählter Kanten. Dabei dominiert ein Kriterium in dieser Reihenfolge alle nachfolgenden. In der zweiten Funktion fehlt die Bewertung der Zahl ausgewählter Kanten. Interessant für die Analyse war, ob dieses „weniger an Informationen" zu einer (asymptotisch) anderen Laufzeit führt.

Die Analyse war insofern interessant, als dass sie auf eine neue Art und Weise funktioniert. Sie basiert darauf, lokale Verbesserungen, d. h. „gute" Mutationen zu betrachten, die den Funktionswert eines Individuums senken. Dabei wurde die erwartete Verbesserung des Funktionswertes bestimmt, genauer gesagt der erwartete Faktor der Verbesserung. Hierbei beobachtete man einen Tradeoff zwischen Wahrscheinlichkeit einer Mutation und dem Grad an Verbesserung. Große Verbesserungen haben kleine Wahrscheinlichkeit und kleine Verbesserungen große Wahrscheinlichkeit. Im Durchschnitt ergibt sich eine genügend große erwartete Verbesserung pro guter Mutation, d. h. ein genügend kleiner Faktor. Nach einer genügend großen Anzahl guter Mutationen ergibt sich durch die fortgesetzte Verbesserung, d. h. die fortgesetzte Multiplikation mit dem erhaltenen Faktor eine so große Verbesserung, dass man das Optimum mit hoher Wahrscheinlichkeit gefunden hat.

Als Ergebnis erhielt man für beide Funktionen und beide Algorithmen eine obere Schranke von $O(m^2(\log n + \log w_{\max}))$ für die im Worst-Case (über alle Eingaben) erwartete Laufzeit bis zum Erreichen des Optimums. Für die erste Funktion wurde für den Fall polynomiell beschränkter Kantengewichte auch ein Worst-Case-Beispiel angegeben, auf dem beide Algorithmen diese Laufzeit erreichen. Die Analyse ist also scharf.

7.2 Experimentbewertung

Vortragende: Andrea Schweer

Literatur: [27]

7.2.1 Einleitung

Dieses Referat stellte einen Beitrag von Sean Luke und Liviu Panait zur Konferenz GECCO 2002 vor, in dem die Autoren die gängige Vorgehensweise, evolutionäre Algorithmen (vor

allem in der Forschung zum genetischen Programmieren) zu vergleichen, kritisieren und eigene Verbesserungsvorschläge vorstellen.

7.2.2 Gängige Vorgehensweise

Wenn die Qualität zweier evolutionärer Algorithmen miteinander verglichen werden soll, unterscheidet sich die gängige Vorgehensweise danach, ob dies im Kontext der Forschung zum genetischen Programmieren geschieht oder nicht.

Außerhalb dieses Kontextes wird als Kriterium für den Vergleich die *durchschnittliche* beste Fitness auf einer großen Stichprobe verwendet. Begleitet wird dieser Wert meist von einem Test auf Verschiedenheit der Mittelwerte.

Im Kontext des genetischen Programmierens werden stattdessen häufig Maße benutzt, die messen, wie viele Individuen ausgewertet werden müssen, bis mit gegebener Wahrscheinlichkeit die *ideale* Lösung gefunden worden ist.

7.2.3 Kritik an der Vorgehensweise

Die Autoren des referierten Beitrags haben drei große Kritikpunkte an der vorgestellten Vorgehensweise im Kontext des genetischen Programmierens. Diese Punkte ergeben sich aus statistischen Gesichtspunkten, der hinter der Vorgehensweise stehenden Motivation sowie aus der mangelnden Korrelation mit anderen Maßen.

Statistische Gesichtspunkte. Die verwendeten Maße basieren auf Punkt-Stichproben. Für diese Art von Stichproben gibt es keinen Test auf signifikanten Unterschied, so dass Werte dieses Maßes für zwei verschiedene Algorithmen nicht verglichen werden können. Außerdem sind einige der Maße abhängig zwischen den Generationen. Üblicherweise werden für Aussagen über verschiedene Generationen aber keine getrennten, unabhängigen Stichproben verwendet, was deshalb eigentlich nötig wäre. Dazu kommt, dass kleine Änderungen in der Anzahl idealer Lösungen große Änderungen in den Werten einiger der Maße verursachen.

Motivation. Im genetischen Programmieren wird üblicherweise versucht, Algorithmen zu entwickeln, die *möglichst gute* Fitness erreichen. In diesem Licht erscheint es seltsam, als Gütekriterium die Anzahl der *idealen* Lösungen zu verwenden.
Die Autoren erklären sich das mit einer vorherrschenden philosophischen Täuschung, beim genetischen Programmieren ginge es darum, korrekte Programme zu finden. Ein Optimum oft genug zu erreichen, dass eine statistische Aussage darüber möglich ist, gehe aber nur bei „Spielzeugproblemen“, nicht bei echten Optimierungsaufgaben.

Korrelation mit anderen Maßen. Die Intuition könnte vielleicht zu folgender Annahme verleiten: Ein System, das bei der Suche nach Lösungen mit hoher Fitness gut ist, sollte auch mehr ideale Lösungen finden.
Die Autoren haben allerdings Experimente durchgeführt, die diese Annahme widerlegen. Diese Experimente haben drei verschiedene Szenarien untersucht. Da bei der

Evolution von Syntaxbäumen die Anzahl der hintereinander verschachtelten Crossover-Operatoren entscheidend ist, wurden für jedes Szenario 500 unabhängige Läufe für jeweils 1 bis 10 Operatoren durchgeführt. Mit steigender Anzahl der Operatoren fällt die durchschnittliche beste Fitness; die Anzahl der gefundenen idealen Lösungen verhält sich für jedes Szenario unterschiedlich, ebenso wie die Werte der darauf basierenden Maße.

7.2.4 Lösungsvorschlag

Um die beschriebenen Probleme mit den üblicherweise im Kontext des genetischen Programmierens verwendeten Maßen zu umgehen, empfehlen die Autoren folgendes: Die Anzahl gefundener idealer Lösungen sollte nur dann verwendet werden, wenn das Finden einer perfekten Lösung wichtig ist, und auch dann nicht als einziges Maß. Für andere Fälle haben die Autoren ein neues Maß definiert: die erwartete maximale beste Fitness eines Laufs. Dieses Maß hat aber ebenfalls statistische Probleme und sollte daher nur *zusätzlich* zur Angabe der besten Fitness eines Laufs verwendet werden.

7.3 Das Ising-Modell auf dem Ring

Vortragender: Dirk Sudholt

Literatur: [11]

7.3.1 Einleitung

In diesem Referat wurde das Ising-Modell vorgestellt und das Verhalten randomisierter und evolutionärer Algorithmen auf dem Ising-Modell auf speziellen Graphen untersucht.

Ising-Modelle haben ihren Ursprung in der statistischen Physik und beschreiben das Verhalten von magnetischen Teilchen mit zwei Ausrichtungen (spins), die die Tendenz besitzen, ihre Ausrichtung ihren Nachbarn anzugleichen. Solche Teilchen und ihre Nachbarschaftsbeziehungen lassen sich durch Graphen modellieren. Aus Sicht der Informatik ist das Ising-Modell daher ein umgedrehtes Färbbarkeitsproblem mit dem Ziel, möglichst viele adjazente Knoten gleich zu färben.

Wir betrachten die so genannte Ising-Funktion auf Graphen, die für jede Kante den Wert 1 zur Fitness beiträgt, wenn die beiden Endpunkte gleich gefärbt sind, und 0 sonst. Die Graphen beschränken wir auf Ringe, das sind Graphen, die nur aus einem Kreis der Länge $n = |V|$ bestehen. Suchpunkte sind Knotenfärbungen $x = (x_1, \ldots, x_n)$, wobei wir uns auf die Betrachtung zweier Farben 0 und 1 beschränken. Damit gilt $x \in \{0, 1\}^n$.

Obwohl die Optima, alle einheitlichen Färbungen der Knoten, das Problem trivial erscheinen lassen, ist die Optimierung nicht unproblematisch: die Ising-Funktion ist spin-flip-symmetrisch, d. h. es gilt $f(x) = f(\overline{x})$ für die Fitness eines Individuums x und seines bitweisen Komplements $\overline{x}$. Auf Ringen gilt diese Symmetrie auch für bestimmte Teile des Genotyps, so dass die Gefahr besteht, dass sich Teile des Genotyps in unterschiedliche Richtungen entwickeln und letztendlich komplexe Variationen nötig sind, um das globale Optimum zu erreichen.

7.3.2 Die Analyse mutationsbasierter Algorithmen

Nach dieser allgemeinen Einführung in die Problematik wurden verschiedene Algorithmen vorgeführt und analysiert. Der einfachste Algorithmus ist die (1+1) randomisierte lokale Suche (RLS), bei der zur Erzeugung des Nachkommens ein uniform zufällig gewähltes Bit geflippt wird. Bei der Optimierung entstehen Plateaus von Suchpunkten mit i Blöcken, wobei wir mit Blöcken maximale zusammenhängende Folgen gleich gefärbter Knoten bezeichnen. Blöcke können ihre Länge verändern, wenn ein Bit an einem der beiden Enden geflippt wird. Damit wird die Zeit, bis ein Block verschwindet und die Fitness dadurch erhöht wird, durch einen Random Walk bestimmt. Eine einfache, aber genaue Analyse des Random Walks für einen ausgewählten kleinen Block B liefert eine obere Schranke $O(n^3)$ für die erwartete Optimierungszeit.

Der (1+1) EA lässt sich nur sehr aufwändig direkt analysieren, daher wurde ein Vergleich des (1+1) EA mit der Analyse der (1+1) RLS durchgeführt. In verschiedenen Zwischenschritten wurden dabei Algorithmen betrachtet, die „zwischen" dem (1+1) EA und der (1+1) RLS liegen, in dem Sinne, dass sie Komponenten von beiden Algorithmen beinhalten. Im Unterschied zur (1+1) RLS kann der (1+1) EA mehrere Bits auf einmal flippen. Da sich jedoch relativ schnell Blöcke bilden und Bitflips innerhalb von Blöcken die Fitness verringern, sind Mehrbitmutationen meist wenig hilfreich. Zudem ist es in manchen Situationen wahrscheinlicher, einen Block um k Bits zu verlängern, als ihn um k Bits zu verkürzen.
Allerdings zeigt sich, dass große Änderungen der Blocklänge unwahrscheinlich sind und die Wahrscheinlichkeiten nur bei sehr kurzen und sehr langen Blöcken asymmetrisch sind, so dass sich die Laufzeit hier nur um einen zusätzlichen Faktor $o(1)$ vergrößert. Insgesamt unterscheiden sich (1+1) EA und (1+1) RLS nur um höchstens konstante Terme, so dass auch für den (1+1) EA die obere Schranke $O(n^3)$ für die erwartete Optimierungszeit gilt.

7.3.3 Die Analyse paralleler mutationsbasierter Algorithmen

Auf dem Weg zu genetischen Algorithmen wurde ein paralleler mutationsbasierter Algorithmus untersucht, die (1+λ) RLS. Die Schwierigkeit bei Populationen ist die Selektion des neuen Individuums: da viele Nachkommen die gleiche Fitness haben, erhält der Algorithmus keine Informationen darüber, welches Individuum selektiert werden soll. Daher selektiert der Algorithmus eines der fittesten Nachkommen zufällig oder übernimmt den Elter, falls alle Nachkommen schlechter sind.

Wir betrachten wie bei der Analyse der (1+1) RLS einen kleinen Block B und schätzen die erwartete Zeit ab, bis B verschwindet. Ein günstiger Fall wäre daher, dass eine Mutation einen Nachkommen x erzeugt, in dem die Länge von B verändert wurde, da dieses Ereignis den Random Walk von B voran treibt. Allerdings schlägt sich diese bessere Pseudo-Fitness i. A. nicht in der Fitness des Nachkommens nieder, so dass wir darauf hoffen müssen, dass sich x gegenüber den Nachkommen mit gleicher Fitness durchsetzt und als neuer Elter selektiert wird.
Es zeigt sich jedoch, dass für $\lambda := n$ die Zahl nötiger Generationen durch $O(n^2 \log n)$ beschränkt ist und sich die Zahl an Fitnessauswertungen mit $O(n^3 \log n)$ nur um einen Faktor $\log n$ von der (1+1) RLS unterscheidet.
Mit einer besseren Wahl von λ lässt sich diese Zahl auf ebenfalls $O(n^3)$ senken.

Diese Ergebnisse für die (1+λ) RLS lassen sich direkt auf den (1+λ) EA übertragen.

7.3.4 Die Analyse genetischer Algorithmen

Als nächstes wurde ein spezieller, angepasster genetischer Algorithmus vorgestellt: der so genannte (1+1) GIGA (Gene Invariant Genetic Algorithm) verwaltet eine Population, die stets aus einem Individuum x und seinem bitweisen Komplement $\overline{x}$ besteht und die damit stets maximale Diversität aufweist. In einem Schritt werden x und $\overline{x}$ miteinander rekombiniert und die Nachkommen y und $\overline{y}$ übernommen, falls sie nicht schlechter sind als ihre Eltern. Da x und $\overline{x}$ redundant sind, kann die Rekombination von x mit $\overline{x}$ auch als Mutation von x angesehen werden.

Der (1+1) GIGA wurde daraufhin für Two-Point-Crossover und Ein-Punkt-Crossover untersucht. Mit Zwei-Punkt-Crossover wird eine Fitnessverbesserung erreicht, wenn beide Kreuzungspunkte auf Trennstellen, das sind Zwischenräume zwischen zwei Blöcken, treffen. Die Bits innerhalb der beiden Kreuzungspunkte werden geflippt, wodurch beide Trennstellen verschwinden und sich die Fitness um 2 erhöht. Über alle Fitnessschichten von Suchpunkten mit i Blöcken hinweg erhalten wir eine erwartete Optimierungszeit von $O(n^2)$.

Für Ein-Punkt-Crossover erhalten wir die gleiche asymptotische Laufzeit. Da in jedem Schritt die Bits rechts des einzigen Kreuzungspunktes geflippt werden, wird in jedem Schritt die Kante $\{x_n, x_1\}$ umgefärbt. Ein-Punkt-Crossover kann daher als spezielles Zwei-Punkt-Crossover mit festem Kreuzungspunkt an $\{x_n, x_1\}$ angesehen werden. Um die Fitness zu erhöhen, muss der Algorithmus i. A. nacheinander zwei Schritte ausführen, bei denen der Kreuzungspunkt eine Trennstelle trifft: im ersten solchen Schritt wird $\{x_n, x_1\}$ zweifarbig und im zweiten Schritt werden die Trennstelle an $\{x_n, x_1\}$ und die Trennstelle am Kreuzungspunkt entfernt, wodurch sich die Fitness um 2 erhöht. Auch hier erhalten wir die asymptotische Laufzeit von $O(n^2)$.

Da der (1+1) GIGA ein an Spin-Flip-Symmetrie angepasster Algorithmus ist und die Diversität „künstlich“ aufrecht erhalten wird, wurde zum Schluss die Frage besprochen, ob eine ausreichende Diversitätserhaltung auch mit Standardmethoden wie Fitness Sharing möglich ist.
Beim Fitness Sharing werden nahe beieinander liegende Suchpunkte bestraft und die Fitness eines Individuums wird an der Population gemessen. Ein einfacher genetischer Algorithmus mit Populationsgröße 2 wurde vorgestellt, der mit Fitness Sharing arbeitet. Erwähnenswert dabei ist der Selektionsoperator, der aus der Menge P von Eltern und Nachkommen eine zweielementige Teilmenge P' wählt, die die Fitness der Population maximiert. Die Fitness der Individuen wird dabei also anhand von P' gemessen und nicht anhand der Gesamtpopulation aus Eltern und Nachkommen. Die Laufzeit für diesen Schritt ist hier konstant, da P nur 4 Individuen enthält.

In der Analyse betrachten wir die Fitness der Population $P = \{x, y\}$, da diese aufgrund der Selektion monoton steigend ist. Um die Fitness der Population zu erhöhen, kann der Algorithmus die Zahl der Blöcke in x und y verringern oder den Hammingabstand von x und y vergrößern. Mit der Buchhaltermethode kann die Laufzeit auf verschiedene Typen von Populationen verteilt werden: Falls der Hammingabstand nicht maximal ist, gilt $x \neq \overline{y}$

und die Fitness kann durch eine gute 1-Bit-Mutation um eine Konstante erhöht werden. Ansonsten gilt $x = \overline{y}$ und wir sind in der gleichen Situation wie der (1+1) GIGA. Für beide Fälle erhalten wir erwartete Zeiten von jeweils $O(n^2)$.

Das überraschende Ergebnis lautet also, dass sich auch mit Fitness Sharing eine erwartete Optimierungszeit von $O(n^2)$ erreichen lässt.

7.4 Das Ising-Modell auf dem quadratischen Torus

Vortragender: Matthias Englert

Literatur: [10]

Die Analyse des (1+1) EA auf der Ising-Funktion des quadratischen Torus mit zwei Farben gestaltet sich anders als die anderen Analysen dieses Semesters. Statt der Suche nach unteren und oberen Schranken für die Laufzeit bis zum Erreichen des Optimums, wird eine obere Schranke der Zeit bis zum Erreichen eines stabilen Zustands ermittelt. Das kann entweder das globale Optimum sein oder ein Zustand, in dem kein 1-Bit-Flip mehr akzeptiert werden kann.

Beim Ising-Modell auf dem quadratischen Torus ergeben sich, im Gegensatz zum Ising-Modell auf dem Ring, lokale Optima, aus denen der (1+1) EA nur durch das gleichzeitige Flippen sehr vieler Bits wieder entkommen kann. Experimente zeigen, dass diese stabilen Zustände nicht selten erreicht werden.

7.4.1 Klassifizierung von Färbungen

Definition. *Ein Graph heißt quadratischer Torus, wenn* $V = \{0, \cdots, \sqrt{n}-1\} \times \{0, \cdots, \sqrt{n}-1\}$ *ist und zwei verschiedene Knoten* (x_1, y_1) *und* (x_2, y_2) *genau dann durch eine ungerichtete Kante verbunden werden, wenn gilt:*

$$\begin{aligned}&(x_1 = x_2 \quad \vee \quad x_2 = (x_1+1) \mod \sqrt{n} \quad \vee \quad x_1 = (x_2+1) \mod \sqrt{n}) \quad \wedge \\ &(y_1 = y_2 \quad \vee \quad y_2 = (y_1+1) \mod \sqrt{n} \quad \vee \quad y_1 = (y_2+1) \mod \sqrt{n})\end{aligned}$$

Man kann sich den Torus als quadratisches Gitter von Knoten vorstellen, in dem jeder Knoten mit seinen acht Nachbarn über Kanten verbunden ist. Dabei werden Knoten des oberen Randes mit Knoten des unteren Randes und Knoten des rechten Randes mit Knoten des linken Randes verbunden.

Man kann stabile Färbungen des quadratischen Torus charakterisieren, d. h. Färbungen, in denen keine 1-Bit-Mutationen mehr akzeptiert werden (tatsächlich müssen meist sehr viele Bits zugleich geflippt werden, um aus stabilen Zuständen, die nicht dem Optimum entsprechen, zu entkommen). Hierzu werden einige Begriffe eingeführt, die hier nur kurz skizziert werden sollen:

Ein *verbessernder Block* ist eine horizontale, vertikale oder diagonale Reihe zusammenhängender Knoten gleicher Farbe. Ein verbessernder Block ist dergestalt, dass das Flippen aller seiner Knoten zu einer Verbesserung der Fitness führt. Außerdem kann ein verbessernder Block durch 1-Bit-Flips verkürzt und u. U. auch verlängert werden.

Eine *Blüte* entsteht, wenn sich eine Kante, die zwei 1-gefärbte Knoten verbindet, mit einer Kante die zwei 0-gefärbte Knoten verbindet, kreuzt.

Ringe sind einfarbige Gebiete, die sich ein- oder mehrmals um den Torus wickeln.

Zustände, die nur aus Ringen bestehen, welche sich einmal um den Torus wickeln und die außerdem keine verbessernden Blöcke enthalten, sind stabil (vorausgesetzt alle Ringe haben eine Dicke von wenigstens zwei).

7.4.2 Analyse

Folgender Satz wird in der Analyse gezeigt:

Satz. *Sei G ein quadratischer Torus. In $O(n^3)$ Generationen findet der (1+1) EA auf* $\textsc{Ising}_G$ *eine Färbung, die optimal oder stabil ist.*

Gibt es in einer Färbung verbessernde Blöcke und/oder Blüten, erreicht man nach $O(n^3)$ Generationen eine Färbung ohne verbessernde Blöcke und ohne Blüten. Eine solche Färbung besteht nur noch aus Ringen.

Es genügt nachzuweisen, dass sich nach $O(n^2)$ Generationen die Fitness erhöht, denn nach spätestens $4n$ Fitnesserhöhungen wäre das Optimum erreicht.

Eine hilfreiche Beobachtung ist, dass es genügt, sich auf die Betrachtung von Mutationen zu beschränken, die eine zusammenhängende Menge von gleich gefärbten Knoten flippen.

Im Falle der Blüten kann man nun eine einfache (wenn auch nicht unbedingt kurze) Fallunterscheidung vornehmen, um die Fitnessverbesserung nach $O(n)$ Generationen zu zeigen. Im Fall der verbessernden Blöcke zeigt man, dass man mit einer ausreichend kleinen Fehlerwahrscheinlichkeit entweder die Fitness verbessert oder in keiner der $O(n^2)$ Generationen den verbessernden Block zerstört. Analog zur Analyse des Ising-Modells auf einem Ring kann man dann mit einem Random-Walk-Argument nachweisen, dass der verbessernde Block in $O(n^2)$ Generationen verschwindet und sich damit auch die Fitness erhöht.

Damit ist der Nachweis erbracht, dass man nach $O(n^3)$ Generationen eine Färbung erhält, die nur noch aus Ringen besteht. Diese Ringe sind allerdings nicht stabil, wenn sie sich mehr als einmal horizontal und/oder vertikal um den Torus wickeln. Gibt es i solche Ringe, bricht einer nach $O(n^{5/2}i^{-2})$ Generationen auf und es entsteht ein verbessernder Block, der dann, wie bereits gesehen, mit ausreichender Wahrscheinlichkeit nach $O(n^2)$ Generationen zu einer Fitnesserhöhung führt.

Man kann weiter zeigen, dass man höchstens $4\sqrt{n}$-mal aus einer Situation mit i instabilen Ringen entkommen muss und die Anzahl der Ringe außerdem durch $\sqrt{n}$ beschränkt ist. Summieren wir all dies auf, erhalten wir hier die gewünschte obere Schranke von $\sum_{i=1}^{\sqrt{n}} 4\sqrt{n} \cdot O(n^{5/2}i^{-2}) = O(n^3)$ Generationen für die Zeit, in der wir das Optimum oder einen stabilen Zustand erreichen.

7.4.3 Approximation

Eine einfache Beobachtung ist, dass die Fitness einer stabilen Färbung hauptsächlich von der Anzahl der Ringe abhängt. Gibt es viele Ringe, ist die Fitness schlecht, andererseits muss es dann aber dünne Ringe geben und dünne Ringe werden verhältnismäßig schnell aufgebrochen. Gibt es nur wenige Ringe, können diese zwar dicker sein, die Fitness liegt aber auch sehr nahe beim Optimum. Also sind Färbungen mit schlechter Fitness zwar keine gute Lösung, sie lassen sich aber auch relativ schnell verbessern.

7.5 Parametrisierung

Vortragender: Oliver Heering

Literatur: [17], [34] und [21]

7.5.1 Einleitung

Dieses Referat behandelte das Thema der unterschiedlichen Parametrisierungsmöglichkeiten von evolutionären Algorithmen. In drei Teilen wurden sowohl theoretische als auch experimentelle Analysen auf einigen Beispielproblemen durchgeführt und ausgewertet.

Evolutionäre Algorithmen sind von einer Vielzahl von Parametern abhängig, beispielsweise Populationsgröße, Mutationswahrscheinlichkeit oder Selektionsdruck. Der erste Teil des Referates befasst sich mit der Aussage, eine Mutationswahrscheinlichkeit von $1/n$ sei in der Regel eine gute Wahl und beweist anhand eines theoretischen Beispiels, dass dies in manchen Fällen durchaus sogar sehr ungeeignet sein kann. Des Weiteren wird erläutert, wie man dem Problem der unbekannten optimalen Mutationswahrscheinlichkeit begegnen kann.

Der zweite und dritte Teil des Vortrags erläutert, wie unterschiedliche Wissenschaftler an realen Problemen auf experimentellem Wege Hypothesen bezüglich optimaler Parameterwerte überprüfen und zu welchen Schlussfolgerungen sie dadurch kommen.

7.5.2 Teil 1: Über die Wahl der Mutationswahrscheinlichkeit des (1+1) EA

Zunächst wird anhand einiger trivialer Beispiele gezeigt, dass die oft empfohlene Mutationswahrscheinlichkeit von $p = 1/n$ schlechter sein kann als eine zufällige Suche mit Mutationswahrscheinlichkeit $p = 1/2$ ist. Dieses Phänomen tritt bei Funktionen auf, bei denen es beispielsweise keinen (oder einen falschen) Hinweis auf die Richtung, in der das Optimum liegt, gibt (Needle), oder aber bei Funktionen, bei denen eine Falle zu überwinden ist, um zum Optimum zu gelangen (Trap). In beiden Fällen können nur größere Sprünge zum Optimum führen.

Aus dieser Beobachtung wird eine nicht-triviale Funktion PathToJump konstruiert, die aus analytischer Sicht die Individuen zunächst auf einem Pfad entlang laufen lässt, um anschließend in die Menge der Optima zu springen. Die Analyse des (1+1) EA auf dieser Funktion

erfolgt in mehreren Schritten entsprechend der Teilmengen, die bei der Definition der Funktion benutzt werden. Sie dient dem Beweis der Behauptung, dass der Algorithmus im Erwartungswert superpolynomiell viele Schritte benötigt, falls die Mutationswahrscheinlichkeit $p(n)$ stark von $(\log n)/n$ abweicht. Im Weiteren wird noch eine obere Schranke von $O(n^{2,361})$ für die erwartete Laufzeit und eine untere Schranke von $1 - e^{-\Omega(n)}$ für die Wahrscheinlichkeit, dass der (1+1) EA für eine geeignete Mutationswahrscheinlichkeit $p(n)$ ein Optimum in $O(n^{3+c} \ln^{-1} n + n^{1+c-\log c - \log \ln 2})$ Schritten findet, gezeigt.

Aber auch bei nicht bekannter guter oder optimaler Mutationswahrscheinlichkeit lässt sich eine Modifikation des (1+1) EA angeben, die bessere Laufzeiten garantiert. Die Rede ist hierbei vom dynamischen (1+1) EA, dessen Mutationswahrscheinlichkeit zyklisch Werte zwischen $1/n$ und $1/2$ durchläuft. Für den dynamischen (1+1) EA wird eine obere Schranke für die erwartete Anzahl von Schritten von $O(n^2 \log n)$ gezeigt und die untere Schranke für die Wahrscheinlichkeit, dass der (1+1) EA ein Optimum in $O(n^{3+c} \ln^{-1} n + n^{1+c-\log c - \log \ln 2})$ Schritten findet, liegt beim dynamischen (1+1) EA ebenfalls bei $1 - e^{-\Omega(n)}$.

7.5.3 Teil 2: Geltungsbereich und Einschränkungen der $1/n$ Heuristik

Der zweite Teil des Referats beschäftigte sich mit einer Arbeit, die sich ebenfalls um die Behauptung, $1/n$ sei eine ausreichend gute Mutationswahrscheinlichkeit, dreht. Diesmal wurden allerdings zwei konkrete Probleme, eines real aus dem Maschinenbau (WingBox) und eines aus der Kombinatorik (Multiple Knapsack) stammend, anhand von Experimenten und Statistiken untersucht. Dazu wurden jeweils aus 50 Läufen mit unterschiedlicher Mutationswahrscheinlichkeit der Mittelwert und die Standardabweichung gebildet und miteinander verglichen. Heraus kam, dass in de facto allen Testreihen und auf beiden Problemen die Mutationswahrscheinlichkeit von $1/n$ die beste Wahl darstellte.

Daraufhin variierte man einige andere Parameter des zugrunde liegenden genetischen Algorithmus und untersuchte, ob sich dadurch andere Mutationswahrscheinlichkeiten als günstiger erwiesen. Die variierten Parameter waren zum einen der Selektionsdruck, realisiert durch eine Turnierselektion mit Turniergrößen 2 und 4, und zum anderen die Populationsgröße. Man fand heraus, dass die Mutationswahrscheinlichkeit von $1/n$ bis auf Ausnahmen stets eine gute Wahl war. Ungünstig soll sich diese Wahl bei folgenden drei Szenarien auswirken: schwacher Selektionsdruck, extrem starker Selektionsdruck oder sehr kleine Population.

Als Schlussfolgerung wurde gesagt, dass diese Untersuchung die Behauptung unterstütze, $1/n$ sei stets eine gute Wahl der Mutationswahrscheinlichkeit. Man beachte hierbei, dass Teil 1 des Referats ein Gegenbeispiel gezeigt hat.

7.5.4 Teil 3: Erforschung des Parameterraums eines genetischen Algorithmus für das Training eines neuronalen Netzes

Der dritte Teil des Referates behandelt ebenfalls ein real existierende Problem, nämlich das Trainieren eines neuronalen Netzes mit Hilfe eines genetischen Algorithmus. Das Netz wird auf die bekannten Probleme 4 Bit Parity (4BP) und 5 Bit Parity (5BP) angewandt. Dabei repräsentieren die Synapsengewichte des Netzwerkes ein Individuum. Die Fitness eines solchen

Individuums wiederum wird berechnet, indem gezählt wird, wie oft die Ausgabe des Netzwerks mit den Synapsengewichten des Individuums bezüglich der soeben genannten Probleme korrekt ist. Der genetische Algorithmus verwendet One-Point Crossover, Standard-Mutation und rangbasierte Selektion. Die variierten Parameter sind hierbei die Populationsgröße ϕ, der Ersetzungsanteil ρ der Selektion und die Mutationswahrscheinlichkeit μ. Ein Lauf des GA wird wie folgt bewertet: Der Lauf wird terminiert nach einer bestimmten Anzahl Fitnessauswertungen. Falls der Mittelwert der Fitnesswerte des besten Individuums der letzten fünf Generation über einem bestimmten Threshold-Wert liegt, gilt der Lauf als „erfolgreich“. Von einer gewissen Anzahl N_E von Läufen werden die erfolgreichen Läufe N_S gezählt. Somit stellt der Quotient $p = N_E/N_S$ die Erfolgswahrscheinlichkeit dar. Für die Experimente wird nun der Yield-Wert $Y = p \cdot 100$ betrachtet.

Im ersten Experiment wurde der Einfluss unterschiedlicher Populationsgrößen ϕ und Mutationswahrscheinlichkeiten μ auf Y untersucht. Die so bestimmten Messwerte wurden durch mathematische Kurven approximiert und man kam zu dem Schluss, dass bei konstantem Ersetzungsanteil von $\rho = 20\%$ und starker Abhängigkeit von Rekombination eine Populationsgröße von 15 und Mutationsraten um die 1% die besten Ergebnisse erzielt.

Das zweite Experiment betrachtete den Einfluss des Ersetzungsanteils und der Mutationsrate auf Y. Die Population war festgelegt auf 100 Individuen. Das Resultat war, dass die Performance des Algorithmus mit zunehmendem Ersetzungsanteil und dadurch reduziertem Einfluss der Rekombination auf die Population steigt. In diesem Fall müssen allerdings auch höhere Mutationsraten benutzt werden.

7.6 Stoppkriterien

Vortragender: Michael Leifhelm

Literatur: [12], [26], [19] und [6]

7.6.1 Einleitung

Bei schwierigen Problemen finden evolutionäre Algorithmen in einem Lauf selten ein globales Optimum. In der Regel ist es nicht einmal möglich zu sagen, ob ein globales Optimum überhaupt schon entdeckt wurde. Aus diesem Grund werden Neustartstrategien verwendet.

Da ein Lauf i. d. R. selten ein Optimum findet, wird in der Praxis ein Algorithmus periodisch mit anderen Zufallszahlen neu gestartet. Um den Zeitpunkt zu bestimmen, zu dem ein Algorithmus neu gestartet werden soll, werden Stoppkriterien eingesetzt. Diese starten einen Lauf neu, wenn sie z. B. auf einer Metrik die Stagnation der Population erkennen, wenn der Algorithmus über einen längeren Zeitraum keinen Fortschritt mehr zeigt oder wenn einfach ein bestimmter Grenzwert (z. B. bei der Fitness, Zeit oder Generationenanzahl) überschritten wurde.

Es gibt eine Hierarchie der Neustartstrategien. Die *statischen Strategien* sind eine Untermenge der *dynamischen Strategien* und diese eine Untermenge der *adaptiven Strategien*. Außerdem

gibt es noch *selbst-adaptive Strategien*, die eine andere Untermenge der *adaptiven Strategien* darstellen.

Dieses Referat beschäftigt sich nun mit drei Papern. Das erste von Sean Luke [26] versucht eine optimale Strategie mit statischen Stoppkriterium zu finden, das zweite von Erick Cantú-Paz und David E. Goldberg [6] versucht zu belegen, dass i. d. R. ein Lauf mit größtmöglicher Population besser ist als mehrere Läufe mit kleineren Populationen. Im letzten Paper von Thomas Jansen [19] werden zwei dynamische Neustartstrategien untersucht.

7.6.2 Mehrere kurze Läufe schlagen lange Läufe

In dem Paper *When Short Runs Beat Long Runs* [26] stellt Sean Luke Möglichkeiten vor, um zu bestimmen, ob ein Schedule (Ablaufplan) besser ist als ein anderer. Besonders beschäftigte er sich mit der Frage, ob es nun sinnvoll ist, einen einzelnen langen Lauf mit n Generationen durchzuführen oder stattdessen besser m Läufe mit $\frac{n}{m}$ Generationen.

Um zwei Schedules miteinander vergleichen zu können, muss der Erwartungswert der besten Fitness aus m Läufen bekannt sein. Sean Luke unterscheidet dabei zwei Fälle. Im kontinuierlichen Fall muss die komplette Dichtefunktion der besten Fitness von allen möglichen Populationen für verschiedene Generationenanzahlen bekannt sein. Da dies leider praktisch nicht möglich ist, analysiert er zusätzlich den diskreten Fall: Wenn man viele Läufe bei einer gegebenen Länge durchführt, kann man das erwartete Ergebnis bei nur n durchgeführten Läufen dieser Länge abschätzen und so bestimmen, ob ein Schedule besser ist als ein anderer. Dies geht allerdings von der Annahme aus, dass alle Läufe, die durchgeführt werden, *exakt repräsentativ* für alle Populationen aller Läufe dieser Länge sind.

Um nun statistisch unabhängige Läufe mit unterschiedlichen Längen zu bekommen, muss man diese für jede untersuchte Länge separat durchführen. Da dies den Rechenaufwand stark erhöhen kann, beschränkt sich Luke auf Lauflängen von Zweierpotenzen bis zur maximalen Lauflänge. Zwei statistische Probleme bleiben allerdings: Es fehlt ein Test auf Abweichung vom Mittelwert, da keiner für diese Art der Ordnungsstatistik bekannt ist und es gibt einen erhöhten Alpha-Fehler, da Daten von einem Schedule wiederholt mit mehreren anderen Schedules verglichen werden.

Sean Luke führte nun Experimente basierend auf seiner Analyse durch. Dazu wählte er die drei verschiedene Probleme *Symbolic Regression*, *Artificial Ant* und *Even-10 Parity* aus. Die Ergebnisse von *Symbolic Regression* und *Artificial Ant* zeigen, dass es bei diesen Problemen keinen Sinn macht, Läufe länger als eine bestimmte Zahl von Generationen (16 für Symbolic Regression und zwischen 32 und 64 für Artificial Ant) laufen zu lassen, da mehrere Läufe mit kürzerer Länge mindestens genauso gut sind, wenn nicht besser. Nur bei *Even-10 Parity* seien lange Läufe vielen kurzen überlegen.

7.6.3 Große Läufe schlagen mehrere kleine Läufe

In *Are Multiple Runs of Genetic Algorithms Better than One?* [6] beschäftigten sich Erick Cantú-Paz und David E. Goldberg mit der Frage, ob mehrere unabhängige Läufe mit kleinen

Populationen bessere Ergebnisse oder schneller akzeptable Lösungen finden als ein einzelner Lauf mit einer großen Population. Sie untersuchten dies auf *additiv-separablen Funktionen*.

Unter der Begrenzung einer festen Anzahl von Funktionsauswertungen sehen die Autoren zwei Möglichkeiten, um die erwartete Qualität zu maximieren. Entweder alle Auswertungen in einen Lauf stecken mit *größtmöglicher* Population oder mehrere unabhängige Läufe mit kleinen Populationen starten. Bei letztgenannter ist die erwartete Qualität pro Lauf zwar niedriger, aber es gibt mehr Möglichkeiten, eine gute Lösung zu finden.

Ihre Experimente beziehen sich auf sehr einfache Funktionen, wie z. B. OneMax mit Dimension 25. Dazu verwenden sie paarweise Turnierselektion ohne Ersetzung, Ein-Punkt-Crossover mit Wahrscheinlichkeit 1 und schließen Mutation aus. Als Populationsgrößen wählen sie Werte zwischen 2 und 50 und führen 1 bis 8 Läufe je Populationsgröße durch. Die Ergebnisse ihrer Experimente zeigen, dass in allen Fällen ein einzelner großer Lauf mehreren kleinen Läufen überlegen ist.

7.6.4 Zwei einfache dynamische Neustartstrategien

Thomas Jansen analysiert in seinem Paper *On the Analysis of Dynamic Restart Strategies for Evolutionary Algorithms* [19] zwei einfache dynamische Neustartstrategien für den (1+1) EA auf drei künstlichen Beispielfunktionen.

Als Neustartstrategien wählt er eine additive und eine multiplikative Variante. Bei der Additiven darf jeder Lauf um eine festgelegte Anzahl von Generationen länger laufen als der Lauf zuvor. Bei der Multiplikativen darf jeder Lauf doppelt so lange laufen wie der Lauf zuvor.

Die künstlichen Beispielfunktionen gliedern sich in eine mit einem relativ leicht zu erreichenden globalen Optimum, eine, bei dem die Wahrscheinlichkeit, das globale Optimum direkt zu finden und nicht „hängen zu bleiben“, bei 1/2 liegt und eine Beispielfunktionen ähnlich der zweiten, allerdings mit polynomiell gegen Null konvergierender Chance, das globale Optimum direkt zu finden.

Ohne und mit additiver und multiplikativer Neustartstrategie hat die erste Beispielfunktion eine polynomielle erwartete Optimierungszeit.

Ohne Neustartstrategie benötigt die zweite in ca. der Hälfte aller Läufe exponentielle Zeit, um das Optimum zu finden. Mit additiver und multiplikativer Neustartstrategie hat sie wie die erste Funktion nur noch eine polynomielle erwartete Optimierungszeit. Hierbei sei anzumerken, dass es bei Verwendung der additiven und multiplikativen Neustartstrategie keinen Unterschied beim Erwartungswert macht, ob die Erfolgswahrscheinlichkeit gegen 1 oder eine andere positive Konstante konvergiert.

Bei der dritten künstlichen Beispielfunktionen ohne Neustartstrategie konvergiert die Wahrscheinlichkeit, das Optimum in polynomieller Zeit zu finden, polynomiell gegen Null. Mit additiver Strategie wird die erwartete Optimierungszeit polynomiell, mit multiplikativer allerdings exponentiell, da man polynomiell viele Läufe benötigt, deren Länge exponentiell anwächst. Geht die Erfolgswahrscheinlichkeit gegen Null, so ist die multiplikative Neustartstrategie der additiven deutlich unterlegen.

7.7 Evolutionäre Algorithmen und Pseudozufallszahlengeneratoren

Vortragender: Patrick Briest

Im Rahmen des Vortrags werden zwei Artikel vorgestellt, die sich mit dem Einfluss von Pseudozufallszahlengeneratoren (PRNGs) auf die Performance evolutionärer Algorithmen befassen [28] [5]. Die Ergebnisse dieser Artikel finden sich im zweiten und dritten Teil des Vortrags wieder. Der erste Teil bildet eine Einführung in die theoretische Analyse des Simple Genetic Algorithm (SGA) nach Michael D. Vose [45], dessen Ergebnisse im Rahmen des ersten vorgestellten Artikels als Ausgangspunkt experimenteller Untersuchungen verwendet werden.

7.7.1 Michael D. Vose - The Simple Genetic Algorithm

Ziel dieses Abschnittes ist es, das Verhalten des SGA in Form einer Markoff-Kette zu beschreiben und Aussagen über das erwartete Verhalten zu treffen. Wir betrachten den Suchraum Ω und eine Population $P \subseteq \Omega$, wobei $|\Omega| = n$ gelte. Dann können wir P als Element des n-dimensionalen Simplex $\Lambda_n = \{(x_1, \ldots, x_n) \mid \forall\, 1 \leq j \leq n, 1^T x = 1, x_j \geq 0\}$ auffassen. Wir identifizieren P mit einem Vektor $p \in \Lambda_n$, indem wir p_j als Anteil des j-ten Elementes von Ω an P interpretieren. Formal müssen wir hier natürlich auch die Populationsgröße berücksichtigen. Bei Populationsgröße r beschreibt $\frac{1}{r} X_n^r := \frac{1}{r}\{(x_1, \ldots, x_n) \mid \forall\, 1 \leq j \leq n, 1^T x = r, x_j \in N_0\}$ die Menge der zulässigen Populationen, wobei $\frac{1}{r} X_n^r \subset \Lambda_n$.

Deterministische Suchheuristiken lassen sich offenbar als Transition $\tau : \Lambda_n \to \Lambda_n$ auf diesen Vektoren beschreiben. Im Fall von randomisierten Suchheuristiken ergibt sich τ als Kombination aus einer deterministischen Heuristik $G : \Lambda_n \to \Lambda_n$ und einer randomisierten Komponente. Formal wenden wir zunächst G auf eine Population p an. Das Ergebnis $G(p)$ interpretieren wir dann nicht als Population, sondern als Wahrscheinlichkeitsverteilung über Ω. Die randomisierte Komponente entspricht dann einem Sampling gemäß der Verteilung $G(p)$. Im Folgenden betrachten wir solche randomisierte Suchheuristiken.

Wir sind an der Wahrscheinlichkeit interessiert, mit der bei einem Generationsübergang die Population q aus der Population p entsteht. Eine einfache Rechnung liefert:

$$P(q = \tau(p)) = \prod_{j=1}^{n} \binom{r - rq_1 - \cdots - rq_{j-1}}{rq_j} (G(p)_j)^{rq_j} = \cdots = r! \prod_{j=1}^{n} \frac{(G(p)_j)^{rq_j}}{(rq_j)!}$$

Was können wir nun aber über die erwartete Nachfolgepopulation von p aussagen? Durch eine etwas trickreichere Rechnung und unter Ausnutzung des Multinomialtheorems erhält man

$$E(\tau(p)) = G(p),$$

was sicherlich auch der Intuition entspricht.

Die Wahrscheinlichkeit $P(q = \tau(p))$ lässt sich mit Hilfe der Stirlingschen Formel approximieren:

$$P(q = \tau(p)) = \exp\left(-r \sum_{j=1}^{n} q_j \log \frac{q_j}{G(p)_j} - \sum_{j=1}^{n} \left(\log \sqrt{2\pi r q_j} + \frac{1}{12 r q_j + \Phi(r q_j)} \right) + O(\log r) \right)$$

Der Ausdruck $\sum_{j=1}^{n} q_j \log \frac{q_j}{G(p)_j}$ wird als *Vose Discrepancy* bezeichnet. Eine genauere Untersuchung der Vose Discrepancy zeigt, dass diese ein Maß für den Abstand zwischen q und $E(\tau(p)) = G(p)$ darstellt. Es sei erwähnt, dass eine Berechnung der Vose Discrepancy natürlich möglich ist, allerdings eine in der Länge der Individuen exponentielle Rechenzeit benötigt.

7.7.2 Mark M. Meysenburg, Dan Hoelting, Duane McElvain: How Random Generator Quality Impacts Genetic Algorithm Performance

Motivation des Artikels sind frühere Experimente der Autoren [31], in denen ein statistischer Zusammenhang zwischen der Wahl eines PRNGs und der Performance evolutionärer Algorithmen festgestellt wurde. Der festgestellte Zusammenhang war allerdings weit weniger intuitiv, als man dies erwartet hätte. So führte auf einigen der Testfunktionen der Einsatz von schlechten PRNGs zu einer verbesserten Performance gegenüber dem Einsatz guter PRNGs. Die Qualität von PRNGs wird hierbei anhand eines ebenfalls von den Autoren entwickelten Tests gemessen.

Ziel der folgenden Experimente ist es, die beobachteten Performanceverbesserungen zu erklären. Vose beschreibt in seiner Arbeit sog. *Fixpunkte* $p_f \in \Lambda_n$, d. h. Punkte, die von Punkten in ihrer Umgebung aus mit hoher Wahrscheinlichkeit erreicht, aber nur mit sehr kleiner Wahrscheinlichkeit wieder verlassen werden. Dies sind solche Punkte, in denen die gesamte Population aus wenigen verschiedenen suboptimalen Individuum besteht, entsprechen also gerade der Konzentration der Population auf lokale Optima. Man könnte nun vermuten, dass schlechte PRNGs dazu führen, dass der Lauf eines EAs weit von seinem erwarteten Pfad abweicht und somit Fixpunkte erreicht, die im erwarteten Lauf nicht erreicht würden. Bei Experimenten, in denen kein Lauf mit einem guten PRNG das globale Optimum erreicht, könnte dies die beobachtete Performanceverbesserung erklären. Im Folgenden wird also folgende Hypothese untersucht: PRNGs schlechterer Qualität verursachen höhere Vose Discrepancy.

Im Rahmen der Experimente wird der SGA wie von Vose beschrieben mit 14 verschiedenen PRNGs auf 42 Testfunktionen untersucht. Jeder Lauf wird 32-mal mit verschiedenen Random Seeds wiederholt. Für jede Kombination aus Funktion und Random Seed wird die Population identisch (bezogen auf alle getesteten PRNGs) mit echten Zufallszahlen von random.org initialisiert. In jedem Lauf wird für jeden Generationsübergang die Vose Discrepancy berechnet. Aufgrund des bereits erwähnten Rechenaufwands arbeiten alle Funktionen auf Genotypen mit einer Länge zwischen 8 und 20 Bits.

Die gemessenen Unterschiede werden bei der Experimentauswertung nach dem nichtparametrischen Wilcoxon-Rangsummentest auf statistische Signifikanz getestet. Die Ergebnisse stimmen auf allen getesteten Funktionen im Wesentlichen überein. Lediglich einer der eingesetzten PRNGs verursacht auf allen Funktionen statistisch signifikante Performanceabweichungen. Hierbei handelt es sich um den Java-Zufallszahlengenerator, dessen Periodizität künstlich auf 1000 herabgesetzt wird, indem nach jeweils 1000 Anfragen eine Reinitialisierung mit dem ursprünglichen Random Seed vorgenommen wird.

Nach Ansicht der Autoren belegen die Ergebnisse, dass die Qualität des gewählten PRNGs

und die zu erwartende Vose Discrepancy offenbar (positiv) korreliert sind. In weiteren Experimenten soll dieser Zusammenhang weitergehend untersucht werden.

7.7.3 Erick Cantu-Paz: On Random Numbers and the Performance of Genetic Algorithms

Ausgangspunkt dieses Artikels ist ebenfalls der in früheren Experimenten beobachtete Zusammenhang zwischen der Wahl des verwendeten PRNGs und der Performance evolutionärer Algorithmen. Mit Hilfe der folgenden Experimente soll geklärt werden, wie die Qualität eines PRNGs sich auf die verschiedenen Komponenten (Initialisierung, Selektion, Mutation, Crossover) eines EAs auswirkt.

Der getestete Algorithmus entspricht im Wesentlichen wieder dem SGA wie von Vose beschrieben. Die untersuchten Quellen für (Pseudo-)Zufallszahlen sind echte Zufallszahlen von random.org (True), der Mersenne Twister (MT) und eine Variante des Mersenne Twister (MT1000), bei der wie eben nach jeweils 1000 Anfragen eine Reinitialisierung mit dem ursprünglichen Random Seed vorgenommen wird, um die Periodizität künstlich zu verkürzen. Als Fitnessfunktion dient bei allen Experimenten die Konkatenation von Trap-Funktionen. Die Fitness eines Individuums erhöht sich hierbei mit der Anzahl enthaltener Nullen, wobei das globale Optimum jedoch bei dem Bitstring, der nur Einsen enthält, liegt. Länge und Anzahl dieser sog. Building Blocks variieren von Experiment zu Experiment.

In einer ersten Experimentreihe werden Algorithmen, die ausschließlich MT bzw. MT1000 verwenden, mit Algorithmen verglichen, die für drei von vier Komponenten MT und für die verbleibende Komponente MT1000 verwenden. Die Läufe werden nach 500 Generationen gestoppt und die Anzahl der optimalen Building Blocks gemessen. Die Ergebnisse werden mit z-Tests auf statistische Signifikanz getestet. In der graphischen Darstellung der Ergebnisse finden sich nur zwei Kurven wieder. Die Verwendung von MT1000 für Selektion, Mutation oder Crossover führt nicht zu einer Abweichung von der Erfolgsrate des Algorithmus, der lediglich MT verwendet. Der Einsatz von MT1000 für die Initialisierung erzeugt die gleichen Ergebnisse wie der Einsatz von MT1000 für alle Komponenten des Algorithmus. Bei der vorgestellten Auswahl der Ergebnisse tritt dieser Effekt allerdings nur dann auf, wenn die Länge der Building Blocks als Teiler von 1000 gewählt wird. Für Länge 7 z. B. ist die Performance aller getesteten Konstellationen im Wesentlichen gleich. Dies lässt sich vermutlich dadurch erklären, dass in diesen Fällen die Periodizität von MT1000 nicht unmittelbar auf das Verhalten des Algorithmus auf den einzelnen Building Blocks übertragbar ist.

Die Ergebnisse legen die Vermutung nahe, dass der Einfluss von verschiedener PRNG-Qualität auf die Performance evolutionärer Algorithmen hauptsächlich von Unterschieden bei der Initialisierung herrührt. Diese Hypothese soll in den folgenden drei Experimenten weiter untermauert werden.

Im ersten Experiment soll untersucht werden, warum der Crossover-Operator von schlechter PRNG-Qualität scheinbar nicht beeinflusst wird. In einem Lauf über 500 Generationen wird gemessen, wie häufig jeder der möglichen Crossoverpunkte gewählt wird. Für einen guten PRNG erwarten wir eine annähernde Gleichverteilung. Tatsächlich zeigen die Ergebnisse aber, dass MT1000 im Gegensatz zu MT und True sehr stark von der erwarteten Verteilung abweicht.

Das zweite Experiment soll den Einfluss des PRNGs auf den Selektions-Operator näher untersuchen. Hierzu wird der Selektionsoperator einmalig auf eine sehr große Population angewandt. Gemessen wird dann die durchschnittliche Fitness der selektierten Population. Keiner der getesteten PRNGs weicht in diesem Test statistisch signifikant von den erwarteten Werten ab.

Mit dem dritten und letzten Experiment soll der scheinbar gravierende Einfluss des PRNGs auf die Initialisierung der Population bestätigt werden. Wir lassen jeden PRNG für verschieden lange Building Blocks eine Population initialisieren und bestimmen jeweils die Anzahl der optimal initialisierten Building Blocks. Es wird wieder eine sehr große Population gewählt, um zu erzwingen, dass MT1000 mehrfach reinitialisiert werden muss. Das Ergebnis zeigt, dass die mit MT1000 erzielten Werte wieder nicht statistisch signifikant von den Werten der anderen PRNGs abweichen. Allerdings weisen die Ergebnisse für MT1000 eine erheblich höhere Standardabweichung auf, als dies für MT und True der Fall ist.

Als Fazit stellen die Autoren fest, dass die Wahl verschiedener PRNGs offenbar erheblichen Einfluss auf die Performance evolutionärer Algorithmen hat, wobei diese scheinbar vor allem bei der Initialisierung der Population zustande kommen. Aus diesem Grunde sollte darauf geachtet werden, bei allen Experimenten die verwendeten PRNGs zu dokumentieren.

7.8 Maximale Matchings

Vortragender: Stefan Tannenbaum

Dieser Vortrag über maximale Matchings ist eine direkte Fortsetzung des Vortrags aus der ersten Seminarreihe. Er basiert ebenfalls auf [14], sowie auf einer bislang noch unveröffentlichten Fortsetzung dieser Arbeit.

7.8.1 Worst-Case-Beispiel

Zu Beginn des zweiten Vortrags wurde eine Klasse von Graphen, die bereits im ersten Vortrag kurz angesprochen wurde, genauer untersucht. Die Graphklasse hat zwei freie Parameter, die Graphen bestehen aus k Linien der Länge $2l + 1$. Entlang der Linien sind jeweils im Wechsel entweder alle Knoten benachbarter Schichten oder nur Knoten aus derselben Linie verbunden (Abbildung 7.1).

Für den (1+1) EA und RLS konnte bei $k > 2$ eine exponentielle, für $k = 2$ eine superpolynomielle erwartete Laufzeit in der Länge der Linien bewiesen werden. Der Vortrag gab einen Überblick über den Beweis und ging dabei auf einige besonders interessante Aspekte genauer ein.

Die Graphen der Klasse sind bipartit und gradbeschränkt durch $k + 1$, für $k = 2$ zusätzlich auch noch planar. Dadurch schließt der Beweis, insbesondere für den Fall $k = 2$, eine polynomielle erwartete Zeit für eine Reihe von wichtigen Graphklassen aus. Eine genaue Kategorisierung aller Graphklassen, die nicht in polynomieller erwarteter Zeit optimiert werden können, ist noch nicht gefunden und dies scheint auch nicht realistisch.

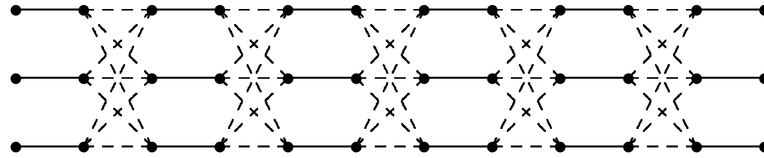


Abbildung 7.1: Graph mit 3 Linien der Länge 11 und sein perfektes Matching.

7.8.2 Bäume

Um einer Kategorisierung aber zumindest ein Stück näher zu kommen, wurden Bäume untersucht. Im Vortrag, aber auch in der Vorbesprechung wurde versucht eine Klasse von Bäumen zu finden, die für den (1+1) EA besonders schwierig zu optimieren sind, was jedoch niemandem überzeugend gelang. Die im ersten Teil untersuchten Graphen sind (für $k, l > 1$) jedenfalls keine Bäume, eine Übertragung ihrer besonderen geometrischen Eigenschaften auf Bäume erscheint auch nicht möglich.

Wir vermuten, dass alle Bäume in erwarteter Zeit $O(n^4)$ durch den (1+1) EA und die RLS optimiert werden können. Im Vortrag wurden dann wesentliche Teile eines Beweises für eine erwartete Laufzeit von $O(n^6)$ für die RLS vorgeführt. Für vollständige k-äre Bäume wurde zusätzlich eine bessere Laufzeitschranke von $O(n^4)$ gezeigt. Ein Beweis für den (1+1) EA ist in beiden Fällen noch nicht gelungen.

7.8.3 Experimente

Für Experimente mit FrEAK bietet sich besonders ein Vergleich zwischen verschiedenen Bäumen an. Wir vermuten, dass die Linie unter allen Bäumen gleicher Größe die längste erwartete Laufzeit hat, und dass diese bei $\Theta(n^4)$ liegt. Es könnte aber auch Klassen von Bäumen geben, deren erwartete Laufzeit zwischen $\Omega(n^4)$ und $O(n^6)$ liegt. Ebenfalls denkbar wäre, dass Linien prinzipiell in erwarteter Zeit $o(n^4)$ optimiert werden.

FrEAK bietet die Möglichkeit, einen Term für die erwartete Laufzeit von Linien durch Experimente zu bestimmen. FrEAK kann ebenfalls die erwartete Laufzeit auf zufälligen Bäumen einer bestimmten Größe experimentell bestimmen und ermöglicht, diese mit den Laufzeiten der entsprechenden Linie zu vergleichen. Gibt es die Vermutung, dass ein bestimmter Baum besonders schwer optimiert werden kann, so kann FrEAK dazu eingesetzt werden, diese Annahme statistisch zu bestätigen oder zu widerlegen.

7.9 Multikriterielle Optimierung

Vortragende: Dimo Brockhoff und Heiko Röglin

Literatur: Ausgangspunkt des Vortrages sind die Arbeiten von Giel [13], Laumanns, Thiele, Zitzler, Welzl und Deb [25] sowie Scharnow, Tinnefeld und Wegener [39].

SEMO

> wähle $x \in \{0,1\}^n$ uniform gleichverteilt
>
> berechne $f(x)$
>
> $A \leftarrow \{x\}$
>
> **loop**
>
> > selektiere ein $x \in A$ uniform gleichverteilt
> >
> > erzeuge durch Mutation x' aus x
> >
> > berechne $f(x')$
> >
> > **if** $\forall z \in A : x' \not\preceq_f z$
> > $A \leftarrow \{z \in A \mid z \not\preceq_f x'\} \cup \{x'\}$
> > **end if**
>
> **end loop**

Algorithmus 1: SEMO (simple evolutionary multi-objective optimizer).

Der Vortrag über multikriterielle Optimierung war in kleinere Abschnitte gegliedert, die sich die beiden Vortragenden sowohl in der Vorbereitung als auch im Vortrag selbst untereinander aufteilten. Nach einer kurzen Einleitung in das Thema multikriterielle Optimierung stellten die Vortragenden vier verschiedene Funktionen (Kürzeste Wege (SSSP), MOCO, LOTZ und eine Testfunktion) und zugehörige theoretische Ergebnisse vor. Dabei diente nur der Teil über SSSP als Ausgangspunkt für Experimente, die übrigen Vortragsthemen waren eher weiterführender Natur.

7.9.1 Einführung in multikriterielle Optimierung

Der Grund, überhaupt einen Vortrag über multikriterielle Optimierung innerhalb der Projektgruppe zu halten, ist die Tatsache, dass konkrete Probleme aus dem Alltag fast nie monokriteriell sind. Als Beispiel diente im Vortrag der Autokauf, bei dem einige Kriterien, wie Preis, Verbrauch, Komfort, etc. die Kaufentscheidung beeinflussen.

Um für die folgenden Vortragsthemen die Grundlage zu schaffen, wurden Begriffe wie Dominierung, maximale Elemente, Decision und Objective Space, Pareto Front usw. erklärt.

Ebenso wurden einige Algorithmen für die multikriterielle Optimierung vorgestellt, darunter auch der Algorithmus SEMO, wobei genauer auf die Unterscheidung zwischen local und global SEMO eingegangen wurde. Die beiden Algorithmen unterscheiden sich nur in ihren Mutationsoperatoren, wobei der local SEMO einfache 1-Bit-Mutation verwendet, der global SEMO hingegen Standardbitmutation als Mutationsoperator benutzt. Der local SEMO ist damit eine multikriterielle Version der RLS und der global SEMO entspricht der Verallgemeinerung des (1+1) EA auf mehrere Kriterien.

Weiterhin wurde bemerkt, dass sich die erwartete Worst-Case-Rechenzeit für den global SEMO asymptotisch nicht von der des (1+1) EAs unterscheidet.

7.9.2 Kürzeste Wege

Das Problem, in einem Graphen kürzeste Wege ausgehend von einem ausgezeichneten Knoten zu allen anderen Knoten des Graphen zu finden (das so genannte Single Source Shortest Path Problem), ist ein typisches Problem der kombinatorischen Optimierung, das in den Vorlesungen des Grundstudiums behandelt wird. Dort lernt man auch den Algorithmus von Dijkstra kennen, der dieses Problem für Graphen mit n Knoten in Laufzeit $\Theta(n^2)$ löst. Basierend auf der Arbeit von Scharnow et al [39] wurde vorgestellt, wie dieses Problem mit einem einfachen evolutionären Algorithmus gelöst werden kann.

Eine Probleminstanz ist gegeben durch eine $n \times n$ Distanzmatrix $D = (d_{ij})$ mit Einträgen aus $\mathbb{N} \cup \{\infty\}$, die die „Abstände" der Knoten angeben. Ein Eintrag $d_{ij} = \infty$ ist dabei so zu verstehen, dass es keine Kante vom Knoten i zum Knoten j im Graphen gibt. Weiterhin wird o. B. d. A. davon ausgegangen, dass der Quellknoten, von dem aus die kürzesten Wege berechnet werden, der Knoten n ist und dass es von n aus zu jedem anderen Knoten einen Pfad mit einer endlichen Länge gibt.

Bevor eine geeignete Fitnessfunktion definiert werden kann, muss zunächst der Suchraum festgelegt werden. Schaut man sich eine Lösung an, die vom Dijkstra-Algorithmus berechnet wird, so stellt man fest, dass die kürzesten Wege dort zusammengenommen einen Baum bilden, dessen Wurzel n ist. Man könnte also als Suchraum einfach die Menge aller Bäume nehmen, deren Wurzel n ist und die alle Knoten des Graphen enthalten. Um diese Menge im Hinblick auf eine Implementierung greifbarer zu machen, überlegt man sich, dass man solche Bäume eindeutig rekonstruieren kann, wenn man den Vorgänger jedes Knotens $i \neq n$ kennt. Da als mögliche Vorgänger eines Knotens alle anderen Knoten in Frage kommen und kein Knoten sein eigener Vorgänger sein soll, führt diese Idee auf den Suchraum

$$S = \{(x_1, \ldots, x_{n-1}) \in \{1, \ldots, n\}^{n-1} \mid \forall i : x_i \neq i\}.$$

Man stellt fest, dass die Menge S zwar alle Bäume mit Wurzel n enthält, jedoch auch weitere Elemente, die keine Bäume repräsentieren, sondern Graphen, in denen es Kreise geben kann und Knoten, die von n aus gar nicht erreicht werden. Das ist kein Fehler der Definition von S, sondern lediglich eine Eigenschaft des Suchraums, die bei der Definition der Fitnessfunktionen berücksichtigt werden muss.

Sei $l_i : S \to \mathbb{N} \cup \{\infty\}$ die Funktion, die einem Suchpunkt die Länge des durch ihn repräsentierten Weges von n nach i zuweist. $l_i(x)$ sei unendlich, wenn x entweder gar keinen Weg von n nach i beschreibt, oder der Weg von n nach i, der durch x beschrieben wird, eine Kante mit Kosten unendlich beinhaltet. Ein erster Ansatz für die Fitnessfunktion wäre es, einfach $f(x) := l_1(x) + \ldots + l_{n-1}(x)$ zu definieren. Das ist aber deswegen nicht ratsam, weil es dann ein großes Plateau mit dem Fitnesswert unendlich gibt, für dessen Verlassen im Erwartungswert lange gebraucht wird. Der Worst-Case dabei ist, dass der Graph nur aus einer Linie besteht, auf der alle Knoten liegen, und keine anderen Kanten existieren. Dann gibt es nur einen Suchpunkt mit einem endlichen Fitnesswert, wir haben also eine Situation analog zu NEEDLE vorliegen. Sei d^* der größte Eintrag in der Distanzmatrix D, der kleiner als unendlich ist. Dann gibt es zu jedem Knoten von n aus einen Weg mit Kosten kleiner als nd^*. Wenn wir $f(x)$ so definieren, dass wir für jeden Knoten, der von n aus in der Lösung x nicht erreichbar ist, anstatt unendlich die Strafkosten n^2d^* addieren und für jeden Knoten,

der in der Lösung x zwar erreichbar ist, aber über eine Kante mit Kosten unendlich, anstatt unendlich die Strafkosten nd^* addieren, geben wir dem Algorithmus Hinweise, das frühere Plateau zu verlassen.

Die auf diese Weise entstehende Fitnessfunktion wurde nicht weiter theoretisch analysiert, weil der folgende multikriterielle Ansatz für das Problem viel natürlicher erscheint. Wir definieren die multikriterielle Fitnessfunktion $f' : S \to (\mathbb{N} \cup \{\infty\})^{n-1}$ einfach durch $x \mapsto (l_1(x), \ldots, l_{n-1}(x))$, denn im Grunde ist das Problem inhärent multikriteriell. Untypisch für multikriterielle Probleme ist jedoch die Tatsache, dass es genau einen pareto-optimalen Fitnessvektor gibt.

Wegen der Eigenschaft, dass die Pareto-Front nur aus einem einzigen Punkt besteht, können wir eine Version des (1+1) EA zur Optimierung benutzen, was für multikriterielle Probleme untypisch ist, weil dort in der Regel populationsbasierte Lösungen zum Finden der Pareto-Front benutzt werden. Der für unseren Suchraum S angepasste Mutationsoperator wählt zunächst die Anzahl von lokalen Mutationen poissonverteilt mit $\lambda = 1$, in Analogie zur Standardmutation auf Bitstrings, die asymptotisch ebenfalls so viele Bitflips durchführt. Eine lokale Mutation besteht dabei aus der Veränderung einer uniform zufällig gewählten Stelle. Der analysierte Algorithmus ist der (1+1) EA mit obigem Mutationsoperator und der Konvention, dass in dem Falle, dass die Fitness des neuen Individuums unvergleichbar mit der seines Elters ist, das neu erzeugte Individuum in der Selektion unterliegt.

Die zur Analyse dieses abgeänderten (1+1) EA auf der multikriteriellen Fitnessfunktion notwendigen Methoden haben wir bereits in der ersten Seminarphase kennen gelernt. Die Analyse ist sehr einfach und basiert im Wesentlichen auf der Methode der Fitnessschichten. Sie liefert für die erwartete Zeit, bis die Pareto-Front gefunden wurde, auf jeder Instanz eine obere Schranke von $O(n^3)$. Die eigentlich gezeigte Schranke hängt vom betrachteten Graphen ab und liegt bei „typischen“ vollständigen Graphen in der Größenordnung $O(n^2 \log n)$.

Es wurden im zweiten PG-Semester Experimente gemacht werden, um den (1+1) EA auf der einkriteriellen Fitnessfunktion mit dem abgeänderten (1+1) EA auf der multikriteriellen Fitnessfunktion zu vergleichen.

7.9.3 Moco

Multi-objective Counting Ones (kurz Moco) ist eine weitere der vier vorgestellten multikriteriellen Fitnessfunktionen (siehe Giel [13]). Mit $\varphi(x) := 2\pi \frac{\|x\|}{n}$ ist sie für alle $n = 4k$ definiert als:

$$\text{Moco}(x) := (\cos \varphi(x),\ \sin \varphi(x))\,.$$

Der Vortrag versuchte die graphische Veranschaulichung aus Abbildung 7.2 durch eine eingehendere Betrachtung der Pareto Front und der Pareto Set von Moco zu vertiefen.

Zur weiteren Veranschaulichung wurde anschließend die Optimierung der Funktion Moco mittels local und global SEMO mit FrEAK vorgeführt, obwohl multikriterielle Optimierung nach anfänglicher Absprache nicht in FrEAK integriert werden sollte.

Falsche Vermutungen über die erwartete Optimierungszeit, wie sie in der Literatur getätigt wurden (Thierens [43]), sind sowohl in Giel [13] als auch im Vortrag theoretisch widerlegt

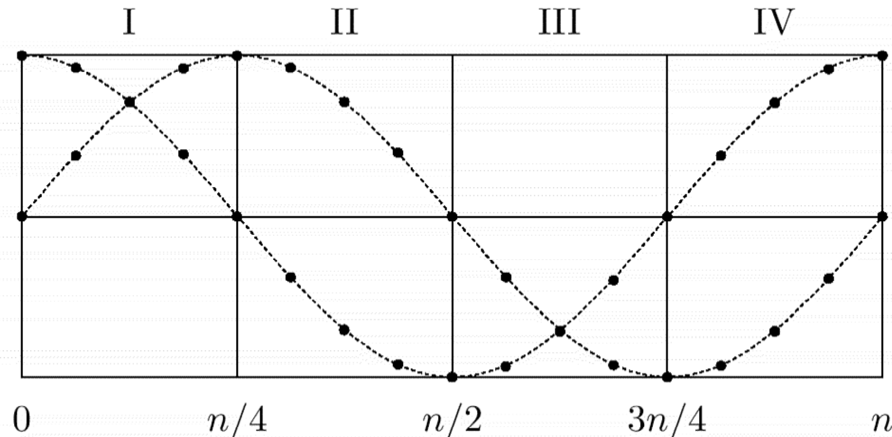


Abbildung 7.2: Anschauliche Darstellung der Funktion Moco für $n = 16$.

worden. Dazu gehörte die Vorhersage, dass die erwartete Optimierungszeit sowohl für den local als auch für den global SEMO im Bereich $n^2 \log n$ liegt. Die Realität ist, dass Moco, was die erwartete Optimierungszeit angeht, eine Art Worst-Case-Beispiel darstellt. Local SEMO hat gar keine endliche erwartete Optimierungszeit und die erwartete Optimierungszeit für global SEMO liegt bei $n^{\Omega(n)}$.

Die Optimierungszeit von $O(n^2 \log n)$ wird für den global SEMO mit einer polynomiell nahe bei Eins liegenden Wahrscheinlichkeit angenommen. Eine Erklärung dafür ist, dass die Funktion Moco sich im Wesentlichen wie Onemax verhält. Der zusätzliche Faktor n in der Laufzeit wird nötig, weil im letzten Abschnitt der Optimierung die Population die Größe $\Theta(n)$ besitzen kann. Die Wahrscheinlichkeit, nun Punkte an den Rändern der Population auszuwählen, liegt im Bereich $O(1/n)$. Diese Randpunkte können nun durch einen 1-Bit-Flip in die neue Population aufgenommen werden. Von Punkten innerhalb der Population ist es nur durch Mehrbit-Flips möglich, Suchpunkte zu generieren, die auch in die neue Population aufgenommen werden.

7.9.4 Eine Testfunktion

Die so genannte Testfunktion (siehe Giel [13]) wurde ebenfalls im Vortrag vorgestellt. In der Praxis wird die Funktion

$$x \mapsto (x^2, (x-2)^2)$$

häufig benutzt, um Algorithmen in reellwertigen Räumen zu testen. Die vorgestellte Testfunktion stellt eine Verallgemeinerung dieser reellwertigen Funktion im diskreten Raum $\{0,1\}^n$ dar.

Dabei gibt es zwei unterschiedliche Interpretationen des Bitvektors $x = (x_0, \ldots, x_{n-1})$ als Zahl: Auf der einen Seite ist es die bekannte Binärdarstellung $\mathrm{BV}(x) := \sum_{0 \le i \le n-1} x_i \cdot 2^i$ einer

Zahl, auf der anderen Seite kann der Bitvektor durch die Anzahl der Einsen als Codierung einer Zahl angesehen werden: $\|x\| := \sum_{0 \le i \le n-1} x_i$

So gibt es also zwei verschiedene Verallgemeinerungen der oben vorgestellten reellwertigen Funktion, bei der wir an einer Minimierung beider Kriterien interessiert sind:

$$\begin{aligned} f_{a,b} &= \left((\|x\| - a)^2, (\|x\| - b)^2\right) \quad 0 \le a < b \le n, \\ g_{a,b} &= \left((\mathrm{BV}(x) - a)^2, (\mathrm{BV}(x) - b)^2\right) \quad 0 \le a < b \le 2^n - 1. \end{aligned}$$

Im Vortrag wurden zusätzlich zu den Pareto Fronten der beiden Funktionen auch die Laufzeiten vorgestellt. Die erwartete Laufzeit der Algorithmen local SEMO und global SEMO auf der Funktion $f_{a,b}$ ist $O\left(n \log n + n(b-a)\log(b-a)\right)$. Die erwartete Laufzeit des global SEMO auf $g_{a,b}$ ist $\Theta(n^{k+1})$, wenn man $a = 2^k - 1$, $b = 2^k$ und $1 \le k \le n-1$ wählt. Die Beweisideen für die Laufzeiten wurden ebenfalls kurz angeschnitten.

7.9.5 Lotz

Dieser Teil des Referats basierte auf den Arbeiten von Laumanns et al [25] und von Giel [13].

Zunächst wurde die pseudoboolesche Funktion Lotz : $\{0,1\}^n \to \mathbb{N}^2$ vorgestellt. Dabei handelt es sich um eine sehr einfache Testfunktion, die durch folgende Abbildungsvorschrift definiert ist:

$$(x_1, \ldots, x_n) \mapsto \left(\sum_{i=1}^{n} \prod_{j=1}^{i} x_j, \sum_{i=1}^{n} \prod_{j=i}^{n} (1 - x_j) \right).$$

Die erste Komponente der Funktion entspricht also der bekannten Funktion LeadingOnes, während die zweite Komponente der Funktion TrailingZeroes entspricht.

Bevor wir mit der Analyse eines konkreten Algorithmus auf dieser Funktion beginnen, ist es nützlich, zunächst einige strukturelle Eigenschaften der Funktion zu untersuchen. Offenbar besteht die Pareto-Front genau aus den Fitnessvektoren (l, z) mit $l + z = n$, also aus genau $n + 1$ vielen Elementen. Eine weitere wichtige Eigenschaft, die bei der Analyse ausgenutzt wird, ist, dass eine Population P, die keine Individuen $x, x' \in P$ mit $x \prec x'$ enthält, aus höchstens $n + 1$ vielen Elementen besteht.

Die vorgestellten Analysen sind alle sehr einfach und beruhen auf Standardmethoden wie z. B. der Methode der Fitnessschichten und der Methode der typischen Läufe. Dennoch gab es bis zum Jahr 2002 keine theoretischen Analysen von evolutionären Algorithmen auf multikriteriellen Problemen, in denen Aussagen über die erwartete Optimierungszeit getroffen wurden. Die Arbeit von Laumanns et al [25] war die erste, die sich mit solchen Fragestellungen beschäftigte und die erwartete Laufzeit des lokalen SEMO und eines weiteren Algorithmus auf der Funktion Lotz untersuchte.

Es wurde gezeigt, dass der lokale SEMO im Erwartungswert $O(n^2)$ viele Fitnessauswertungen benötigt, um einen pareto-optimalen Suchpunkt zu finden, und dann nochmal $O(n^3)$ viele Schritte, um die komplette Pareto-Front zu finden. Diese Analyse nutzt zwei spezielle Eigenschaften des lokalen SEMO auf Lotz aus und lässt sich deswegen nicht direkt auf den

globalen SEMO übertragen. Die erste dieser Eigenschaften ist, dass in der ersten Phase, die beendet ist, sobald der erste pareto-optimale Suchpunkt gefunden wurde, die Population aus genau einem Individuum besteht. Das liegt daran, dass eine 1-Bit-Mutation nicht sowohl eine Änderung des Blocks der führenden Einsen als auch eine Änderung der Zahl der Nullen am Ende gleichzeitig bewirken kann, solange kein pareto-optimales Individuum mutiert wird. Die zweite Eigenschaft, die in der Analyse ausgenutzt wird, ist, dass wegen der 1-Bit-Mutationen der gefundene Teil der Pareto-Front zu jedem Zeitpunkt einen zusammenhängenden Block bildet.

Es wurde eine Analyse des globalen SEMO basierend auf der Arbeit von Giel [13] vorgestellt. Es zeigt sich, dass auch der globale SEMO mit einer erwarteten Zahl $O(n^3)$ an Fitnessauswertungen auskommt. Die entscheidende Eigenschaft, um das zu zeigen, ist, dass die Populationsgröße zu jedem Zeitpunkt durch $n+1$ beschränkt ist. Die Schranke für die erste Phase verschlechtert sich allerdings ebenfalls zu $O(n^3)$.

Die beiden SEMO-Varianten wählen jeweils uniform zufällig aus der aktuellen Population ein Individuum, das einen Nachkommen erzeugen darf. Betrachtet man den lokalen SEMO, so stellt man fest, dass lediglich die beiden Individuen, die zu den Randpunkten des bisher gefundenen Blocks der Pareto-Front gehören, durch eine 1-Bit-Mutation einen neuen Punkt auf der Pareto-Front erzeugen können. Deswegen beschreiben Laumanns et al in ihrer Arbeit den Algorithmus FEMO. Dieser ist bis auf die Selektion des Individuums, das einen Nachkommen erzeugen darf, identisch mit dem lokalen SEMO. Es wird beim FEMO ein Individuum als Elter gewählt, das bisher unter allen Individuen in der Population am wenigsten oft gewählt wurde. Wird also ein neuer Punkt der Pareto-Front gefunden, so wird das zu ihm gehörige Individuum, sofort zur Mutation ausgewählt. Das senkt die erwartete Anzahl an Fitnessauswertungen auf $O(n^2 \log n)$. Eine Analyse der globalen Variante des FEMO auf LOTZ gibt es bisher nicht.

7.10 Nutzen von Crossover und echten Populationen am Beispiel der Funktion Jump

Vortragender: Christian Gunia

Dieser Vortrag basiert vollständig auf [23].

Evolutionäre Algorithmen werden in der Theorie oft „nur“ ohne echte Populationen, d. h. mit einer Populationsgröße von 1, analysiert. In der Praxis hingegen werden fast immer Populationsgrößen von $\omega(1)$ (in der Regel $O(n)$) verwendet. Diese haben (abgesehen von der erhöhten Schwierigkeit der Analyse) einen größeren Rechenaufwand pro Generation. Des Weiteren gibt es dann zwar einen neu anwendbaren Operator Crossover, allerdings erweist es sich als schwierig, einen passenden für die jeweiligen Probleme zu finden. Es scheint als überwögen auf den ersten Blick die Nachteile. Daher wird in dem vorliegenden Paper von Thomas Jansen und Ingo Wegener [23] ein Beweis geführt, dass echte Populationen und insbesondere der Operator Crossover von Nutzen sein können, also die Anzahl an Generationen, die ein evolutionärer Algorithmus braucht, bis er erstmals das globale Optimum gefunden hat, deutlich senken kann. Der Effekt ist so groß, dass auch die Anzahl an Funktionsauswertungen sinkt.

Um den Vorteil von Crossover richtig nutzen zu können, schafft man (künstlich) eine Situation, in der viele Bits gleichzeitig flippen müssen, um einen Fortschritt zu erzielen. Dies ist für evolutionäre Algorithmen i. A. schwierig. Für einen Algorithmus, der Crossover und damit echte Populationen verwendet, darf man jedoch die Hoffnung haben, dass der mit einem Crossover von zwei passenden (in der Population vorhandenen) Individuen, dieses Ziel viel schneller erreicht.

Eine mögliche Realisierung ist die folgende Fitnessfunktion mit $n \in \mathbb{N}$ und $k \in \{1, \ldots, n\}$:

$$\text{Jump}_{n,k}(x) := \begin{cases} k + \|x\|_1 & \text{falls } \|x\|_1 \leq n - k \text{ oder } \|x\|_1 = n \\ n - \|x\|_1 & \text{sonst} \end{cases}$$

wobei $\|x\|_1 := x_1 + x_2 + \cdots + x_n = \text{OneMax}(x)$.

Im Prinzip entspricht diese Funktion der bekannten Fitnessfunktion OneMax, allerdings existiert „vor dem Optimum“ eine Streifen der Breite k mit den niedrigsten Fitnesswerten, der von einem evolutionären Algorithmus ohne Crossover in einem Schritt übersprungen werden muss. Im Fall $k > 1$ bestimmt die Wartezeit auf diesen Schritt die Optimierungszeit $O(n^k)$ und im Fall $k = 1$ entspricht die Funktion OneMax, so dass wir $O(n \log n)$ als Optimierungszeit erhalten. Insgesamt benötigt ein (1+1) EA für die Optimierung der Funktion eine Laufzeit von $\Theta(n \log n + n^k)$.

In dem Paper wird ein Steady-State GA als Gegenstück betrachtet. Dieser ist im Wesentlichen ein (μ+1) EA, der eine beinahe beliebige Selektion verwendet, die lediglich bessere Individuen nicht benachteiligen darf, ein uniformes Crossover mit Wahrscheinlichkeit p_c und auf jeden Fall eine Standardbitmutation ausführt. Sollte nur die Mutation ausgeführt werden, kann man optional verhindern, dass das Individuum unverändert bleibt.

Für eine nicht zu große Lücke $k_n = O(\log n)$, eine geringe Crossoverwahrscheinlichkeit $p_c \leq \frac{1}{\text{Const} \cdot k_n n}$ und eine Populationsgröße μ, die den Ungleichungen $\mu \geq k_n \log^2 n$ und $\mu = n^{O(1)}$ genügt, beträgt die erwartete Laufzeit des Steady-State GAs auf $\text{Jump}_{n,k}$ unter Vermeidung von Kopien

$$O\left(\mu n \left(k_n^2 + \log(\mu n)\right) + \frac{2^{2k_n}}{p_c}\right).$$

Ohne die Vermeidung von Kopien kann man die obere Schranke

$$O\left(\mu n^2 k^3 + \frac{2^{2k}}{p_c}\right)$$

zeigen. Insgesamt sind beide Laufzeiten jedoch im Allgemeinen deutlich unter der Laufzeit des (1+1) EAs, somit hat das Crossover eine Wirkung erzielt.

Zum Beweis wird der Lauf des Algorithmus in drei typische Phasen eingeteilt. Eine Phase beginnt, sobald die vorherige endet und die erste beginnt sofort. Die Phasen lauten:

- Erreichen der Sprungstelle, d. h. $\|x\|_1 = n - k$ für alle Individuen der Population bzw. Optimum erreicht.

- Verteilen an der Sprungstelle, d. h. die nicht gesetzten Bits verteilen sich in den Individuen „gleichmäßig“ bzw. Optimum erreicht.
- Sprung ins Optimum.

In den ersten beiden Phasen werden Crossoverschritte dadurch analysiert, dass ihre positiven Effekte nicht berücksichtigt werden, und nur die dritte Phase verwendet Crossover, um die Laufzeit abzuschätzen. Die Analyse der Phase 1 ist recht kanonisch und einfach. Es reicht eine passende 1-Bit-Mutation aus, um den Funktionswert zu erhöhen, da die Funktion in Phase 1 im Wesentlichen OneMax entspricht. Daher erhalten wir aus dem Coupon-Collectors-Theorem die Schranke $O(\mu n \log \mu n)$. In Phase 3 müssen wir lediglich zwei passende Individuen selektieren und ein passendes Crossover durchführen. Die Individuen dürfen an keiner Stelle gemeinsame Nullen haben. Danach wählen wir in dem uniformen Crossover an jeder Stelle eine der mindestens zwei vorhandenen Einsen. Da die Wahrscheinlichkeit für solche eine Wahl durch $p_c \cdot \frac{1}{2} \cdot 2^{-2k} \cdot \frac{1}{2e}$ beschränkt ist, reichen dafür mit großer Wahrscheinlichkeit $O\left(\frac{2^{2k}}{p_c}\right)$ Generationen aus.

Lediglich Phase 2 bereitet etwas Schwierigkeiten, da die günstigen bzw. ungünstigen Ereignisse nicht leicht zu überblicken und somit deren Wahrscheinlichkeiten auch nicht leicht abzuschätzen sind. Daher werden diese „unübersichtlichen“ Ereignisse in kleinere, übersichtlichere zerlegt, deren Wahrscheinlichkeiten bekannt sind, und dann aus diesen zusammengesetzt.

Wir betrachten die Population als $\mu \times n$-Matrix. Damit enthält jede Zeile k Nullen (in dieser Phase) und wir haben insgesamt somit μk Nullen in der Matrix. Nun zeigen wir, dass die Anzahl die Nullen sich über die Spalten „verteilen“, wir also mit großer Wahrscheinlichkeit nach $O(\mu n k^2)$ Generationen (unter Kopienvermeidung) nur noch höchstens $\frac{\mu}{4k}$ Nullen pro Spalte haben. Dazu betrachten wir nur die erste Spalten und multiplizieren die Fehlschlagwahrscheinlichkeit hinterher als Ausgleich mit n. Dies ist offensichtlich korrekt, da die Spalten völlig äquivalent sind.

In einem Schritt des Algorithmus kann sich die Anzahl der Nullen pro Spalte höchstens um 1 verändern. Es gelingt uns nachzuweisen, dass die Anzahl der Nullen tendenziell sinkt, solange ihre Anzahl mindestens $\frac{\mu}{8k}$ beträgt. Chernoff-Schranken ermöglichen es uns die Wahrscheinlichkeit abzuschätzen, dass ihre Anzahl in den $O(\mu n k^2)$ Generationen nicht unter die gewünschte Schranke fällt. Diese Wahrscheinlichkeit ist nach unten durch eine positive Konstante beschränkt und das Argument der unabhängigen Wiederholungen liefert uns die erwartete Phasenlänge von $O(\mu n k^2)$. Den Fall ohne Kopienvermeidung beweisen wir völlig analog.

Wir haben also gesehen, dass echte Populationen und Crossover eine superpolynomielle Laufzeit zu einer polynomiellen verändern können. Leider ist der betrachtete Steady-State GA insofern untypisch, dass die Crossoverwahrscheinlichkeit von $\frac{1}{\text{Const}\cdot k_n n}$ extrem niedrig und vor allem nicht durch eine Konstante nach unten beschränkt ist.

7.11 Monotone Polynome

Vortragender: Bastian Degener

Literatur: Der gesamte Vortrag beruht auf [48].

Die Analyse evolutionärer Algorithmen auf der Klasse der monotonen Polynome ist dadurch motiviert, dass diese Klasse eine große Funktionenmenge umfasst, die Suche auf Plateaus möglich ist und der Hammingabstand zum Optimum sich während der Suche vergrößern kann. Andererseits ist die Optimierung von monotonen Polynomen algorithmisch einfach genug, so dass wir auch mit EAs auf gute Laufzeiten hoffen können. Bei der Analyse wurden erstmals Techniken angewendet, die auch in Zukunft bei der Analyse evolutionärer Algorithmen von Vorteil sein können.

Pseudoboolesche Polynome haben die Form

$$f(x) = \sum_{A \subseteq \{1,\dots,n\}} w_A \cdot \prod_{i \in A} x_i.$$

Bei monotonen Polynomen sind entweder alle Gewichte positiv oder alle Gewichte negativ. Im Folgenden werden o. B. d. A. nur Polynome mit positiven Gewichten untersucht, die ihr Optimum bei 1^n haben. Den größten Grad bezeichnen wir als d.

Zunächst wird die Optimierung von Monomen betrachtet, da Polynome aus Monomen bestehen. Das Ziel ist die Analyse des (1+1) EAs, die allerdings recht schwierig ist, und erst am Ende des Vortrags vorgestellt wird. Die RLS ist im Prinzip einfacher zu analysieren. Um die Lücke zwischen diesen beiden Algorithmen zu schließen, wird RLS_p betrachtet. Dies ist eine Modifikation von RLS. Zusätzlich einem Bit, das auf jeden Fall flippt, flippt jedes weitere Bit mit Wahrscheinlichkeit p. Für $p = 0$ entspricht dies der RLS, bei $p = \frac{1}{n}$ ist RLS_p dem (1+1) EA sehr ähnlich. Daher werden Werte zwischen 0 und $1/n$ betrachtet.

7.11.1 Monome

Bei der Analyse von Monomen werden die entstehenden Markovketten der betrachteten Algorithmen ohne den Selektionsoperator untersucht. Die erwartete Zeit für den Übergang von jedem Zustand in den nächst höheren Zustand wird abgeschätzt und anschließend wird über alle diese Übergänge summiert. Dabei ergibt sich eine obere Schranke von $O(\frac{n}{d} \cdot 2^d)$. Diese Schranke ist asymptotisch optimal, wie sich aus der Betrachtung der Blackbox-Komplexität ergibt.

7.11.2 RLS

Die RLS ist einfach zu untersuchen, da einmal aktivierte Monome, das heißt, alle Bits im Monom sind 1, nicht mehr deaktiviert werden. Außerdem sorgen aktivierte Monome dafür, dass andere Monome, die von ihnen überlappt werden, im Grad verringert werden, da diese Stellen fest auf 1 bleiben. Es ergibt sich eine obere Schranke von $O(\frac{n}{d} \cdot \log(\frac{n}{d} + 1) \cdot 2^d)$. Dass diese Schranke auch scharf ist, zeigt der nächste Abschnitt.

7.11.3 Worst-Case-Beispiel

Wie aus den Überlegungen zur RLS deutlich wird, hat ein Worst-Case-Beispiel viele nicht-überlappende Monome von möglichst großem Grad. Dies wird von Royal Road Funktionen erfüllt. Tatsächlich lässt sich hiermit eine untere Schranke von $\Omega(\frac{n}{d} \cdot \log(\frac{n}{d} + 1) \cdot 2^d)$ für die RLS zeigen. Auch RLS_p und der (1+1) EA brauchen asymptotisch im Wesentlichen genauso lange.

7.11.4 RLS_p

Die Analyse wird deutlich schwieriger, wenn drei oder mehr Bits flippen können, weil aktive Monome wieder deaktiviert werden können, und sich die Analyse der Monome nicht mehr auf einfache Markovketten beschränkt. Aus den bisherigen Überlegungen folgt, dass wir uns auf $d = O(\log(n))$ beschränken können, und höchstens $O(\frac{n}{d} \cdot 2^d)$ für die erwartete Zeit bis zur Optimierung eines Monoms erwarten können. Für kleine p kann dies bewiesen werden.

Im Beweis wird die Wahrscheinlichkeit von Fehlschlagsereignissen abgeschätzt. Eines dieser Ereignisse ist, dass drei oder mehr Bits in einer Phase flippen. Ist dies nicht der Fall, können die bereits betrachteten, sowie weitere Ergebnisse aus der Theorie der einfachen Markovketten benutzt werden. Tatsächlich ist der Vergleich zweier Markovketten in diesem Beweis ein wesentliches Analysetool.

Der Optimierungsfortschritt eines monotonen Polynoms wird mit Hilfe einer Pseudofitness, die die essenziellen Einsen zählt, gemessen. Essenziell sind Einsen in aktivierten Monomen. Mit Hilfe einer Driftanalyse wird eine Schranke von $O(\frac{n^2}{d} \cdot 2^d)$ gezeigt. Wesentlich ist, dass die Wahrscheinlichkeit für flippende Einsbits bei gleichzeitig flippenden Nullbits durch die Beschränkung auf kleine p sehr gering gehalten wird.

7.11.5 (1+1) EA

Die Analyse der durchschnittlichen Wartezeit bis zur Aktivierung eines Monoms verläuft analog zur Analyse von RLS_p.

Ein analoges Resultat für das gesamte Polynom lässt sich nicht erzielen, da die Flip-Wahrscheinlichkeit zu hoch ist. Allerdings lässt sich eine Schranke von $O(N \cdot \frac{n}{d} \cdot 2^d)$ mit Hilfe von Fitnessschichten zeigen. Dabei ist N die Anzahl der nichttrivialen Monome. Es wird aber vermutet, dass die Schranke noch deutlich verbessert werden kann.

Kapitel 8

Hypothesenfindung

Inhalt

8.1 Das Ising-Modell 116

8.1.1 Das Ising-Modell auf Cliquen 116

8.1.2 Das Ising-Modell auf Tori 129

8.1.3 Das Ising-Modell auf booleschen Hypercubes 132

8.2 Minimale Spannbäume 136

8.2.1 Experimente auf einem asymptotischen Worst-Case-Beispiel 136

8.2.2 Experimente zu Kantengewichten 139

8.2.3 Experimente zum Vergleich der Algorithmen (1+1) EA und RLS . . 140

8.2.4 Experimente zur Darstellung von Spannbäumen als Prüfernummern 141

8.3 Maximale Matchings 143

8.3.1 Matchings auf Bäumen und Pfaden 143

8.3.2 Nutzen von Uniform Crossover bei Matchings auf Pfaden 144

8.3.3 Matchings auf Semi-Random-Graphen 145

In der zweiten Seminarphase der Projektgruppe wurden wöchentlich Vorträge zu verschiedenen Themen gehalten und diese Themen anschließend ausführlich diskutiert. Einige dieser Vorträge beschäftigten sich mit Fragen der Methodik wissenschaftlicher Untersuchungen, andere wiederum mit der Analyse evolutionärer Algorithmen auf konkreten Problemen. Aus den Vorträgen, die konkrete Probleme behandelten, entwickelten sich interessante Fragestellungen, die daraufhin näher untersucht wurden.

In einer ersten Phase wurden nach dem Vorträgen zu den einzelnen Themen Erkundungsexperimente durchgeführt, die zur Hypothesenfindung dienten. Die Experimente waren durch die Ergebnisse und die aufgeworfenen Fragen aus den Vorträgen und den dazu gehörigen Diskussionen motiviert und waren dementsprechend breit angelegt. Dadurch wurde viel Information zu den Themengebieten erworben und es ergaben sich neue Einsichten und Fragestellungen, die zu konkreten Hypothesen und demzufolge zielgerichteten Experimenten führten.

Anschließend wurden alle Hypothesen gesammelt und die Hypothesen heraus gesucht, die durch umfassendere Experimente und fundierte Hypothesentests weiter untersucht werden sollten.

In diesem Kapitel werden zunächst die auf diese Weise gefundenen Hypothesen motiviert und erläutert. In Kapitel 9 ist dann die konkrete Durchführung der geplanten Experimente beschrieben.

8.1 Das Ising-Modell

Das Ising-Modell wurde bereits in zwei Seminarvorträgen behandelt, die in den Abschnitten 7.3 und 7.4 beschrieben sind. Auf dem Ring wurde das Ising-Modell von Simon Fischer und Ingo Wegener [11] bereits umfassend analysiert, während auf dem Torus noch einige Fragen offen waren. Diese Fragen wurden daraufhin in der PG diskutiert, u. a. die Frage, unter welchen Umständen und mit welcher Wahrscheinlichkeit verschiedene Typen von Ringen entstehen können.

Auch andere Typen von Graphen wurden in die Diskussionen mit einbezogen. Dies sind zum einen Graphen, die aus zwei miteinander verbundenen Cliquen bestehen, da man auf ihnen gut beobachten kann, wie sich das Ising-Modell auf dichten Komponenten verhält. Eine weitere spannende Frage ist hier, in wie weit sich die Cliquen gleichartig färben und wann es vorkommt, dass nur ein lokales Optimum erreicht wird, bei dem die Cliquen unterschiedlich gefärbt sind.

Zum anderen wurde der boolesche Hypercube betrachtet, da Hypercubes eine kanonische Erweiterung des Torus und des Ringes sind. Auf dem Torus können sich Ringe als stabile Unterstrukturen entwickeln, die den weiteren Optimierungsprozess erschweren können. Interessant war daher u. a. die Frage, wie solche stabilen Unterstrukturen in höherdimensionalen Räumen aussehen.

Die Hypothesen, die sich im Laufe der Diskussionen und in Gruppenarbeit entwickelt haben, sollen im Folgenden motiviert und beschrieben werden.

8.1.1 Das Ising-Modell auf Cliquen

Einleitung und Motivation

Das Ising-Modell mit zwei Farben 0 und 1 wurde auf Graphen $G = (V, E)$, $V := \{v_0, \dots, v_{n-1}\}$ untersucht, die aus zwei Cliquen $C_1 := \{v_0, \dots, v_{n/2-1}\}, C_2 := \{v_{n/2}, \dots, v_{n-1}\}$ mit jeweils $n/2$ Knoten bestehen. Dabei betrachten wir nur gerade n, um Cliquen gleicher Größe zu erhalten. Zusätzlich enthalten die betrachteten Graphen Kanten, die Knoten verschiedener Cliquen miteinander verbinden. Diese Kanten bezeichnen wir im Folgenden als *Brückenkanten.*

Knotenfärbungen notieren wir durch $x = (x_0, \dots, x_{n-1}) \in \{0,1\}^n$, wobei x_i die Färbung des Knotens v_i ist. Insbesondere beziehen sich also $x_0, \dots, x_{n/2-1}$ auf die erste Clique und $x_{n/2}, \dots, x_{n-1}$ auf die zweite Clique.

Auf Graphen G ohne Brückenkanten ist die Ising-Funktion separabel: die Fitness einer Clique ist unabhängig von der Fitness der anderen Clique. Zudem ist die Ising-Funktion auf einzelnen Cliquen spin-flip symmetrisch, d. h. wenn x^i die Färbung der i-ten Clique ist, gilt

$$f_G(x^1x^2) = f_G(\overline{x^1}x^2) = f_G(x^1\overline{x^2}) = f_G(\overline{x^1x^2}).$$

Allgemeine Suchheuristiken wie die randomisierte lokale Suche oder der (1+1) EA haben daher eine Wahrscheinlichkeit von genau 1/2, einen einheitlich gefärbten Graphen zu erhalten. Wenn man die Cliquen jedoch durch Brückenkanten verbindet, erhöht sich die Tendenz des Ising-Modells, beide Cliquen gleichartig zu färben. Hier ergeben sich interessante Fragestellungen nach Anzahl und Struktur der Brückenkanten. Welche Strategie für die Wahl der Brückenkanten hat den größten Einfluss auf das Verhalten der betrachteten Algorithmen und die größte Chance, eine einheitliche Färbung des Graphen zu erreichen?

Beobachtungen und Folgerungen

Anfangs wurden Erkundungsexperimente durchgeführt, um Hypothesen zu entwickeln. Dazu wurden in FrEAK entsprechende Module implementiert: eine Fitnessfunktion für das Ising-Modell auf Cliquen, ein Stoppkriterium, das den Lauf abbricht, sobald alle Cliquen jeweils einheitlich gefärbt sind und ein Anzeigemodul (View), das die Cliquen und die Knotenfärbungen grafisch darstellt.

Über die Konfiguration des Suchraums kann man die Zahl der Knoten und der Farben einstellen. Die Fitnessfunktion lässt sich wie folgt parametrisieren: die Zahl der Cliquen lässt sich einstellen (sofern die Zahl der Cliquen ein Teiler der Knotenzahl ist, da alle Cliquen die gleiche Größe haben sollen), ebenso die Zahl der Brückenkanten und die Verteilungsstrategie für die Brückenkanten. Brückenkanten können gleichverteilt zufällig gewählt werden oder möglichst gleichmäßig auf alle Knoten verteilt werden, so dass alle Knoten einen möglichst ähnlichen Grad bekommen. Unabhängig von der Verteilungsstrategie gibt es die Möglichkeit, die Knotenmenge für Anfangs- und Endpunkte von Brückenkanten einzuschränken, so dass sich Brückenkanten auf Teilmengen der Cliquenknoten konzentrieren lassen.

Sowohl der (1+1) EA als auch die (1+1) randomisierte lokale Suche (RLS) wurden daraufhin mit Hilfe dieser Module simuliert und der Verlauf des Optimierungsprozesses in FrEAK beobachtet.

Dabei wurden folgende Beobachtungen gemacht, aus denen später teilweise Hypothesen entstanden sind.

Beobachtung 1. *Wenn die maximale Zahl adjazenter Brückenkanten d für alle Knoten nicht zu groß ist, haben die Brückenkanten keinen großen Einfluss auf das Verhalten der Algorithmen.*

Dies kann man wie folgt erklären: jeder Knoten ist zu maximal d Knoten in der anderen Clique adjazent und zu $n/2 - 1$ Knoten in der eigenen Clique. Wenn d daher klein ist, fällt der Anteil der Kanten in der anderen Clique kaum ins Gewicht.

Beobachtung 2. *Sowohl der (1+1) EA als auch die (1+1) RLS nähern sich einem lokal optimalen Punkt recht schnell an. In einem Lauf steht daher schon früh fest, in welche Richtung sich die Färbung der Cliquen entwickeln wird.*

Betrachtet man die Zahl 1-gefärbter Knoten in einer Clique, so stellt man fest, dass sich dieser Wert über die Zeit auffallend monoton verhält und bis auf kleine Abweichungen entweder monoton steigt oder monoton sinkt. Wenn also zu einem Zeitpunkt eine ausreichend große Mehrheit der Knoten in der Clique mit einer Farbe c gefärbt ist, dann ist es sehr wahrscheinlich, dass die gesamte Clique nach und nach mit dieser Farbe gefärbt wird.

Theoretische Ergebnisse für die (1+1) RLS

Für die randomisierte lokale Suche und Graphen mit beschränkter Zahl adjazenter Brückenkanten für jeden Knoten lässt sich diese Beobachtung theoretisch untermauern. Man kann sogar leicht beweisen, dass ab einer genügend großen Mehrheit einer Farbe c, o. B. d. A. $c = 1$, das erreichte lokale Optimum deterministisch vorherbestimmt ist.

Lemma 1. *Sei C_i eine Clique mit $k := n/2$ Knoten, bei dem jeder Knoten zu höchstens d Brückenkanten inzident ist. Dann gilt: wenn $||C_i||_1 \geq k/2+d/2+1$, dann färbt die (1+1) RLS C_i mit Farbe 1 mit Wahrscheinlichkeit 1 in erwarteter Zeit $O(n \log n)$.*

Beweis. Der Beweis ist einfaches Rechnen.
Wenn ein 0-gefärbter Knoten geflippt wird, erhöht sich die Fitness um mindestens

$$||C_i||_1 - (||C_i||_0 - 1) - d = 2||C_i||_1 - k - d + 1 \geq 3$$

und der Nachkomme wird akzeptiert. Wenn ein 1-gefärbter Knoten geflippt wird, verändert sich die Fitness um höchstens

$$-(||C_i||_1 - 1) + ||C_i||_0 + d = -2||C_i||_1 + k + d + 1 \leq -1$$

und der Nachkomme wird nicht akzeptiert.
Induktiv folgt, dass die Bedingung $||C_i||_1 \geq k/2+d/2+1$ für den jeweils aktuellen Suchpunkt stets erfüllt bleibt, was zu der Wahrscheinlichkeit von 1 führt. Die Fitness kann durch das Flippen 0-gefärbter Knoten erhöht werden. Wenn i die Zahl 0-gefärbter Knoten ist, beträgt die Wahrscheinlichkeit einer solchen Verbesserung i/n. Die erwartete Zeit, bis C_i mit Farbe 1 gefärbt ist, ist daher nach oben beschränkt durch

$$\sum_{i=k/2+d/2+1}^{k-1} \frac{n}{i} = O(n \log n).$$

□

Das Lemma können wir nun anwenden, um auch Beobachtung 1 in folgendem Korollar theoretisch zu begründen: bei einer kleinen Zahl von Brückenkanten kommt es sehr oft vor, dass bei der (1+1) RLS das Ergebnis des Laufs bei der Initialisierung bereits fest steht, so dass die Brückenkanten in diesem Fall keinen Einfluss auf das Ergebnis haben.

Korollar 1. *Wenn* $0 < d < \sqrt{k}$*, erreicht die (1+1) RLS entweder ein lokales oder ein globales Optimum mit jeweils konstanter Wahrscheinlichkeit in erwarteter Zeit* $O(n \log n)$*.*

Beweis. Die Farbverteilung in beiden Cliquen bei der zufälligen Initialisierung kann durch zwei unabhängige Binomialverteilungen beschrieben werden, mit Parametern jeweils $k/2$ und $1/2$. Die Wahrscheinlichkeiten für die Ereignisse $||C_i||_1 \leq k/2 - \sqrt{k}$ und $||C_i||_1 \geq k/2 + \sqrt{k}$ sind beide durch eine Konstante $0 < c < 1/2$ nach unten beschränkt. Wenn für beide Cliquen jeweils eines dieser beiden Ereignisse eintritt, können wir Lemma 1 anwenden. Aus Symmetriegründen erreicht die (1+1) RLS daher mit einer Wahrscheinlichkeit von mindestens $2c^2$ ein lokales Optimum und mit einer Wahrscheinlichkeit von mindestens $2c^2$ ein globales Optimum in erwarteter Zeit $O(n \log n)$. □

Übertragung der Ergebnisse auf den (1+1) EA

Wir vermuten, dass eine zu Lemma 1 ähnliche Aussage auch für den (1+1) EA gilt, wenn man die Schranke $k/2 + d/2 + 1$ um einen kleinen sublinearen Term erhöht. Dieser Term wird vermutlich nötig sein, um die Wahrscheinlichkeit zu senken, dass die Mehrheit in einem Schritt durch eine Mutation mehrerer Bits verloren geht.

Wenn in einer Clique C_i ein Knoten mit Farbe c geflippt wird und die Mehrheit dieser Farbe groß genug ist, $||C_i||_c \geq k/2 + d/2 + 1$, zeigt der Beweis von Lemma 1, dass der Nachkomme mit Wahrscheinlichkeit 1 nicht akzeptiert wird, da die Fitness verschlechtert wird. Der (1+1) EA kann solche Schritte akzeptieren, wenn gleichzeitig die Fitness in der anderen Clique entsprechend erhöht wird. Es ist daher nicht auszuschließen, dass die Mehrheit der Farbe c in mehreren solcher Schritte verloren geht, was eine theoretische Analyse wesentlich erschwert.

Eine Übertragung der Ergebnisse von der (1+1) RLS auf den (1+1) EA ist daher noch nicht gelungen.

Lokale Optima und der Treppengraph

In den meisten Szenarien von Brückenkanten gibt es beim Ising-Modell auf zwei verbundenen Cliquen vier lokale Optima, von denen zwei globale Optima sind, nämlich die beiden einheitlichen Färbungen 0^n und 1^n. Rein lokale Optima sind die Färbungen $0^{n/2}1^{n/2}$ und $1^{n/2}0^{n/2}$, in denen beide Cliquen jeweils einheitlich, jedoch voneinander verschieden gefärbt sind.

Von dieser Faustregel gibt es jedoch mehrere Ausnahmen, die hier aufgeführt werden sollen.

1. Wenn es keine Brückenkanten gibt, gibt es keine rein lokalen Optima, sondern vier globale Optima.

2. Wenn die Zahl an Brückenkanten das Maximum $n^2/4$ annimmt, bilden die beiden Cliquen und die Brückenkanten den vollständigen Graphen auf n Knoten und es gibt nur zwei globale Optima. Dieser Effekt kann auch bei einer kleineren Zahl von Brückenkanten auftreten.

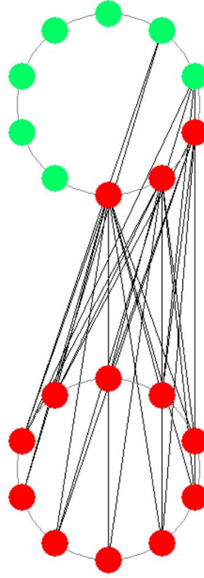


Abbildung 8.1: Eine Visualisierung des Treppengraphen G^{stairs} für $n = 20$ mit zufälliger Wahl der Endpunkte in der unteren Clique. Die Cliquen sind durch Kreise symbolisiert; lediglich die Brückenkanten sind eingezeichnet.

3. Wenn die Brückenkanten stark konzentriert sind (z. B. wenn sie eine Clique bilden, deren Größe im Intervall $(n/3, n/2)$ liegt), können neue lokale Optima entstehen, bei denen innerhalb einer Clique die zu vielen Brückenkanten adjazenten Knoten anders gefärbt sind als die zu wenigen Brückenkanten adjazenten Knoten.

Ein Beispiel für das zweite Szenario mit einer stark verringerten Zahl an Brückenkanten ist der von uns als „Treppengraph“ bezeichnete Graph G^{stairs}. Dieser enthält zwei Cliquen $\{v_0, \ldots, v_{n/2-1}\}, \{v_{n/2}, \ldots, v_{n-1}\}$ und für $0 \leq i < n/4$ gilt: v_i ist zu $n/2-2i$ beliebigen Knoten der anderen Clique adjazent; die Knoten $v_{n/4}, \ldots, v_{n/2-1}$ haben keine Brückenkanten. Der Graph G^{stairs} ist exemplarisch für $n = 20$ in Abbildung 8.1 dargestellt, wobei die Endpunkte in der zweiten Clique zufällig gewählt wurden.

Die Zahl der Brückenkanten in G^{stairs} beträgt $n^2/16 + n/4$, was in etwa einer Kantendichte von $1/4$ entspricht, bezogen auf die maximal mögliche Zahl $n^2/4$ von Brückenkanten.

Wenn der (1+1) EA auf G^{stairs} den Punkt $0^{n/2}1^{n/2}$ erreicht hat ($1^{n/2}0^{n/2}$ analog), kann die Fitness verbessert und ein globales Optimum erreicht werden, indem die Knoten $v_0, \ldots, v_{n/4-1}$ in dieser Reihenfolge geflippt werden. Daraus folgt: Das Ising-Modell auf G^{stairs} ist unimodal und enthält zwei symmetrische Pfade $(0^{n/2}1^{n/2}, 10^{n/2-1}1^{n/2}, \ldots, 1^{n/4-1}0^{n/4+1}1^{n/2})$ und $(1^{n/2}0^{n/2}, 01^{n/2-1}0^{n/2}, \ldots, 0^{n/4-1}1^{n/4+1}0^{n/2})$ der Länge $n/4$ mit streng monoton steigenden Fitnesswerten.

Wenn sich zeigen ließe, dass der (1+1) EA mit konstanter Wahrscheinlichkeit einen Punkt auf der ersten Hälfte eines Pfades erreicht, ließe sich daraus eine untere Schranke $\Omega(n^2)$ für die erwartete Zeit ableiten, bis zum ersten Mal ein lokales Optimum erreicht wird. Dies würde anfängliche Vermutungen widerlegen, dass der (1+1) EA nur $O(n \log n)$ erwartete Schritte braucht, um ein lokales Optimum zu erreichen.

Hypothese 1. *Die Wahrscheinlichkeit, dass der (1+1) EA hat auf dem Ising-Modell auf dem Treppengraphen G^{stairs} einen Punkt auf der ersten Hälfte eines der beiden Pfade erreicht, ist durch eine Konstante $c > 0$ nach unten beschränkt.*

Es ist prinzipiell nicht möglich, diese Hypothese experimentell zu bestätigen, da wir mit Experimenten keine asymptotischen Aussagen machen können. Wenn wir jedoch annehmen, dass diese Hypothese für alle n gilt, lässt sich die Schranke $\Omega(n^2)$ für beide Algorithmen wie folgt zeigen.

Definition. *Wir bezeichnen mit x^i für $0 \leq i \leq n-1$ den Suchpunkt, der aus $x = (x_0, \ldots, x_{n-1})$ dadurch entsteht, dass x_i flippt: $x^i := (x_0, \ldots, x_{i-1}, \overline{x_i}, x_{i+1}, \ldots, x_{n-1})$.*
Dann sei $g(x, i)$ für $0 \leq i \leq n-1$ die Veränderung der Fitness, wenn nur das i-te Bit flippt:

$$g(x, i) := f(x^i) - f(x).$$

Theorem 1. *Unter der Annahme, dass die (1+1) RLS auf dem Ising-Modell auf G^{stairs} mit einer durch eine Konstante $c > 0$ nach unten beschränkten Wahrscheinlichkeit einen Punkt auf der ersten Hälfte eines Pfades erreicht, ist die erwartete Zeit, bis die (1+1) RLS einen optimalen Punkt findet, nach unten beschränkt durch $\Omega(n^2)$.*

Beweis. Wir nehmen an, dass die (1+1) RLS einen Suchpunkt x auf der ersten Hälfte eines Pfades erreicht hat. Falls dieses Ereignis nicht eintritt, schätzen wir die Laufzeit mit der trivialen Schranke 0 ab.

Für den i-ten Suchpunkt eines Pfades, $x = 1^i 0^{n/2-i} 1^{n/2}$ oder $x = 0^i 1^{n/2-i} 0^{n/2}$ für $0 \leq i < n/4$ gelten nach Konstruktion von G^{stairs} folgende Werte von g für alle Knoten der ersten Clique.

$$\forall 0 \leq j < n/2 \qquad g(x, j) = \begin{cases} -2(i-j)+1, & 0 \leq j < i \\ 1, & j = i \\ -2(j-i)+1, & i < j < n/4 \\ -n/2+2i+1, & n/4 \leq j < n/2 \end{cases}$$

Für alle Knoten der zweiten Clique, $n/2 \leq j < n$, gilt $g(x, j) \leq n/4 - (n/2 - 1) = -n/4 + 1$, denn da nur $n/4$ Knoten der ersten Clique zu Brückenkanten inzident sind, ist jeder Knoten $v_{n/2}, \ldots, v_{n-1}$ zu maximal $n/4$ 0-gefärbten Knoten adjazent.

Die einzige Mutation, die zu einer nicht negativen Fitnessveränderung führt und die daher als einzige akzeptiert wird, ist daher die Mutation des Bits x_i. Die erwartete Zeit, bis die zweite Hälfte des Pfades durchquert wird, ist nach unten beschränkt durch $c \cdot (1/8)n^2 = \Omega(n^2)$. $\square$

Um für den Beweis der Schranke für den (1+1) EA Abhängigkeiten zwischen mehreren flippenden Bits in den Griff zu bekommen, benötigen wir folgendes Lemma, das Fitnessveränderungen an mehreren Stellen auf lokale Änderungen zurück führt.

Lemma 2. *Sei $V' \subseteq V$ eine Menge von Knoten und x' der Suchpunkt, der aus x entsteht, indem genau die Bits x_i mit $v_i \in V'$ flippen. Dann gilt*

$$f(x') - f(x) \leq \sum_{v_i \in V'} g(x, i) + |V'|(|V'| - 1)$$

Beweis. Für eine Kante $e = \{v_i, v_j\}$ notieren wir $f(x, e) := 1$ genau dann, wenn $x_i = x_j$, und $f(x, e) := 0$ sonst. Weiterhin sei $d(e) := f(x', e) - f(x, e)$. In der folgenden Rechnung nutzen wir aus, dass die Endpunkte von zu v_i inzidenten Kanten $\{v_i, v_j\}$ in x^i genau so gefärbt sind wie in x', falls $v_j \notin V'$. Dann können wir $g(x, i)$ für $v_i \in V'$ schreiben als

$$\begin{aligned} g(x,i) &= \sum_{v_j \in \mathrm{Adj}(v_i)} \big(f(x^i, \{v_i, v_j\}) - f(x, \{v_i, v_j\})\big) \\ &= \sum_{v_j \in \mathrm{Adj}(v_i) \setminus V'} \big(f(x^i, \{v_i, v_j\}) - f(x, \{v_i, v_j\})\big) + \\ &\quad \sum_{v_j \in \mathrm{Adj}(v_i) \cap V'} \big(f(x^i, \{v_i, v_j\}) - f(x, \{v_i, v_j\})\big) \\ &\geq \sum_{v_j \in \mathrm{Adj}(v_i) \setminus V'} d(\{v_i, v_j\}) - |V'| + 1 \end{aligned}$$

Wir nutzen aus, dass sich x und x' nur auf Kanten unterscheiden, die zu genau einem Knoten in V' inzident sind. Für alle anderen Kanten e gilt $d(e) = 0$.

$$\begin{aligned} f(x') - f(x) &= \sum_{e \in E} \big(f(x', e) - f(x, e)\big) = \sum_{e \in E} d(e) \\ &= \sum_{v_i \in V'} \sum_{v_j \in \mathrm{Adj}(v_i) \setminus V'} d(\{v_i, v_j\}) \\ &\leq \sum_{v_i \in V'} \big(g(x, i) + |V'| - 1\big) \\ &= \sum_{v_i \in V'} g(x, i) + |V'|(|V'| - 1) \end{aligned}$$

□

Theorem 2. *Unter der Annahme, dass der (1+1) EA auf dem Ising-Modell auf G^{stairs} mit einer durch eine Konstante $c > 0$ nach unten beschränkten Wahrscheinlichkeit einen Punkt auf der ersten Hälfte eines Pfades erreicht, ist die erwartete Zeit, bis der (1+1) EA einen optimalen Punkt findet, nach unten beschränkt durch $\Omega(n^2)$.*

Beweis. Da die Ising-Funktion spin-flip symmetrisch ist, betrachten wir o. B. d. A. nur einen der beiden Pfade. Weiterhin gehen wir davon aus, dass die folgenden Ereignisse eintreten. Falls eines dieser Ereignisse nicht eintritt, schätzen wir die erwartete Optimierungszeit mit der trivialen Schranke 0 ab.

E_1: Der (1+1) EA erreicht einen Punkt $x = 1^i 0^{n/2-i} 1^{n/2}$ mit $0 \leq i \leq n/8$. Die Zeit bis zum Erreichen dieses Punktes schätzen wir mit 0 ab.

E_2: Es kommt in polynomiell vielen Schritten nicht vor, dass in einem Schritt $\Omega(n^{1/3})$ Bits flippen.

E_1 hat nach Voraussetzung eine Wahrscheinlichkeit von mindestens c, $\text{Prob}(\overline{E_2})$ lässt sich durch $\text{poly}(n) \cdot 1/(n^{1/3})! = \text{poly}(n) \cdot 2^{-\Omega(n^{1/3})}$ abschätzen. Damit lässt sich $\text{Prob}(E_1 \cap E_2)$ nach unten abschätzen durch $1 - (\text{Prob}(\overline{E_1}) + \text{Prob}(\overline{E_2})) = \Omega(1)$.

Wir zeigen, dass der (1+1) EA unter den genannten Annahmen genügend lange auf dem Pfad verbleibt und daher im Erwartungswert $\Omega(n^2)$ Generationen benötigt, um die ersten drei Viertel des Pfades zu verlassen.

Wenn der aktuelle Suchpunkt x der i-te Punkt des Pfades ist, $0 \leq i \leq (3/16)n$, und der Nachkomme x' aus x durch Flips von $o(n^{1/3})$ Bits entsteht und V' die Menge flippender Knoten bezeichnet, können die drei folgenden Fälle eintreten.

Fall 1 $V' \cap \{v_{n/4}, \ldots, v_{n-1}\} \neq \emptyset$

Wir zeigen, dass ein Schritt, bei dem Bits flippen, die nicht zum Pfad gehören, nicht akzeptiert wird. Für alle Bits x_j mit $n/4 \leq j < n$ gilt $g(x,j) \leq -(1/8)n + 1$. Die maximale lokale Fitnessverbesserung über alle Knoten j ist $g(x,j) = 1$. Nach Lemma 2 ist damit

$$\begin{aligned} f(x') - f(x) &\leq \sum_{v_l \in V'} g(x,l) + |V'|(|V'|-1) \\ &\leq -(1/8)n + 1 + \sum_{v_l \in V',\ l \neq j} g(x,l) + |V'|(|V'|-1) \\ &\leq -(1/8)n + 1 + (|V'|-1) + o(n^{2/3}) = -\Omega(n). \end{aligned}$$

Der Suchpunkt x' wird also nicht akzeptiert.

Fall 2 $V' \subseteq \{v_0, \ldots, v_{n/4-1}\}, v_i \notin V'$

Wir zeigen, dass ein Schritt mit flippenden Bits auf dem Pfad nicht akzeptiert wird, wenn dabei x_i nicht flippt. Für alle Punkte x', die wir dabei durch jeweilige V' erzeugen können, ist eine wichtige Beobachtung, dass die Fitness bei fester Zahl an Einsen maximal ist, wenn alle Einsen im Bitstring möglichst weit links stehen.

Sei also $S := \{x \in \{0,1\}^n \mid x_i = 0 \wedge x_{n/4} = \cdots = x_{n/2-1} = 0 \wedge x_{n/2} = \cdots = x_{n-1} = 1\}$ die Menge aller möglichen x', die durch $V' \subseteq \{v_0, \ldots, v_{n/4-1}\}, v_i \notin V'$ aus x entstehen können. Für $0 \leq l < n/4$ seien z^l die Punkte aus S, die auf den ersten $n/4$ Positionen l Einsen enthalten und bei denen die Einsen möglichst weit links stehen unter der Nebenbedingung $z^l_i = 0$. Formal:

$$\begin{aligned} z^0 &:= 0^{n/2}1^{n/2} \\ &\ \vdots \\ z^i &:= 1^i 0^{n/2-i} 1^{n/2} \\ z^{i+1} &:= 1^i 010^{n/2-i-2} 1^{n/2} \\ &\ \vdots \\ z^{n/4-1} &:= 1^i 01^{n/4-i-1} 0^{n/4} 1^{n/2} \end{aligned}$$

Wenn wir mit $s(x)$ die Zahl der Einsen auf den ersten $n/4$ Bitpositionen von x bezeichnen, dann gilt

$$\forall x^* \in S: \; f(x^*) \leq f(z^{s(x^*)})$$

Dies ist leicht einzusehen: ein Punkt $x^* \neq z^{s(x^*)}$ enthält zwei Bitpositionen $j < k, j \neq i$, bei denen eine Null vor einer Eins steht, $x^*_j = 0, x^*_k = 1$. Wenn wir diese beiden Bits vertauschen, bleibt die Zahl der Einsen konstant, die Fitness erhöht sich aber um $2(k-j)$, da v^*_j zu $2(k-j)$ mehr Brückenkanten inzident ist und alle Knoten der zweiten Clique 1-gefärbt sind. Durch Iteration können wir somit x^* in $z^{s(x^*)}$ überführen und die Fitness erhöhen.

Wenn wir also zeigen können, dass die Fitness aller $z^l, l \neq i$ kleiner ist als die Fitness des aktuellen Suchpunkts $x = z^i$, folgt daraus, dass kein Suchpunkt aus S akzeptiert wird.

Für z^l und $l < i$ ist z^l der l-te Punkt auf dem Pfad und da die Fitnesswerte auf dem Pfad streng monoton sind, gilt $f(z^l) < f(z^i)$ für $l < i$.

Damit der Punkt z^l mit $l > i$ aus x erzeugt wird, müssen $k := |V'| = l - i$ Bits flippen. Für alle Bitpositionen $i < j < n/4$ gilt $g(x, j) = -2(j - i) + 1$. Die Fitnessdifferenz zwischen z^l und x lässt sich damit wie folgt nach Lemma 2 abschätzen.

$$\begin{aligned} f(z^l) - f(x) &\leq \sum_{d=1}^{k}(-2d+1) + |V'|(|V'|-1) \\ &= k - 2\sum_{d=1}^{k} d + k(k-1) \\ &= k - k(k+1) + k(k-1) \\ &= -k \end{aligned}$$

Die Punkte z^l für $l > i$ werden daher auch nicht akzeptiert und insgesamt folgt, dass der (1+1) EA keinen anderen Suchpunkt als x aus S akzeptiert.

Fall 3 $\quad V' \subseteq \{v_0, \ldots, v_{n/4-1}\}, v_i \in V'$

Falls x_i flippt, erhöht dieser Bit-Flip die Fitness um 1 und falls im gleichen Schritt andere Bits flippen, können Schritte akzeptiert werden, die zusammen betrachtet eine Fitnessveränderung von mindestens -1 ergeben. Wir betrachten nun die Anwendung der einzelnen Bit-Flips in einem Mutationsschritt nacheinander und addieren die Fitnessveränderungen aller Bit-Flips. Als erstes flippen wir x_i. Dadurch erreichen wir den $(i+1)$-ten Punkt auf dem Pfad und falls $i + 1 \leq (3/16)n$ gilt, wiederholen wir nun unsere Betrachtungen mit $V'' := V' \setminus \{v_i\}$.

Da das Flippen von x_i die Fitness bereits um 1 erhöht hat, akzeptiert der (1+1) EA diesen Schritt, wenn alle weiteren Bit-Flips zusammen die Fitness um mindestens -1 erhöhen. Da x_i in einem Schritt nicht zwei Mal geflippt wird, kommen nur Mengen von Bit-Flips in Frage, für die gilt: $V'' = \{v_{i+2}\}$ (aus Fall 2) oder $v_{i+1} \in V''$ (aus Fall 3), da alle übrigen V'' die Fitness um mehr als 1 senken.

Einen Schritt, in dem x_i flippt, bezeichnen wir als *erfolgreich*. Falls die genannten beiden Möglichkeiten von Bit-Flips in einem erfolgreichen Schritt auftreten, schätzen wir die erwartete Optimierungszeit ab durch die Zahl der bisher benötigten Generationen, d. h. wenn

in einem Schritt x_i und x_{i+1} oder x_i und x_{i+2} flippen, sehen wir den Optimierungsprozess als beendet an, da wir die Auswirkungen dieser Bit-Flips nicht kontrollieren können. Dieses Ereignis bezeichnen wir mit E_3.

Die bedingten Wahrscheinlichkeiten, dass x_{i+1} bzw. x_{i+2} flippt unter der Bedingung, dass x_i flippt, sind jeweils $1/n$. Damit gilt $\text{Prob}(E_3) \leq 2/n$. Da wir, falls E_3 nicht eintritt, insgesamt noch mindestens $n/16$ erfolgreiche Schritte benötigen, um die ersten drei Viertel des Pfades zu verlassen, folgt eine erwartete Zahl erfolgreicher Schritte, bis die ersten drei Viertel des Pfades verlassen werden oder E_3 eintritt von mindestens

$$\begin{aligned}
\text{E}(T(n)) &\geq \sum_{t=1}^{n/16} t \cdot (1 - \text{Prob}(E_3))^{t-1} \cdot \text{Prob}(E_3) \\
&\geq \frac{2}{n} \sum_{t=1}^{n/16} t \cdot \left(1 - \frac{2}{n}\right)^{t-1} \\
&\geq \frac{2}{n} \sum_{t=1}^{n/16} t \cdot \left(1 - \frac{2}{n}\right)^{\frac{n}{2}-1} \\
&\geq \frac{2}{n} \sum_{t=1}^{n/16} t \cdot e^{-1} \geq \frac{2}{en} \cdot (n/16)^2 = \frac{1}{128 \cdot e} \cdot n
\end{aligned}$$

Ein erfolgreicher Schritt hat eine Wahrscheinlichkeit von höchstens $1/n$, daher folgt die untere Schranke von $\Omega(1) \cdot n \cdot T(n) = \Omega(n^2)$ erwarteten Schritten. □

Ergebnisse der Erkundungsexperimente

Das Ziel der ersten Erkundungsexperimente war es, für den (1+1) EA und die (1+1) RLS den Zusammenhang zu untersuchen zwischen der Anzahl bzw. der Anordnung der Brückenkanten und der Wahrscheinlichkeit, dass beide Cliquen mit der gleichen Farbe gefärbt werden. Wir haben Experimente für zwei Cliquen mit je 50 Knoten bzw. je 75 Knoten durchgeführt und jeweils die Anzahl m der Brückenkanten von 0 bis zur maximalen Anzahl 2500 bzw. 5625 kleinschrittig variiert. Es wurden Experimente mit drei verschiedenen Strategien zur Anordnung der Brückenkanten durchgeführt:

- Zufällig: Eine Teilmenge der Menge aller möglichen Brückenkanten mit m Elementen wird uniform zufällig aus der Menge aller Teilmengen mit m Elementen ausgewählt.
- Gleichmäßig: Die Idee ist, die Brückenkanten gleichmäßig auf die Knoten aufzuteilen. Bei m zu verteilenden Brückenkanten erhält also jeder Knoten entweder $\lceil \frac{2m}{n} \rceil$ oder $\lfloor \frac{2m}{n} \rfloor$ Nachbarn in der anderen Clique.
- Konzentriert: So wenig Knoten wie möglich sollen Nachbarn in der anderen Clique haben. Bei m zu verteilenden Brückenkanten erhalten also nur $\lceil \sqrt{m} \rceil$ viele Knoten in jeder Clique Nachbarn in der anderen Clique.

Wir haben dann beobachtet, wie groß die Wahrscheinlichkeit ist, dass ein globales Optimum erreicht wird, ohne vorher ein lokales Optimum erreicht zu haben. Diese Wahrscheinlichkeit werden wir im Folgenden auch als Erfolgswahrscheinlichkeit bezeichnen. Um Experimente mit verschiedenen Cliquengrößen miteinander vergleichen zu können, haben wir uns die Erfolgswahrscheinlichkeit in Abhängigkeit von dem Quotienten aus m und der maximalen Anzahl an Brückenkanten, also $n^2/4$ bei zwei Cliquen mit je $n/2$ Knoten, angeschaut. Wir stellten fest, dass die Kurven, die sich für 50 und 75 Knoten pro Clique ergaben, in jedem getesteten Szenario in etwa übereinstimmten, was die folgende Hypothese motivierte.

Hypothese 2. *Für den (1+1) EA und die (1+1) RLS und für jede der drei Strategien zur Anordnung der Brückenkanten ist die Erfolgswahrscheinlichkeit nur abhängig von dem Quotienten $q(m,n)$ aus der Anzahl der Brückenkanten m und der maximalen Anzahl $n^2/4$ an Brückenkanten.*

Ferner verglichen wir das Verhalten der Algorithmen auf Graphen, bei denen die Brückenkanten zufällig gewählt wurden, mit dem Verhalten auf Graphen mit gleichmäßig verteilten Brückenkanten und stellten für beide Algorithmen fest, dass beide Strategien zur Kantenauswahl zu sehr ähnlichen Ergebnissen führen.

Hypothese 3. *Für den (1+1) EA und die (1+1) RLS ist die Erfolgswahrscheinlichkeit für jede feste Anzahl an Brückenkanten und für jede Cliquengröße unabhängig davon, ob die Brückenkanten gleichmäßig oder zufällig verteilt werden.*

Die Erfolgswahrscheinlichkeiten bei konzentrierten Brückenkanten verhielten sich anders als bei zufälligen oder gleichmäßigen Brückenkanten. Folgende Hypothesen ergaben sich.

Hypothese 4. *Für den (1+1) EA ist die Erfolgswahrscheinlichkeit für jede Cliquengröße bei konzentrierten Brückenkanten größer als bei zufällig oder gleichmäßig verteilten Brückenkanten, falls die Anzahl der Brückenkanten nicht zu klein ist und nicht so groß, dass fast immer eine Gleichfärbung beobachtet wird.*

Hypothese 5. *Für die (1+1) RLS ist die Erfolgswahrscheinlichkeit für jede Cliquengröße bei konzentrierten Brückenkanten für wenige Brückenkanten kleiner als bei zufällig oder gleichmäßig verteilten Brückenkanten. Erhöht man die Anzahl der Brückenkanten, so beobachtet man, dass die Erfolgswahrscheinlichkeit bei konzentrierten Brückenkanten zunächst größer wird als bei zufällig oder gleichmäßig verteilten Brückenkanten, aber ab einer bestimmten Anzahl an Brückenkanten wieder darunter fällt.*

Weiterhin verglichen wir die Erfolgswahrscheinlichkeiten des (1+1) EA und die der (1+1) RLS. In den Erkundungsexperimenten stellte sich heraus, dass der (1+1) EA oft eine höhere Erfolgsrate erreichte als die (1+1) RLS. Daraus resultiert folgende Hypothese.

Hypothese 6. *Für jede der drei Strategien zur Anordnung der Brückenkanten und für jede Cliquengröße ist die Erfolgswahrscheinlichkeit des (1+1) EA größer als die der (1+1) RLS, sofern die Anzahl der Brückenkanten nicht zu klein ist und nicht so groß, dass bei beiden Algorithmen fast immer eine Gleichfärbung beobachtet wird.*

Erfolgsvorhersage mittels Potenzialfunktionen

Die Beobachtungen und Überlegungen in Abschnitt 8.1.1 haben die Vermutung nahe gelegt, dass der (1+1) EA relativ schnell und geradlinig zu einem lokalen Optimum gelangt, sobald eine entsprechende Mehrheit an Farben vorhanden ist.

Eine Funktion mit ähnlichem Verhalten wurde bereits vollständig untersucht: Wegener und Witt [47] definieren die quadratische Funktion $\mathrm{CH}_k(y, x)$ („Choice") auf einem Bit y und n Bits $x = (x_1, \ldots, x_n)$. Abhängig von der Belegung von y entspricht die Funktion $\mathrm{CH}_{k|y}$ bei passender Wahl des Parameters k im Wesentlichen den Funktionen OneMax oder −OneMax, d. h. abhängig von y soll die Zahl der Einsen in x entweder maximiert oder minimiert werden. Beide Subfunktionen sind affin transformiert, so dass es nur ein globales Optimum gibt und dass sie sich bei $||x||_1 = n/2$ schneiden.

Wenn wir die Ising-Funktion auf Graphen G mit nur einer Brückenkante betrachten, haben die Ising-Funktion auf G und CH_k folgende Gemeinsamkeiten.

1. Beide Funktionen sind annähernd spin-flip symmetrisch und besitzen jeweils gleich viele rein lokale und globale Optima. Alle Optima haben einen linearen Hammingabstand voneinander.

2. Bei beiden Funktionen sind die Wahrscheinlichkeiten, in ein rein lokales oder ein globales Optimum zu laufen, jeweils ungefähr 1/2.

3. Die Fitnesslandschaften sind multimodal und bieten fast keine Plateaus; beide Funktionen geben gute Hinweise in Richtung des nächst gelegenen Optimums.

Wenn wir Graphen mit mehr Brückenkanten betrachten, verändert sich nur die Wahrscheinlichkeit, in ein rein lokales Optimum zu laufen; die anderen Charakteristika bleiben erhalten, so lange sich die Menge der lokalen Optima nicht verändert.

Wegener und Witt zeigen für den (1+1) EA auf CH_k, dass sich in einer Anfangsphase von $n^{3/4}$ entscheidet, in welche Richtung der Optimierungsprozess läuft: der (1+1) EA startet mit einer erwarteten Zahl von $n/2$ Einsen in x und nach $n^{3/4}$ Schritten hat der (1+1) EA mit einer Wahrscheinlichkeit von $1 - o(1)$ einen Suchpunkt erzeugt, in dem x entweder weniger als $n/2 - n^{2/3}$ oder mehr als $n/2 + n^{2/3}$ Einsen enthält. Wenn eine solche Mehrheit an Nullen bzw. Einsen vorherrscht, kann der (1+1) EA diese Mehrheit nur noch dadurch verringern, dass in einem Schritt die Mehrheit auf höchstens $n/2$ Nullen bzw. Einsen verringert wird; ein Ereignis, das nur mit exponentiell kleiner Wahrscheinlichkeit auftritt.

Aufgrund der genannten Ähnlichkeiten zwischen den beiden Fitnesslandschaften vermuten wir ein ähnliches Ergebnis für den (1+1) EA auf dem Ising-Modell auf zwei verbundenen Cliquen. Wir vermuten, dass nach einer kleinen Anfangsphase der (1+1) EA einen Suchpunkt erreicht hat, in dem in jeder Clique jeweils eine Farbe mit genügend großer Mehrheit vorherrscht, so dass nach dieser Anfangsphase das Ergebnis des Laufs bereits mit hoher Wahrscheinlichkeit feststeht, wobei der Lauf abgebrochen wird, sobald alle Cliquen jeweils einheitlich gefärbt sind.

Diese Vermutung soll nun zu einer überprüfbaren Hypothese konkretisiert werden, die dann experimentell untersucht wird.

Dazu definieren wir eine Potenzialfunktion, die die Färbung der Knoten misst:

$$\varphi(x) := ||x||_0 - n/2.$$

Bei der Initialisierung ist $\mathrm{E}(\varphi(x)) = 0$ und für Suchpunkte, in denen beide Cliquen jeweils einheitlich mit verschiedenen Farben gefärbt sind, gilt $\varphi(0^{n/2}1^{n/2}) = \varphi(1^{n/2}0^{n/2}) = 0$. Die beiden einheitlichen Färbungen des gesamten Graphen haben die Werte $\varphi(1^n) = -n/2$ und $\varphi(0^n) = +n/2$.

In Experimenten konnte beobachtet werden, dass sich der Wert der Potenzialfunktion über die Zeit entweder bei Werten nahe 0 bewegt oder sich sehr monoton verändert, je nachdem, ob in beiden Cliquen die gleiche Farbe vorherrscht oder ob sich unterschiedliche Färbungen entwickeln.

Aus dieser Beobachtung entstand ein Test, der schon früh vorhersagt, ob der Lauf erfolgreich sein wird: nach $n^{3/4}$ Generationen wird für den aktuellen Suchpunkt x getestet, ob $|\varphi(x)| \geq n^{3/4}/8$ ist. Der Exponent 3/4 ist einerseits groß genug, um zuverlässige Vorhersagen zu ermöglichen und gleichzeitig klein genug, um einen greifbaren Vorteil gegenüber der erwarteten Laufzeit zu bieten. Die Schranke $n^{3/4}/8$ hat sich in vorab durchgeführten Erkundungsexperimenten als geeignet erwiesen.

Da nach unseren Vorbetrachtungen wenige Brückenkanten keinen großen Einfluss haben, konzentrieren wir uns dabei nur auf quadratische Zahlen von Brückenkanten.

Hypothese 7. *Sei G ein Graph mit $\Theta(n^2)$ Brückenkanten und x der Suchpunkt, den der (1+1) EA auf dem Ising-Modell auf G nach $n^{3/4}$ Generationen erreicht. Dann gilt mit Wahrscheinlichkeit $1 - o(1)$: $|\varphi(x)| \geq n^{3/4}/8 \Leftrightarrow$ der (1+1) EA erreicht ein globales Optimum 0^n oder 1^n, ohne einen Punkt $0^{n/2}1^{n/2}$ oder $1^{n/2}0^{n/2}$ zu erreichen.*

Brückenbasierte Erfolgsvorhersage

In unsere ersten theoretische Überlegungen, Lemma 1 und Korollar 1, ging die genaue Färbung der Brückenkanten nicht ein; der Einfluss der Brückenkanten wurde lediglich grob abgeschätzt. Wenn die maximale Anzahl an Brückenkanten pro Knoten durch einen kleinen Wert d beschränkt ist, wenn also wenige Brückenkanten vorhanden sind, erhalten wir mit dieser Strategie aussagekräftige Resultate.

Falls der Graph jedoch viele Brückenkanten enthält oder falls einzelne Knoten eine hohe Anzahl von Brückenkanten haben, ist es für eine theoretische Analyse wenig sinnvoll, die Färbung der Brückenkanten außer Acht zu lassen. Statt dessen bietet es sich an, eine weitere Potenzialfunktion zu untersuchen: den Anteil korrekt gefärbter Brückenkanten.

$$\psi(x) := \frac{\text{Zahl korrekt gefärbter Brückenkanten}}{\text{Zahl der Brückenkanten}}$$

Den Brückenkanten kommen hier zweierlei Funktionen zu: zum einen beeinflussen die Brückenkanten die Fitness. Insbesondere wenn die Zahl der Brückenkanten $\Theta(n^2)$ ist und die Kanten gleichmäßig oder zufällig verteilt sind, ist jeder Knoten zu einer erwartet linearen Zahl von Brückenkanten inzident, so dass der Einfluss der Brückenkanten nur um einen konstanten Faktor kleiner ist als der Einfluss der Knoten innerhalb der Clique.

Zum anderen fungieren Brückenkanten als Indikatoren dafür, wie ähnlich die beiden Cliquen gefärbt sind. Bei zufälliger Initialisierung ist $\mathrm{E}(\psi(x)) = 1/2$, für verschieden farbige Cliquen gilt $\psi(0^{n/2}1^{n/2}) = \psi(1^{n/2}0^{n/2}) = 0$ und für einheitliche Färbungen gilt $\psi(0^n) = \psi(1^n) = 1$. Anders als bei der Knotenfärbung φ gibt es hier also zwei Richtungen, in die sich der Wert von ψ nach der Initialisierung entwickeln kann.

Um den Einfluss und die Aussagekraft dieser Potenzialfunktion zu untersuchen und damit vielleicht eine spätere theoretische Analyse zu erleichtern, wurden auch mit dieser Potenzialfunktion Tests durchgeführt. Nach einer Anfangsphase von $n^{3/4}$ Schritten wurde getestet, ob $\psi(x) > 1/2$ ist. Falls ja, ist die Mehrheit der Brückenkanten korrekt gefärbt und wir sagen deshalb einen Erfolg des Laufs voraus. Anderenfalls wird ein Misserfolg vorhergesagt.

Auch bei diesem Test vermuten wir, dass die Wahrscheinlichkeit einer falschen Vorhersage mit wachsendem n gegen 0 konvergiert.

Hypothese 8. *Sei G ein Graph mit $\Theta(n^2)$ Brückenkanten und x der Suchpunkt, den der (1+1) EA auf dem Ising-Modell auf G nach $n^{3/4}$ Generationen erreicht. Dann gilt mit Wahrscheinlichkeit $1 - o(1)$: $\psi(x) > 1/2 \Leftrightarrow$ der (1+1) EA erreicht ein globales Optimum 0^n oder 1^n, ohne einen Punkt $0^{n/2}1^{n/2}$ oder $1^{n/2}0^{n/2}$ zu erreichen.*

8.1.2 Das Ising-Modell auf Tori

Wie schon bei den Cliquen, so betrachten wir auch hier das Ising-Modell mit nur 2 Farben. Allerdings wählen wir als zugrunde liegenden Graphen den zweidimensionalen Torus. Dieser besteht aus Knoten, die in einer regelmäßigen Gitterstruktur (entsprechend einer Matrix) angeordnet und horizontal und vertikal – oder zusätzlich noch diagonal – mit ihren direkten Nachbarn verbunden sind. Dementsprechend hat jeder Knoten exakt Grad 4 bzw. 8.

Es gibt natürlich wiederum zwei triviale Maxima, die Färbungen 0^n und 1^n, wobei die Codierung als Bitstring kanonisch zeilenweise geschieht. In der der Seminarphase zugrunde liegenden Diplomarbeit von Simon Fischer [10] wurde vor allem das Modell mit Diagonalkanten (also Grad 8) untersucht, wobei die Vermutung geäußert wurde, dass es keinen gravierenden Unterschied zwischen den Tori mit und den Tori ohne Diagonalkanten gäbe. Unsere Experimente sollten dies bestätigen bzw. widerlegen.

Es ist bekannt, dass der (1+1) EA eine lange erwartete Laufzeit hat, wenn er während typischer Läufe viele Bits auf einmal flippen muss, es also gut erreichbare lokale Optima gibt, die nicht einfach zu verlassen sind. In der Diplomarbeit wurde gezeigt, dass es solche in der Tat gibt: stabile Ringe. Diese sind in Abbildung 8.2 beispielhaft dargestellt (wobei Typen 3a und 3b nur mit Diagonalkanten stabil sind). Aufgrund der Symmetrie des Suchraums ist es relativ leicht zu sehen, dass sich horizontale von vertikalen Ringen nicht unterscheiden, sie also insbesondere mit gleicher Wahrscheinlichkeit auftreten.

In dieser Situation muss der (1+1) EA $O(\sqrt{n})$ Kanten flippen, um einen Fortschritt zu erzielen, was im Erwartungswert Zeit $\Omega(n^{\sqrt{n}})$ benötigt. Weiterhin hat Simon Fischer gezeigt, dass man im Erwartungswert nach $O(n^3)$ Generationen entweder die Funktion optimiert hat oder in einer solchen Situation fest steckt.

Ungeklärt blieben aber unter anderem folgende Fragestellungen:

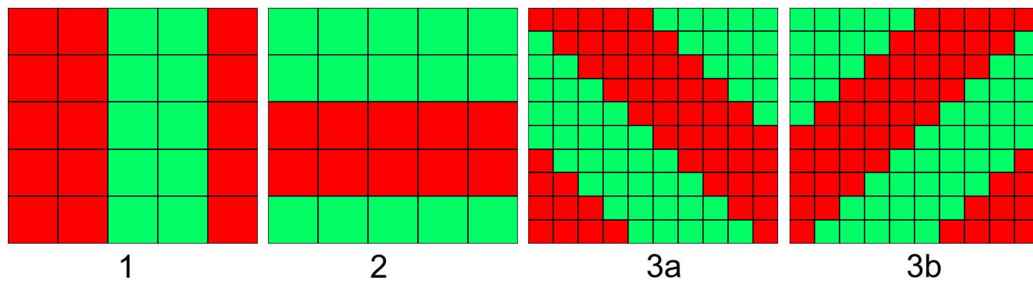


Abbildung 8.2: Die verschiedenen stabilen Ringtypen.

1. Wie groß ist die Wahrscheinlichkeit in einem lokalen Optimum hängen zu bleiben, das kein globales Optimum ist?
2. Ist diese Wahrscheinlichkeit unabhängig von der Suchraumdimension?
3. Was ändert sich beim Übergang zwischen Tori mit und ohne Diagonalkanten?
4. Unterscheidet sich beim Ising-Modell mit Diagonalkanten die Wahrscheinlichkeit, einen diagonalen Ring zu erreichen, von der Wahrscheinlichkeit, einen horizontalen oder vertikalen Ring zu erreichen?
5. Benötigt der (1+1) EA wirklich $O(n^3)$ Generationen bis zu einem stabilen Zustand?

Zunächst widmeten wir uns der Frage nach der Wahrscheinlichkeit, mit der stabile Ringe auftreten. Um die Anzahl der Parameter möglichst gering zu halten, beschränkten wir uns hier zunächst auf quadratische Tori. Eine erste Experimentreihe mit Tori verschiedener Kantenlänge führte unmittelbar zu einer ersten Hypothese im Bezug auf diese Fragestellung.

Hypothese 9. *Die Wahrscheinlichkeit, einen Ring zu erreichen, ist bei quadratischen Tori konstant, also unabhängig von der Suchraumdimension. Dies gilt für Tori mit und ohne Diagonalkanten.*

Unserer Intuition (und den bereits gemachten Experimenten) folgend stellten wir eine Hypothese über die zu erwartende Ringbreite auf.

Hypothese 10. *Die erwartete Breite der Ringe ist linear in der Kantenlänge des Torus, d. h.*

$$\mathrm{E}(\textit{Breite des Rings}) = c \cdot \sqrt{n},$$

wobei c eine geeignete Konstante ist. Dies gilt für Tori mit und ohne Diagonalkanten.

In Bezug auf diagonale Ringe führten unsere Experimente zu einer weiteren Beobachtung, die sich in der folgenden Hypothese wiederfindet.

Hypothese 11. *Auf dem Torus mit diagonalen Kanten ist die Wahrscheinlichkeit, einen diagonalen Ring zu erreichen, signifikant kleiner als die Wahrscheinlichkeit, einen horizontalen oder vertikalen Ring zu erreichen.*

Entgegen der ersten Intuition ist der Torus mit Diagonalkanten nicht völlig rotationssymmetrisch. Diagonale Ringe unterscheiden sich also maßgeblich von horizontalen bzw. vertikalen Ringen. Man kann dies folgendermaßen einsehen. Betrachten wir den Rand eines solchen Ringes – genauer gesagt die Anzahl an Knoten, die einen Nachbarn anderer Farbe haben. Bei horizontalen und vertikalen Ringen (mit Breite mindestens 4) ist diese $4d$, wobei d die Kantenlänge des quadratischen Torus ist. Diese Anzahl ist jedoch bei diagonalen Ringen (auch Breite mindestens 4) $8d$. Vergleiche dazu Abbildung 8.3.

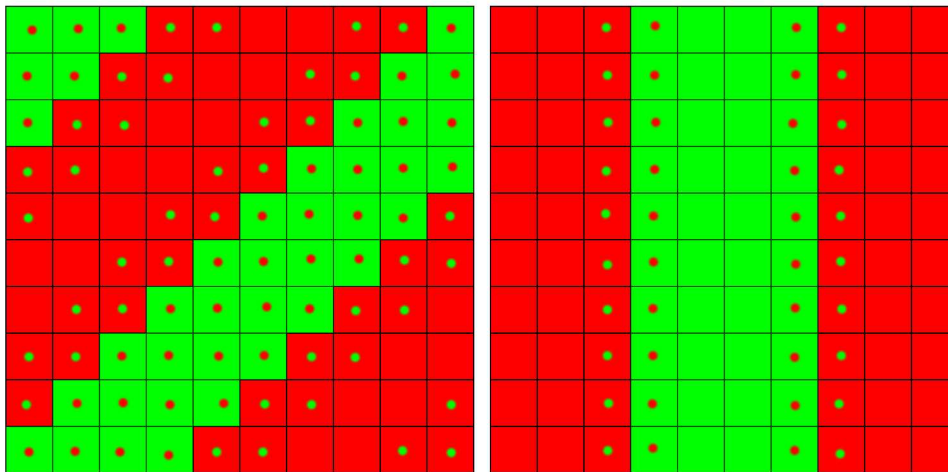


Abbildung 8.3: Nachbarschaften bei Ringen.

Hypothese 12. *Der (1+1) EA erreicht auf dem quadratischen Torus bereits nach $O(n^{3/2})$ Generationen einen stabilen Zustand, also entweder ein globales Optimum oder einen stabilen Ring.*

Während unserer Experimente beobachteten wir auch weitere Phänomene, die sich allerdings nur schwer in Hypothesen fassen lassen und vermutlich auch schwer zu untersuchen sind. Beispielsweise ergaben sich in den Laufzeiten der Läufe auf Tori ohne Diagonalkanten ab und an Ausreißer, die extrem hohe Generationenzahlen bis zum Optimum benötigten. Sie benötigten mitunter auch 5- bis 6-mal so lange wie die durchschnittlichen Läufe. Eine etwas genauere Betrachtung dieser Läufe zeigte, dass stets ein Plateau erheblicher Größe betreten wurde. Der aktuelle Suchpunkt war zu diesem Zeitpunkt einem diagonalen Ring nicht unähnlich. Es zog sich ein diagonales Band durch den Torus. Die Ränder dieses Bandes waren in ständiger Bewegung, es flippten also immer wieder einige Bits gleichzeitig, so dass der Funktionswert nicht verändert wurde. Erst nach einiger Zeit gelang es dem (1+1) EA, diese Situation aufzuheben, indem der „Ring“ aufgebrochen wurde.

Ein weiteres Phänomen betrifft die noch grob untersuchten nicht-quadratischen Tori. Ein Versuch auf einem Torus mit unterschiedlich langen Seiten (im Verhältnis von 1:4) ergab, dass die Ringwahrscheinlichkeit auf 88 Prozent stieg und das Auftreten mehrerer paralleler Ringe nicht unwahrscheinlich wurde. Im Gegensatz hierzu haben wir immer nur einen Ring bei quadratischen Tori beobachtet.

8.1.3 Das Ising-Modell auf booleschen Hypercubes

Die in der Seminarphase bezüglich der Ising-Funktion behandelten Graphen umfassten Ringe und Tori. Es lag also nahe, als Verallgemeinerung auch höherdimensionale Tori zu betrachten. Die strukturell einfachsten dieser Tori erhält man, indem man ihre Kantenlänge durch 2 begrenzt. Dies entspricht gerade den booleschen Hypercubes beliebiger Dimension.

Im Gegensatz zu den vorhergehenden Experimenten zum Ising-Modell ist uns keine Arbeit bekannt, die sich mit der Analyse des Verhaltens von evolutionären Algorithmen auf der zweifarbigen Ising-Funktion auf booleschen Hypercubes beschäftigt. Entsprechend basieren unsere Experimente zu diesem Thema auch nicht auf vorherigen theoretischen oder experimentellen Erkenntnissen.

Unsere ersten Experimente und Überlegungen basieren hier viel mehr auf den Ergebnissen, die für zweidimensionale quadratische Tori existieren. Analog zu diesen Ergebnissen ergeben sich sofort eine ganze Reihe von interessanten Fragen:

- Gibt es (analog zu Ringen auf dem Torus) stabile nichtoptimale Färbungen des Hypercubes?
- Welche Struktur haben diese Färbungen?
- Wie oft wird eine nichtoptimale stabile Färbung erreicht?
- Wie groß ist die Fitness in diesen stabilen Färbungen und wie schnell bricht der (1+1) EA die stabile Färbung wieder auf?

Natürlich spielt bei all diesen Fragen die Dimension des Hypercubes eine Rolle. Es ist wichtig, zwischen der Dimension des Suchraums und der Dimension des Hypercubes zu unterscheiden. Hat der Hypercube die Dimension d, so entspricht dies 2^d Knoten und damit der Suchraumdimension 2^d. Wenn wir von Dimension sprechen, meinen wir hier immer die Dimension des Hypercubes, nicht die des Suchraums.

Es ist hilfreich, sich zunächst ein Bild über die Struktur des Hypercubes der Dimension d zu machen. Jeder Knoten hat genau d Nachbarn. Wird die Farbe genau eines Knotens geflippt, wird dieser Flip genau dann akzeptiert, wenn höchstens die Hälfte der Nachbarknoten die gleiche Farbe hat wie der betrachtete Knoten vor seinem Flip. Sei $e(v)$ die Anzahl der gleichfarbigen Nachbarn des Knotens v. Ein Knoten v kann also nur mit einem akzeptierenden 1-Bit-Flip geflippt werden, wenn $e(v) \leq d/2$ gilt.

Wir widmen uns zunächst der Frage nach der Existenz stabiler nichtoptimaler Färbungen. Wir bezeichnen im Folgenden eine Färbung als stabil, wenn kein 1-Bit-Flip mehr möglich ist, der eine Fitnessverbesserung zur Folge hat. Dieses Vorgehen orientiert sich an den Ergebnissen unserer ersten Experimente. Es wäre natürlich nahe liegend, eine Färbung erst dann als stabil anzusehen, wenn keine 1-Bit-Flips mehr akzeptiert werden können, also auch solche Flips nicht mehr existieren, die die Fitness unverändert lassen. In Hypercubes gerader Dimension gibt es jedoch Plateaus, die in vielen Fällen erreicht werden. Einige dieser Plateaus können durch eine Folge von akzeptierenden 1-Bit-Flips wieder verlassen werden, andere jedoch nicht. Diese zweite Art von Plateaus entsprechen also ebenfalls dem, was man anschaulich als ein lokales

Optimum verstehen könnte. Der Lauf muss in solchen Situationen gestoppt werden, da der (1+1) EA sonst unter Umständen sehr lange (länger als es die uns zu Verfügung stehende Rechenzeit erlaubt) auf dem Plateau verweilt. Die RLS würde ein solches Plateau nie mehr verlassen. Das Erkennen solcher Situationen dauerte im Allgemeinen zu lange, so dass wir uns entschlossen haben einen Lauf bereits in nach obiger Definition stabilen Färbungen zu stoppen. Dieses Stoppkriterium wurde in allen Experimenten verwendet.

Betrachten wir also die Frage nach der Existenz solcher stabiler Zustände. Nach obiger Überlegung zeichnen sich diese durch die Tatsache aus, dass für jeden Knoten v die Bedingung $e(v) \geq d/2$ erfüllt ist. Dies führt sofort zu einer Charakterisierung der Struktur einer ganzen Reihe stabiler Zustände.

Beobachtung 3. *Ist jeder Knoten v in einem Subwürfel gleicher Färbung der Dimension mindestens $d/2$ enthalten, so entspricht dies einer stabilen nichtoptimalen Färbung.*

Die Korrektheit dieser Beobachtung ist leicht einzusehen. Insbesondere ist festzuhalten, dass wir hier nicht von einer disjunkten Zerlegung in Subwürfel ausgehen können. Aus diesem Grunde erweist sich auch die Frage nach der Anzahl der so charakterisierten Färbungen als äußerst schwierig. Es stellt sich weiterhin die Frage, ob sich auf diese Art und Weise zugleich auch alle stabilen Färbungen beschreiben lassen. Abbildung 8.4 zeigt jedoch, dass dies nicht der Fall ist. Das kleinste Gegenbeispiel lässt sich für den Hypercube der Dimension 4 konstruieren.

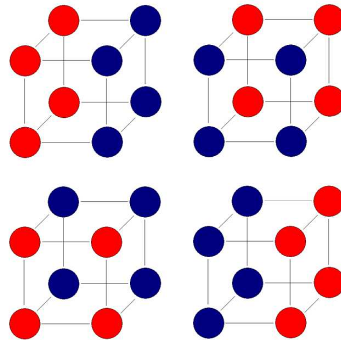


Abbildung 8.4: Stabile Färbung des Hypercube der Dimension 5 mit Subwürfeln der Dimension 2.

Um einen Eindruck davon zu bekommen, welcher Anteil stabiler Färbungen sich entsprechend Beobachtung 3 charakterisieren lässt, wurde FrEAK um die entsprechenden Module ergänzt. Implementiert wurden ein Stoppkriterium, dass den Lauf beim Erreichen eines stabilen Zustands stoppt, sowie ein Algorithmus, der zu jedem Knoten des Hypercube den umschließenden gleich gefärbten Subwürfel maximaler Dimension bestimmt.

Die Ergebnisse liegen hier nur für Hypercubes bis Dimension 12 vor, da der Zeitaufwand für die Berechnung der umschließenden Subwürfel bereits für Dimension 13 erheblich ist. Berücksichtigt sind nur Läufe, in denen das globale Optimum nicht erreicht wurde. Zu jeder Dimension wurden 1000 Läufe durchgeführt. Der (1+1) EA erreichte nur 4-mal einen stabilen

Zustand, der Subwürfel der Dimension kleiner als $d/2$ enthielt. In jedem der vier Fälle handelte es sich um Läufe auf dem Hypercube der Dimension 4. Die RLS erreichte 5-mal einen stabilen Zustand, der Subwürfel der Dimension kleiner als $d/2$ enthielt. Zwei dieser Läufe fanden ebenfalls auf dem Hypercube der Dimension 4 statt, zwei fanden auf dem Hypercube der Dimension 11 statt und einer auf dem Hypercube der Dimension 6. Die Mehrheit der erreichten stabilen Zustände scheint also tatsächlich entsprechend Beobachtung 3 charakterisiert zu sein.

Man könnte nun vermuten, dass diese Ergebnisse Rückschlüsse darüber zulassen, mit welcher Wahrscheinlichkeit der (1+1) EA stabile Zustände wieder aufbricht. Dies ist aber leider nicht der Fall. Mit einer leichten Variation unseres Algorithmus zur Suche maximaler Subwürfel haben wir Teile der stabilen nichtoptimalen Ergebnisse erneut untersucht, wobei diesmal eine Zerlegung in disjunkte gleich gefärbte Subwürfel berechnet wurde. Es wurde getestet, ob eine Fitnessverbesserung durch Flippen eines vollständigen Subwürfels erzielt werden konnte. Die Ergebnisse variieren sehr stark. So finden sich sowohl Individuen, bei denen durch einen 2-Bit-Flip Fitnessverbesserungen erzielt werden können, als auch solche, bei denen hierzu mindestens $2^{d/2}$ Bits flippen müssen.

Welcher Anteil aller Läufe erreicht nun aber überhaupt ein globales Optimum? Experimente mit dem (1+1) EA und der RLS zeigen deutlich, dass die Wahrscheinlichkeit, ein Optimum zu erreichen, mit wachsender Dimension abzunehmen scheint. Des Weiteren haben die ersten Experimente deutliche Unterschiede zwischen Hypercubes gerader und ungerader Dimension sowie zwischen dem (1+1) EA und der RLS gezeigt. Dies führte zu den drei Hypothesen 13, 14 und 15 und vor allem zur allgemein getrennten Betrachtung von geraden und ungeraden Dimensionen.

Hypothese 13. *Die Wahrscheinlichkeit, ein globales Optimum zu erreichen, sinkt mit wachsender Dimension.*

Hypothese 14. *Die Wahrscheinlichkeit, ein globales Optimum zu erreichen, ist für die RLS geringer als für den (1+1) EA.*

Hypothese 15. *Die Wahrscheinlichkeit, ein globales Optimum zu erreichen, ist für ungerade Dimensionen i geringer als für die geraden Dimensionen $i-1$ und $i+1$.*

Weiterhin stellt sich nun die Frage, wie nahe wir der Fitness eines optimalen Individuums kommen, wenn wir das Optimum nicht erreichen. Dazu definieren wir die Approximationsgüte als Quotient aus erreichtem Fitnesswert und dem optimalen Fitnesswert. Die ersten Experimente führten auch hier zu drei Hypothesen:

Hypothese 16. *Die Approximationsgüte steigt mit wachsender Dimension.*

Hypothese 17. *Die Approximationsgüte ist für die RLS geringer als für den (1+1) EA.*

Hypothese 18. *Die Approximationsgüte ist für ungerade Dimensionen i geringer als für die geraden Dimensionen $i-1$ und $i+1$.*

Neben den Fragestellungen, die am Anfang unserer Experimente standen, konnten wir zwei weitere interessante Beobachtungen machen. Zunächst stellte sich heraus, dass die Läufe

sowohl mit dem (1+1) EA als auch mit der RLS bei ungerader Dimension anscheinend länger waren als bei gerader Dimension.

Weiterhin zeigt sich, dass die RLS im Vergleich zum (1+1) EA offensichtlich schneller stabile Zustände erreicht. Da es nahe liegt, davon auszugehen, dass im Optimierungsprozess praktisch nur die 1-Bit-Flips relevant sind, gab es die Vermutung, dass der Faktor zwischen der Laufzeit des (1+1) EA und der Laufzeit der RLS e ist. Obwohl dieser Eindruck von den ersten Experimentdaten weiter gestärkt wurde, stellte sich schnell heraus, dass die Daten der Experimente nicht stabil genug waren um eine Hypothese zu formulieren, die sich auf diesen Faktor bezieht.

Hypothese 19. *Zum Erreichen eines stabilen Zustands benötigt die RLS weniger Generationen als der (1+1) EA.*

Zu guter Letzt sei noch bemerkt, dass es gewisse Anzeichen dafür gibt, dass Läufe, die zu einem Optimum führen, weniger Generationen benötigen als solche, die in einen suboptimalen stabilen Zustand führen und dort durch unser Stoppkriterium angehalten werden. Leider wurde auch hierzu keine Hypothese formuliert, da die ersten Experimente keinen deutlichen Unterschied erkennen ließen. Sollte sich dies aber doch noch bestätigen lassen, würde das darauf hindeuten, dass sich in aller Regel nicht erst kurz vor Ende eines Laufes entscheidet, ob dieser das Optimum erreicht oder nicht.

8.2 Minimale Spannbäume

Die Experimente zum Problem „Minimale Spannbäume“ (MST, minimal spanning trees) wurden durch den Vortrag motiviert, der in Abschnitt 7.1 beschrieben ist. Im Vortrag wurde die Laufzeit des (1+1) EA und der RLS auf dem Problem asymptotisch bestimmt. Man erhielt die gleiche obere Schranke für beide Algorithmen und für kleine Kantengewichte auch ein Worst-Case-Beispiel, für welches diese Laufzeit auch erreicht wird.

Es stellten sich verschiedene Fragen:

- Wie verhalten sich zufällige Graphen gegenüber den Worst-Case-Graphen aus dem Vortrag?
- Was ist der Einfluss von großen Kantengewichten?
- Gibt es konstante Faktoren oder additive Terme zwischen den Laufzeiten der beiden betrachteten Algorithmen (die ja asymptotisch gleich sind)?
- Inwieweit hat die Darstellung der Spannbäume, d. h. die Wahl des Suchraums, einen Einfluss auf die Laufzeit?

Im Folgenden werden die ersten Experimente zu den genannten Fragestellungen beschrieben. Alle Experimente wurden, sofern nicht anders beschrieben, mit der natürlichen Fitnessfunktion w' durchgeführt.

8.2.1 Experimente auf einem asymptotischen Worst-Case-Beispiel

In Abschnitt 7.1 wurde in der Analyse des (1+1) EA auf minimalen Spannbäumen eine Schranke von $O(m^2(\log(n)+\log(w_{\max}))$ für die erwartete Laufzeit des Algorithmus im Worst Case über alle Eingaben gezeigt. Dabei stellt m die Kantenzahl und damit auch die Dimension der Repräsentation von Individuen als Bitstrings dar, n die Knotenzahl und $w_{\max}$ das höchste auftretende Kantengewicht.

Es stellte sich automatisch die Frage, ob diese Schranke bis auf konstante Faktoren exakt ist. Zu dieser Frage konnte eine Graphklasse, in der PG auch *Ingo's Graph* genannt, gefunden werden, die eine Laufzeit von $\Theta(m^2 \log(n))$ erreicht [33]. Sie ist unter allen Graphklassen mit polynomiell beschränkten Kantengewichten ein asymptotisches Worst-Case-Beispiel.

Ingo's Graph hat neben seiner Knotenzahl noch einen weiteren freien Parameter, die Verteilung der Knoten auf die Dreiecke und die Clique. Um mehr über Ingo's Graph zu erfahren, haben wir bei fester Knotenzahl die Verteilung der Knoten variiert und auf den entstehenden Graphen die Laufzeit des (1+1) EA gemessen.

Ingo's Graph degeneriert zu einer Kette von Dreiecken oder zu einer Clique, wenn alle Knoten vollständig auf die eine oder andere Gruppe verteilt werden. Da sich diese beiden Graphen in Zeit $o(n^4 \log(n))$ optimieren lassen, die Laufzeit von Ingo's Graph bei jedem anderen konstanten Verhältnis aber $\Theta(n^4 \log(n))$ ist, erwarteten wir für die Laufzeit in Abhängigkeit von der Verteilung einen Abfall zu beiden Enden und ein Maximum in der Mitte.

Zur Untersuchung des Einflusses der Verteilung auf die Laufzeit wurden alle möglichen Instanzen von Ingo's Graph mit 29 Knoten getestet. Es kann 0 bis 14 Dreiecke in einer Dreieckskette mit maximal 29 Knoten geben. Die übrigen Knoten bilden entsprechend die Clique, wobei ein Knoten immer Teil von beiden Strukturen ist. Die Instanzen mit 13 und 14 Dreiecken sind allerdings identisch, da eine Clique mit drei Knoten ein Dreieck ist.

Es ergeben sich 14 Graphen, die jeweils 100-mal mit dem (1+1) EA optimiert wurden (siehe Tabelle 8.1). Die 14 Graphen haben sehr unterschiedliche Kantenzahlen und entsprechen damit auch sehr unterschiedlichen Dimensionen für den (1+1) EA. Daher überrascht es nicht, dass der schwierigste Graph (zwei Dreiecke) ein Graph mit eher wenigen Dreiecken und damit vielen Kanten ist.

Dreiecke	Mittelwert	Standardfehler
0	2755,68	1,61%
1	86221,54	13,45%
2	412586,03	9,99%
3	370540,72	8,34%
4	293531,26	6,01%
5	212822,48	6,98%
6	166540,49	6,66%
7	105377,22	6,04%
8	64335,79	5,15%
9	44835,65	5,90%
10	17196,18	5,05%
11	15287,86	4,95%
12	12650,93	4,17%
13	12259,28	4,67%

Tabelle 8.1: Ingo's Graph mit 29 Knoten: Mittelwert und Standardfehler.

Um zu verstehen, welchen Einfluss die Dimension auf die Ergebnisse hat, wollten wir untersuchen, wie schwierig oder einfach der jeweilige Graph gegenüber anderen Graphen mit gleicher Dimension ist. Zu diesem Zweck haben wir zufällige Graphen untersucht.

Zu jeder der 14 bisher untersuchten Graphen wurden 100 zufällige Graphen gleichverteilt unter allen zusammenhängenden Graphen mit gleicher Knoten- und Kantenzahl gewählt. Das Kantengewicht für jede Kante wurde zufällig zwischen 1 und 1000 gewählt, wobei 1000 eine relativ willkürliche Obergrenze ist. Die Laufzeiten des (1+1) EA auf diesen Graphen geben uns eine Schätzung für den Average-Case über alle Graphen, die den ersten 14 untersuchten Graphen in ihrer Dimension entsprechen (siehe Tabelle 8.2).

Die Ergebnisse für 354 und 406 Kanten wurden nicht mehr berechnet, da die Berechnungen zu viel Zeit in Anspruch genommen hätten. Auch ohne diese ist aber an der Grafik in Abbildung 8.5 sofort ersichtlich, dass die am längsten laufende Instanz von Ingo's Graph im Vergleich mit zufälligen Graphen nicht mehr die schwierigste Instanz darstellt.

Besonders überrascht hat uns aber die Tatsache, dass in jedem einzelnen Fall zufällige Graphen schwieriger zu optimieren waren als das asymptotische Worst-Case-Beispiel. Dies wirft

Kantenzahl	$\widehat{=}$Dreiecke	Mittelwert	Standardfehler
406	0		
354	1		
306	2	22374901,32	5,11%
262	3	6625366,52	5,35%
222	4	2294090,36	4,55%
186	5	986531,96	5,03%
154	6	434350,52	4,65%
126	7	184285,64	5,05%
102	8	93647,26	4,26%
82	9	47224,92	4,82%
66	10	33744,61	4,50%
54	11	21591,49	4,71%
46	12	16327,17	4,96%
42	13	12268,23	5,37%

Tabelle 8.2: Zufällige Graphen mit 29 Knoten: Mittelwert und Standardfehler.

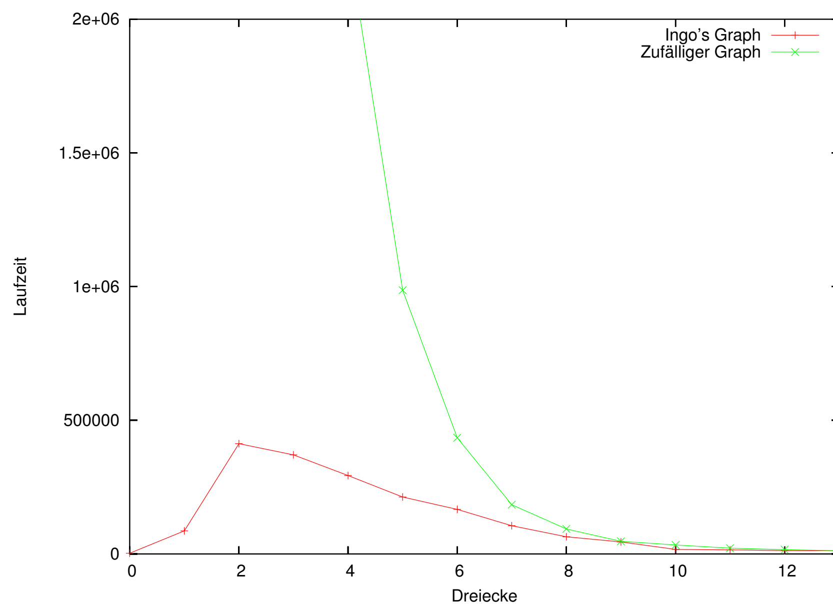


Abbildung 8.5: Durchschnittliche Laufzeiten des (1+1) EA auf Ingo's Graph mit variierender Dreieckszahl und auf zufälligen Graphen mit jeweils gleicher Dimension.

eine interessante These auf. Sollten zufällige Graphen grundsätzlich eine höhere erwartete Laufzeit haben als Ingo's Graph gleicher Dimension, wären der Worst Case und der Average Case für das MST Problem asymptotisch identisch.

Da uns diese These sehr viel interessanter als die ursprüngliche Frage erschien, haben wir in den folgenden Experimenten versucht zusätzliche Daten zu ihrer Bestätigung zu sammeln.

8.2.2 Experimente zu Kantengewichten

Was haben wir gemacht?

Es galt in diesem Experiment, den Einfluss von Graphen mit unterschiedlichen Kantengewichten auf die Laufzeit des (1+1) EAs beim Problem der minimalen Spannbäume (MST) zu untersuchen. In ersten Experimenten wurde versucht, exponentielle Kantengewichte mit polynomiellen Kantengewichten zu vergleichen. Als Ausgangspunkt hatte die gesamte Experimentreihe eine Diskussion über Worst-Case-Eingabegraphen. Nach dem ersten Herumprobieren wurden Hypothesen aufgestellt und diese anschließend statistisch ausgewertet.

Erste Experimente zur Hypothesenfindung

In [33] findet sich ein Beispiel für einen asymptotischen Worst-Case-Graphen und der zugehörige Beweis (siehe auch Abschnitt 7.1). Der dort angegebene Graph ist aber nur ein asymptotisches Worst-Case-Beispiel für Graphen mit polynomiellen Kantengewichten. Im Laufe der Diskussion über Worst-Case-Eingaben stießen wir auf die These, dass der angesprochene Worst-Case-Graph wirklich nur für polynomielle Kantengewichte ein Worst-Case-Beispiel darstellt. Im Vergleich mit exponentiellen Kantengewichten sollte er aber besser abschneiden.

Um diese Vermutung zu bestätigen, sollte ein Experiment durchgeführt werden, dass zwei Typen von Graphen miteinander vergleicht. Auf der einen Seite zufällige Graphen mit Gewichten von 1 oder 2, wobei wir zufällige Graphen dadurch erzeugen, dass zwischen allen Knotenpaaren mit einer Kantenwahrscheinlichkeit von $p = 1/2$ eine Kante eingefügt wird. Auf der anderen Seite zufällige Graphen mit Kantengewichten, die exponentielle Werte haben. Die erste Kante hat Gewicht 1, die i-te Gewicht 2^{i-1}. Diese Gewichtsverteilung nennen wir im Folgenden Binary-Value-Gewichte. Schnell stellte sich heraus, dass das gesteckte Ziel nur für kleine Graphen möglich war, ohne wesentliche Änderungen an FrEAK herbeizuführen, da die exponentiell großen Fitnesswerte nicht mehr mit den primitiven Datentypen von Java dargestellt werden konnten. Ebenfalls zeigte sich, dass unsere eigentliche Vorstellung von Binary-Value-gewichteten Graphen nicht realisierbar war. Dabei schwebte uns ein Graph vor, bei dem im Optimierungsprozess mit dem (1+1) EA in einem Schritt für eine in den Spannbaum flippende Kante beliebig viele, bereits optimale Kanten wieder heraus flippen können.

Nach einer längeren Diskussion einigte sich die Kleingruppe MST darauf, die Experimente abzuändern, um geeignete Hypothesen aufstellen zu können und statistisch zu untermauern.

8.2.3 Experimente zum Vergleich der Algorithmen (1+1) EA und RLS

Was haben wir gemacht?

Wir haben die Laufzeiten verschiedener Algorithmen wie des (1+1) EA, der RLS und von populationsbasierten Plusstrategien miteinander vergleichen.

Erste Experimente zur Hypothesenfindung

Ausgangspunkt dieses Experimentes waren theoretische Aussagen über die Laufzeit der RLS und des (1+1) EAs sowie Vermutungen über Laufzeiten des $(1+\lambda)$ EAs mit $\lambda > 1$ (siehe [33]). Mit RLS ist hier die randomisierte lokale Suche mit angepasstem Mutationsoperator aus [33] gemeint, die sowohl 1-Bit- als auch 2-Bit-Flips erlaubt. Die Erweiterung um 2-Bit-Flips ist zwingend notwendig, da die RLS sonst schnell in lokalen Optima hängen bleibt, andererseits kann auch auf 1-Bit-Flips nicht vollständig verzichtet werden.

In den ersten Experimenten wurden die vier Algorithmen (1+1) EA, (1+10) EA, (1+100) EA sowie RLS auf Graphen mit Knotenzahl 10, 15, 20, 25, 30 und 35 untersucht. Dabei wurden jeweils 10 unterschiedliche zufällige Graphen in 100 Läufen betrachtet. Für jede Kante war die Wahrscheinlichkeit, im Graphen zu existieren, gleich 1/2. Die Kantengewichte wurden aus dem Intervall $[1, \text{Knotenzahl}]$ ausgewählt.

Die ersten Ergebnisse zeigten, das die RLS als einzige Strategie unserer Auswahl mit problemangepasster Komponente den anderen Suchstrategien in Bezug auf die Fitnessauswertungen überlegen ist. Die $(1+\lambda)$ EA unterscheiden sich (in Bezug auf die Anzahl der Fitnessauswertungen) nicht großartig voneinander.

Als erste Erweiterungen der Experimente wurde vorgeschlagen, sich bei der Auswertung den Faktor zwischen den Fitnessauswertungen der RLS und den Fitnessauswertungen des (1+1) EA genauer anzuschauen. Theoretisch sollte er bei $e = 2{,}71\ldots$ liegen, da Spannbäume im Wesentlichen lokal, mit 2-Bit-Flips, optimiert werden können: Die Wahrscheinlichkeit für einen solchen 2-Bit-Flip ist beim (1+1) EA durch $1/2e$ nach unten beschränkt, bei der RLS ist er gleich 1/2. Deshalb vermuten wir, dass man gegenüber dem (1+1) EA einen Faktor von fast $2e$ erreichen kann, wenn man bei der RLS die Wahrscheinlichkeit für 2-Bit-Flips erhöht. 1-Bit-Flips müssen möglich bleiben, um die richtige Anzahl von ausgewählten Kanten bei einer „schlechten“ Initialisierung erreichen zu können.

Um diese Vermutung experimentell zu untermauern, wurden erste Läufe der RLS mit anderen 2-Bit-Flip-Wahrscheinlichkeiten gestartet. Schon 2-Bit-Flip-Wahrscheinlichkeiten von 95% bzw. 90% führten zu einem Rechenzeitgewinn von fast $2e$ gegenüber dem (1+1) EA.

Weil die Experimente abhängige Datenreihen lieferten, wurden die echten Experimente für den Hypothesentest (siehe Abschnitt 9.2.3) mit etwas anderen Voraussetzungen getätigt.

8.2.4 Experimente zur Darstellung von Spannbäumen als Prüfernummern

Was haben wir gemacht?

Die Darstellung von Spannbäumen vollständiger Graphen als Prüfernummer ist eine Bijektion und eignet sich daher gut zum Kodieren von Spannbäumen. Eine Prüfernummer ist dabei in FrEAK ein Individuum des GeneralString-Suchraums. Es gab Vermutungen, aber auch Zweifel, dass der (1+1) EA mit Hilfe der Prüfernummer-Darstellung schneller zum Optimum findet als der (1+1) EA mit Hilfe der üblichen Bitstring-Darstellung, die für jede Kante angibt, ob sie gewählt oder nicht gewählt ist. Das war Grund genug für uns, dieser Fragestellung experimentell nachzugehen. Motivation für diese Untersuchung stammt aus [36].

Erste Experimente zur Hypothesenfindung

Da wir keinerlei theoretische Aussagen oder Analysen bezüglich dieses Problems vorliegen hatten, ließen wir erst einmal einige Durchgänge mit Prüfernummer-Darstellung laufen. Schon bald beobachteten wir, dass die Laufzeiten stark variierten. Teilweise war ein Lauf nach wenigen Sekunden beim Optimum angelangt, teilweise mussten wir aber auch den Lauf nach Stunden abbrechen, da er sonst zu lange gedauert hätte. Dieses Phänomen erforderte eine Umstrukturierung des Versuchsaufbaus. Zunächst einmal beschränkten wir die Knotenzahl der Graphen auf 10 Knoten. Dies schien uns im Hinblick auf die teilweise extrem langen Laufzeiten bei Prüfernummer-Darstellung als hinreichend. Normalerweise hätten wir diese Laufzeiten der Algorithmen miteinander verglichen, allerdings mussten wir nun damit rechnen, die Läufe in Prüfernummer-Darstellung nicht in akzeptabler Zeit zu beenden.

Da wir ebenfalls nicht auf eine ausreichend große Stichprobenmenge verzichten wollten, entschlossen wir uns, die Rechenzeit bei Prüfernummern künstlich zu beschränken und anstelle der Laufzeiten Erfolgswahrscheinlichkeiten innerhalb eines festgelegten Zeitrahmens zu betrachten. Dazu bestimmten wir von einer genügend großen Anzahl von Läufen in Bitstring-Darstellung die mittlere Anzahl von Generationen bis zum Optimum und generierten daraus eine maximale Anzahl von Generationen für die Läufe auf Prüfernummern. Im Weiteren wurde gezählt, wie oft der (1+1) EA auf Prüfernummern unterhalb dieser Grenze lag und daraus die Erfolgswahrscheinlichkeit gebildet.

Einige weitere Versuche zeigten allerdings schnell, dass die Prüfernummer-Läufe lediglich eine Erfolgswahrscheinlichkeit von ca. 20% hatten, das Optimum zu erreichen, so dass eine einfache vorläufige Hypothese lautete: *Der (1+1) EA optimiert das MST-Problem auf Bäumen als Prüfernummer kodiert denkbar schlecht.* Es sei bemerkt, dass wir es der Bitstring-Darstellung schwerer als nötig gemacht haben, da wir stets bei einer rein zufälligen Kantenbelegung begonnen haben. Diese muss kein Spannbaum sein, wohingegen bei der Darstellung als Prüfernummer stets bei einem solchen begonnen wird. Man hätte gleiche Start-Bedingungen schaffen können, indem man bei Bitstring-Darstellung ebenfalls bei einem Spannbaum begonnen hätte, jedoch wären damit die Bitstring-Läufe beschleunigt worden und hätten damit den Prüfernummer-Läufen noch weniger Zeit/Generationen pro Lauf gegeben. Dies hätte die Ergebnisse nur bestätigt.

Aufgrund der langen Laufzeit bei Prüfernummer-Darstellung testeten wir lediglich vollständige Graphen mit 10 Knoten. Einige Versuche zeigten uns, dass schon mit wenig mehr Knoten

die Laufzeit unter Prüfernummer-Darstellung rapide ansteigt und somit die Zeit zur eigentlichen Experimentdurchführung knapp werden könnte. Dazu wurden jedoch im Folgenden keine weiteren Experimente durchgeführt.

In einem vollständigen Graphen, in dem jede Kante Gewicht 1 hat, ist jeder Spannbaum ein optimaler Spannbaum. Daher interessierte uns die Frage, was passieren würde, wenn wir die Anzahl möglicher unterschiedlicher Kantengewichte erhöhen würden. Dies führte zu folgender Hypothese.

Hypothese 20. *Sei* $w \in \{2, 4, 6, \ldots, 50\}$ *die maximale Anzahl unterschiedlicher Kantengewichte in einem vollständigen Graphen. Unter Verwendung von Prüfernummern als Darstellung von Spannbäumen eines vollständigen Graphen benötigt der (1+1) EA für alle* w *deutlich länger zur Optimierung des MST-Problems als unter der Verwendung von Bitstrings zur Darstellung beliebiger Kantenauswahlen.*

Nun kam die Frage auf, auf welchen Graphen Prüfernummern sinnvoll sein können. Wir experimentierten mit vollständigen Graphen, deren einziger minimaler Spannbaum aus einem Stern aus Kanten mit Gewicht 1 um einen einzigen Knoten herum besteht, und entschieden uns, diese Graphen näher zu untersuchen.

Hypothese 21. *Unter Verwendung von Prüfernummern als Darstellung von Spannbäumen eines oben beschriebenen sternförmigen Graphen benötigt der (1+1) EA deutlich weniger Generationen zur Optimierung des MST-Problems als auf vollständigen Graphen mit zufälligen Kantengewichten.*

Es wurde vermutet, dass die Prüfernummer-Darstellung in diesem Fall sogar bessere Laufzeiten erzielt als die Bitstring-Darstellung.

8.3 Maximale Matchings

Die Experimente zum Problem „Maximale Matchings“ wurden durch zwei Vorträge motiviert, die in den Abschnitten 3.10 und 7.8 beschrieben sind. Der erste Vortrag aus der ersten Seminarphase diente dabei als Einleitung in das Thema und zeigte uns insbesondere eine Analyse des (1+1) EA auf Pfaden und eine Klasse von Graphen, die nur in exponentieller Zeit optimiert werden können.

Der zweite Vortrag fand in der zweiten Seminarphase statt und schloss an den ersten Vortrag an. Darin wurde versucht, Graphklassen nach ihrer Schwierigkeit für den (1+1) EA zu kategorisieren; insbesondere drehte sich der Vortrag um die Betrachtung und die Analyse von Bäumen.

Aus den Vorträgen und den zugehörigen Diskussionen entstanden folgende Ideen für Experimente:

- Wie verhält sich der (1+1) EA auf Pfaden und zufälligen Bäumen?
- Könnte eine RLS, die sich nur auf 1- und 2-Bit-Flips beschränkt, bessere Laufzeiten auf Pfaden erreichen?
- Gibt es andere Graphklassen, die, wie der Pfad, ebenfalls schwer sein könnten?
- Bringt uniformes Crossover Vorteile, insbesondere auf Pfaden?
- Wie verhält sich der (1+1) EA auf sogenannten „Semi-Random-Graphen“, d. h. auf zufälligen Graphen mit einem künstlich eingepflanzten perfekten Matching?

Im Folgenden werden die ersten Erkundungsexperimente zu den genannten Fragestellungen beschrieben.

8.3.1 Matchings auf Bäumen und Pfaden

Der Vortrag über maximale Matchings (Abschnitt 7.8) lieferte natürlich gleich Stoff für Experimente. Es wurde im Besonderen über Bäume und Linien bzw. Pfade referiert und in diesem Zusammenhang die Frage aufgestellt, ob Pfade die schwersten Bäume sind. Dazu gab es in kleinen Untergruppen die Aufgabe, sich Graphen zu überlegen, die möglicherweise schwer zu optimieren sind. Leider gelang es keiner der Gruppen, sich einen schwereren Baum als Pfade auszudenken. Als erstes wollten wir nun den (1+1) EA gegen die randomisierte lokale Suche (RLS) auf den vermutlich schwersten Bäumen für maximale Matchings, den Pfaden vergleichen.

Die Struktur des Matching-Problems legt nahe, dass 1- und 2-Bit-Flips die am häufigsten benötigten Flips sind, um gegen Ende des Laufs das entstandene Matching zu einem optimalen Matching werden zu lassen. Daher wählten wir eine RLS, die lediglich diese beiden Mutationstypen erlaubt und jeweils mit Wahrscheinlichkeit 0,5 durchführt. Daraus resultierte unsere erste Hypothese.

Hypothese 22. *Die auf 1- und 2-Bit-Flips beschränkte RLS optimiert das Problem der maximalen Matchings auf Pfaden schneller als der (1+1) EA.*

Wir nahmen uns vor, bei der Untersuchung die Anzahl der Knoten des Pfades zu variieren und die sich so entwickelnden Laufzeiten mit unserer Theorie in Verbindung zu bringen.

Des Weiteren wurde im Vortrag vermutet, dass der Pfad unter allen Bäumen gleicher Größe die längste erwartete Laufzeit hat und dass diese bei $\Theta(n^4)$ liegt. Dies führt unmittelbar zu folgenden beiden Hypothesen.

Hypothese 23. *Der (1+1) EA optimiert das Problem der maximalen Matchings auf Pfaden in erwarteter Laufzeit von* $\Theta(n^4)$.

Natürlich wollten wir im Experiment nicht an allen möglichen Bäumen zeigen, dass der (1+1) EA auf ihnen schneller arbeitet als auf Pfaden, daher beschränkten wir uns auf zufällig gewählte Bäume. Ein zufällig gewählter Baum wird dabei aus einer uniform zufällig ausgewürfelten Prüfer-Nummer generiert. So wird gewährleistet, dass wirklich jeder mögliche Baum mit gleicher Wahrscheinlichkeit gewählt wird.

Hypothese 24. *Der (1+1) EA optimiert das Problem der maximalen Matchings auf Pfaden langsamer als auf zufällig gewählten Bäumen gleicher Größe.*

Auch hier wurde natürlich die Anzahl der Knoten als variabler Parameter definiert.

Erste Versuche bestätigten, dass der (1+1) EA auf Pfaden in der Tat länger lief als beispielsweise auf zufälligen Bäumen. Wir erwarteten keine großen Überraschungen diesbezüglich.

Da im Vortrag ebenfalls vollständige k-äre Bäume angesprochen wurden, wollten wir diese natürlich auch untersuchen. Da wir die Ergebnisse allerdings direkt mit den vorherigen Ergebnissen vergleichen wollten, waren wir gezwungen, auch nicht vollständige k-äre Bäume zuzulassen. Unsere erste Implementierung folgte der Idee, die unterste Ebene des k-ären Baumes so weit wie möglich von links nach rechts mit Knoten bzw. Blättern zu füllen, bis die passende Knotenzahl erreicht wurde.

Interessanterweise ahnte zu diesem Zeitpunkt noch niemand, welch verheerende Auswirkung diese Entscheidung auf die Laufzeiten hatte. Es wurden daraufhin zusätzliche, das Ergebnis verfeinernde Experimente durchgeführt, die die festgestellte Anomalie jedoch nur noch bestätigten. Daraufhin wurde ein zweiter Implementierungsansatz gewählt, der zwar ebenfalls die unterste Ebene des Baumes mit den übrigen Knoten bzw. Blättern füllt, allerdings an zufälligen Stellen und nicht linear von einer Seite.

Da wir aus diesen Ergebnissen leider nicht wirklich schlau wurden, entschlossen wir uns, diesbezüglich keine Hypothese aufzustellen, sondern nur die Beobachtungen zu formulieren.

Die Experimentdurchführung ist in Abschnitt 9.3.1 nachzulesen.

8.3.2 Nutzen von Uniform Crossover bei Matchings auf Pfaden

Während unserer Überlegung zu maximalen Matchings kam die Vermutung auf, dass uniformes Crossover auf kurzen Pfaden sinnvoll sein könnte. Da kurze Pfade nicht wirklich schwer

zu optimieren sind, wollten wir nachprüfen, ob uniformes Crossover auf schwierigen Instanzen überhaupt einen Nutzen bringt.

Um diese Frage zu beantworten, musste ein geeigneter Versuchsaufbau gefunden werden. Als Populationsgröße wählten wir nur zwei Individuen. Größere Populationen verursachen mehr Fitnessauswertungen, bei nur einem Individuum ist uniformes Crossover natürlich nicht möglich. Wenn uniformes Crossover auf maximalen Matchings einen Nutzen hat, sollte es auch bei nur zwei Individuen sichtbar sein.

Uniformes Crossover hat überhaupt nur Sinn, wenn sich die beiden Eltern nur leicht voneinander unterschieden. Die Anzahl möglicher verschiedener Nachkommen steigt exponentiell mit dem Hammingabstand der beiden Eltern, wobei die Anzahl gültiger Matchings, die mindestens so gut sind wie einer der beiden Eltern, nur sehr langsam ansteigt. Somit ist es wünschenswert, dass die beiden Individuen nur einen kleinen Hammingabstand voneinander haben.

Um dies zu erreichen, haben wir mit *Fitness Sharing* experimentiert. Es zeigte sich leider, dass unser Ansatz von Fitness Sharing komplett ungeeignet war, eine leichte Diversität in der Population zu erhalten und gleichzeitig in akzeptabler Zeit das Optimum zu finden. Bei Einsatz von Fitness Sharing versammelten sich die Individuen um das Optimum, wirklich schnell erreichte es aber kein Individuum, da sie sich gegenseitig vom Optimum abhielten.

Ein weiteres Problem bestand im Finden eines geeigneten Operatorgraphen. Ein (2+2) EA, um uniformes Crossover erweitert, liefert eine Population mit sehr kleiner Diversität, die fast nur aus Replikanten besteht. Um diese Replikanten zu verhindern, erweiterten wir eine (2+2) RLS um uniformes Crossover. Aus jedem der beiden Individuen wird mit gleicher Wahrscheinlichkeit ein Nachkomme durch eine 1-Bit-Mutation oder eine 2-Bit-Mutation erzeugt. Mit beiden Eltern wird zusätzlich durch uniformes Crossover ein dritter Nachkomme generiert. Eine Schnittselektion liefert aus den drei Kindern und ihren beiden Eltern die besten beiden Individuen für die nächste Population, wobei bei gleicher Fitness die jüngeren Individuen bevorzugt werden.

Nun könnte man diesen Operatorgraphen mit einer einfachen (1+1) RLS vergleichen. Dies ist aber eigentlich kein fairer Vergleich, da der Operatorgraph drei Individuen pro Generation auswertet, die (1+1) RLS aber nur eins. Man könnte dann einfach eine (2+3) RLS nehmen, die aber keine gute Vergleichsmöglichkeit bietet. Aus diesem Grund haben wir uns für den Vergleich der (2+2) RLS und der (2+2) RLS mit zusätzlichem uniformen Crossover, im Folgenden als (2+3) RLS+Uniform bezeichnet, entschieden.

Hypothese 25. *Der Algorithmus (2+3) RLS+Uniform benötigt für maximale Matchings auf Pfaden im Durchschnitt weniger Generationen als die (2+2) RLS.*

8.3.3 Matchings auf Semi-Random-Graphen

Nachdem wir während der beiden Seminarphasen bereits einige Ergebnisse über allgemeine Graphen und Pfade (als strukturell „einfachste“ Bäume) gesehen haben, wollten wir mit diesen Experimenten erste Ergebnisse über Semi-Random-Graphs sammeln.

Dabei handelt es sich um ein allgemeines Modell, nach dem „zufällige“ Instanzen erzeugt werden. Zunächst wird deterministisch eine Instanz kreiert und anschließend werden nach einer bekannten Verteilung Komponenten randomisiert hinzugefügt.

Auf Graphen angewendet bedeutet das in unserem Fall, dass wir einen Graphen mit m Kanten erzeugen, in dem ein perfektes Matching mit $n/2$ Kanten vorgegeben ist. Die untersuchten Instanzen entstanden also nach folgendem Prinzip.

- Erzeuge ein perfektes Matching auf n Knoten, indem für jedes $i \in \left\{1, \ldots, \frac{n}{2}\right\}$ der Knoten $2i$ mit dem Knoten $2i+1$ verbunden wird.
- Füge uniform zufällig $m - \frac{n}{2}$ noch nicht bestehende Kanten hinzu.

Unser Suchraum besteht demnach aus einem Bitstring der Länge m, wobei jede Auswahl einer Kante durch ein Bit dargestellt ist. Wir können also in unserem Semi-Random-Graphen die Suchraumdimension konstant halten, wobei wir die Größe des vorgegebenen Matchings variieren.

In jeder gegebenen Instanz existiert mindestens ein perfektes Matching, das alle n Knoten abdeckt. Dies wird jedoch durch die zusätzlich eingefügten Kanten „verschleiert“. Jedoch könnte es durch die zugefügten Kanten auch passieren, dass es mehrere perfekte Matchings gibt, die dann vielleicht einfacher zu finden sind. Dies passiert vor allem im Extremfall, wenn wir den vollständigen Graphen vorliegen haben.

Unser Ziel ist es also herauszufinden, wie sich diese beiden Effekte verhalten, bis zu welcher Größe n eines perfekten Matchings (Anzahl der Knoten) bei vorgegebener Kantenzahl m im Graphen ein evolutionärer Algorithmus „schnell“ das Optimum findet. Andererseits wollen wir auch untersuchen, ab welcher Größe es dann wieder „schnell“ geht.

Dies impliziert eine gewisse Vorstellung der erwarteten Kurve, trägt man die Größe des vorgegebenen Matchings gegenüber der mittleren Generationenzahl bis zum Optimum auf. Folgende Hypothese formalisiert unsere Erwartung.

Hypothese 26. *Sei $G = (V, E)$ ein nach dem obigen Algorithmus gewählter Graph. Wir halten im Folgenden $|E| = m$ fest. Sei A ein (1+1) EA, der auf dem Bitstring der Kantenauswahl operiert, $n_0 = \min\left\{n \mid \binom{n}{2} \geq m\right\}$ und $n_1 = 2m$. Trägt man die erwartete Generationenzahl von A bis zum Erreichen des Optimums gegenüber der Größe des gegebenen Matchings im Bereich von n_0 bis n_1 auf, so gibt es eine Zahl k für die gilt:*

- *Im Bereich n_0 bis k steigt die Kurve.*
- *Im Bereich k bis n_1 fällt die Kurve.*

Offensichtlich beschreibt die Hypothese die oben genannte Erwartung. Der Parameterbereich n_0 und n_1 spiegelt die natürlich gegebenen Grenzen wider. Wir benötigen mindestens n_0 Knoten, um m Kanten in dem Graphen zu haben und sobald die Grenze von n_1 Knoten erreicht ist, ist jede vorhandene Kante eine Matching-Kante. Dann ist lediglich noch OneMax zu optimieren und bei größeren Knotenzahlen ändert sich nichts Wesentliches mehr.

Kapitel 9

Hypothesentests

Inhalt

9.1 Das Ising-Modell 148

9.1.1 Das Ising-Modell auf Cliquen 148

9.1.2 Das Ising-Modell auf Tori 160

9.1.3 Das Ising-Modell auf booleschen Hypercubes 165

9.2 Minimale Spannbäume 174

9.2.1 Experimente auf einem asymptotischen Worst-Case-Beispiel 174

9.2.2 Experimente zu Kantengewichten 176

9.2.3 Experimente zum Vergleich der Algorithmen (1+1) EA und RLS . . 181

9.2.4 Experimente zur Darstellung von Spannbäumen als Prüfernummern 188

9.2.5 Vergleich zweier Fitnessfunktionen 193

9.3 Maximale Matchings 199

9.3.1 Matchings auf Bäumen und Pfaden 199

9.3.2 Nutzen von Uniformem Crossover bei Matchings auf Pfaden 207

9.3.3 Matchings auf Semi-Random-Graphen 210

9.4 Kürzeste Wege in Graphen 213

9.4.1 Suchraum, Fitnessfunktionen und Algorithmus 213

9.4.2 Experimente 214

9.4.3 Ergebnisse 214

In diesem Kapitel werden nun die Tests der in Kapitel 8 vorgestellten Hypothesen beschrieben.

Viele dieser Hypothesen wurden mit Methoden aus der Statistik getestet. Wie schon in Kapitel 2 erwähnt, wurden dazu die Programme SPSS, OpenOffice und Gnuplot verwendet. In einigen Hypothesen wurden Wahrscheinlichkeiten abgeschätzt, mit denen bestimmte Ereignisse in einem Lauf des (1+1) EA oder der randomisierten lokalen Suche (RLS) eintreten. Dazu wurde die Rate simulierter Läufe gemessen, in denen dieses Ereignis beobachtet wurde. Dies entspricht einem Bernoulli-Versuch mit fester Erfolgswahrscheinlichkeit, der binomialverteilte Werte liefert. Um diese Werte zu analysieren, wurden separate Java-Programme für Binomialtests geschrieben und verwendet.

Unter dem Begriff *„Signifikanz“* verstehen wir im Folgenden die Wahrscheinlichkeit, dass die gemessenen Werte erhalten wurden, unter der Annahme, dass die Nullhypothese gilt. Eine Signifikanz von $p \leq 0{,}05$ bezeichnen wir dabei, wie allgemein üblich, als „statistisch signifikant“ (*). Als „sehr signifikant“ (**) gelten Tests mit $p \leq 0{,}01$ und als „hoch signifikant“ (***) gelten Tests mit $p \leq 0{,}001$.

9.1 Das Ising-Modell

9.1.1 Das Ising-Modell auf Cliquen

Test von Hypothese 1: Mit konstanter Wahrscheinlichkeit erreicht der (1+1) EA auf dem Treppengraphen die erste Hälfte eines Pfades.

Um Hypothese 1 für einige Knotenzahlen durch Experimente zu untermauern, wurden für Treppengraphen mit verschiedenen Knotenzahlen $n = 128, \ldots, n = 2048$ jeweils 10.000 Läufe des (1+1) EA simuliert und gemessen, in wie vielen Läufen der (1+1) EA einen der Punkte $0^i 1^{n/2-i} 0^{n/2}$ oder $1^i 0^{n/2-i} 1^{n/2}$ mit $0 \leq i < n/8$ erreicht. Die gemessenen Daten sind in Tabelle 9.1 aufgeführt.

Knoten	Rate
128	0,1115
256	0,1220
384	0,1229
512	0,1237
640	0,1243
768	0,1203
896	0,1228
1024	0,1199
1152	0,1208
1280	0,1233
1408	0,1176
1536	0,1223
1664	0,1278
1792	0,1206
1920	0,1257
2048	0,1185

Tabelle 9.1: Die gemessene Rate von Läufen, in denen der (1+1) EA auf dem Treppengraphen die erste Hälfte eines Pfades erreicht hat.

Wie man sieht, liegen die Werte recht nahe beieinander. Daher vermuten wir, dass mit statistischer Signifikanz auch die den gemessenen Raten zugrunde liegenden Wahrscheinlichkeiten in einem kleinen Intervall (l, r) liegen.

Zur Auswertung wurde ein Binomialtest verwendet, der die Signifikanz bestimmt, mit der die den gemessenen Raten r_i zugrunde liegenden Wahrscheinlichkeiten p_i von konstanten

Schranken nach oben bzw. nach unten abweichen. Für ein p_i wurde die Wahrscheinlichkeit, dass $l < p_i < r$ nicht gilt, berechnet durch die Summe der Wahrscheinlichkeiten, dass beide Einzelereignisse nicht gelten:

$$\text{Prob}(l \geq p_i \vee p_i \geq r) = \text{Prob}(l \geq p_i) + \text{Prob}(p_i \geq r)$$

Da die Ereignisse $l < p_i < r$ für alle i paarweise unabhängig sind, ergibt sich die Wahrscheinlichkeit, dass alle p_i im Intervall (l, r) liegen durch

$$\text{Prob}(\forall i \; l < p_i < r) = \prod_i \text{Prob}(l < p_i < r) = \prod_i (1 - (\text{Prob}(l \geq p_i) + \text{Prob}(p_i \geq r)))$$

Für gegebene Schranken l, r wurden dann für alle Raten mit Binomialtests die Signifikanzen für die Hypothesen $l < p_i$ und $p_i < r$ ermittelt und nach den obigen Formeln miteinander verrechnet um die Signifikanz der Gesamthypothese zu erhalten, dass alle Wahrscheinlichkeiten p_i im Intervall (l, r) liegen.

Mit dieser Methode wurden daraufhin passende Werte für l, r ermittelt, bei denen die Gesamthypothese noch mit statistischer Signifikanz bestätigt werden kann. Es zeigte sich, dass mit einer Signifikanz von (gerundet) $p = 0{,}000$ bestätigt werden konnte, dass die zu den gemessenen Raten gehörenden Wahrscheinlichkeiten im Intervall $(0{,}10, 0{,}14)$ liegen.

Test der Hypothesen 2 bis 6

Da die Hypothesen 2 bis 6 alle strukturell ähnliche Aussagen über die Entwicklung der Erfolgswahrscheinlichkeiten in verschiedenen Szenarien treffen, haben wir sie durch Experimente mit sehr ähnlichen Versuchsanordnungen getestet. Wir haben Versuchsreihen mit Cliquen verschiedener Größe durchgeführt. Dabei begannen wir mit zwei Cliquen mit je 30 Knoten und erhöhten die Knotenzahl in jeder Clique jeweils um 10, solange bis jede Clique aus 100 Knoten bestand. Weiterhin untersuchten wir Cliquen mit 150, 200 und 300 Knoten. In jeder dieser Versuchsreihen sollte der Einfluss der drei Strategien zur Anordnung von Brückenkanten und der Einfluss der Anzahl der Brückenkanten auf die Erfolgswahrscheinlichkeiten des (1+1) EA und der (1+1) RLS untersucht werden. Deswegen führten wir für jede Kombination aus Algorithmus und Strategie zur Kantenanordnung eine Messreihe durch, in der wir die Kantenzahl von 0 bis zur maximal möglichen Anzahl $m_{\max}$ in 501 Schritten variierten. Im i-ten Schritt $(0 \leq i \leq 500)$ betrug die Kantenzahl $\text{round}(i \cdot m_{\max}/500)$ und es wurden jeweils 2000 Läufe durchgeführt. Dabei ist zu beachten, dass bei der zufälligen Strategie zur Kantenwahl für jeden Lauf eine neue Instanz ausgewürfelt wurde.

Ein festgelegtes Szenario bestehend aus Algorithmus, Cliquengröße, Brückenkantenzahl und Strategie zur Wahl der Brückenkanten entspricht einem Bernoulli-Versuch mit einer bestimmten Erfolgswahrscheinlichkeit. Jeden dieser Bernoulli-Versuche haben wir also unabhängig 2000-mal wiederholt und erhielten somit Schätzwerte für die Erfolgswahrscheinlichkeiten.

Test von Hypothese 2: Die Erfolgswahrscheinlichkeit ist bei konstantem $q(m, n)$ unabhängig von der Cliquengröße.

Diese Hypothese erwies sich bei genauerer Betrachtung der Daten als nicht haltbar. Die gemessenen Erfolgswahrscheinlichkeiten schienen sogar systematisch für größer werdende Cli-

quen bei konstantem Koeffizienten $q(m, n)$ zu fallen. Um die Erfolgswahrscheinlichkeiten für verschiedene Cliquengrößen für zwei Messreihen zu vergleichen, haben wir den Wilcoxon-Rangsummentest auf die 501 verschiedenen Paare von Schätzwerten, die sich aus den beiden Messreihen ergeben, angewandt. Wir haben dabei zunächst alle Messreihen mit der Messreihe für zwei Cliquen mit je 30 Knoten verglichen und anschließend haben wir alle Paare von aufeinander folgenden Messreihen verglichen. Die Ergebnisse dieser Tests sind in Tabelle 9.2 dargestellt.

	Richtung der Abweichung und Signifikanz											
	(1+1) RLS						(1+1) EA					
Cliquengrößen	zufällig		gleichm.		konz.		zufällig		gleichm.		konz.	
40 – 30	–	0,000	–	0,000	–	0,000	–	0,000	–	0,003	–	0,057
50 – 30	–	0,000	–	0,000	–	0,000	–	0,000	–	0,000	–	0,000
60 – 30	–	0,000	–	0,000	–	0,000	–	0,000	–	0,000	–	0,005
70 – 30	–	0,000	–	0,000	–	0,000	–	0,000	–	0,000	–	0,001
80 – 30	–	0,000	–	0,000	–	0,000	–	0,000	–	0,000	–	0,015
90 – 30	–	0,000	–	0,000	–	0,000	–	0,000	–	0,000	–	0,014
100 – 30	–	0,000	–	0,000	–	0,000	–	0,000	–	0,000	–	0,004
150 – 30	–	0,000	–	0,000	–	0,000	–	0,000	–	0,000	–	0,049
200 – 30	–	0,000	–	0,000	–	0,000	–	0,000	–	0,000	–	0,511
300 – 30	–	0,000	–	0,000	–	0,000	–	0,000	–	0,855	+	0,764
40 – 30	–	0,000	–	0,000	–	0,000	–	0,000	–	0,003	–	0,057
50 – 40	–	0,075	–	0,003	–	0,000	–	0,000	–	0,000	–	0,094
60 – 50	–	0,000	–	0,740	–	0,000	–	0,000	–	0,120	+	0,198
70 – 60	–	0,840	–	0,015	–	0,000	–	0,015	–	0,025	–	0,386
80 – 70	–	0,002	–	0,006	–	0,000	–	0,305	–	0,073	+	0,274
90 – 80	–	0,505	–	0,036	–	0,000	–	0,070	–	0,884	+	0,958
100 – 90	–	0,011	–	0,259	–	0,007	–	0,001	–	0,062	–	0,504
150 – 100	–	0,000	–	0,001	–	0,000	–	0,000	–	0,008	–	0,973
200 – 150	–	0,150	–	0,022	–	0,000	–	0,000	–	0,005	+	0,008
300 – 200	–	0,001	–	0,010	–	0,000	–	0,000	+	0,000	+	0,980

Tabelle 9.2: Die Ergebnisse der Wilcoxon-Rangsummentests. In der Zeile mit den Cliquengrößen „$x - y$“ bedeutet ein $-$, dass die Messreihe mit zwei Cliquen mit je x Knoten zu durchschnittlich kleineren Schätzwerten geführt hat als die Messreihe mit y Knoten pro Clique. Ein + bedeutet das Gegenteil. Die Signifikanzen sind auf drei Stellen gerundet.

Überraschenderweise weichen die Ergebnisse in den verschiedenen Szenarien deutlich voneinander ab. Betrachtet man zunächst die Ergebnisse für die (1+1) RLS, so stellt man fest, dass die Messreihen mit Cliquen mit mehr als 30 Knoten alle zu signifikant kleineren Schätzwerten geführt haben als die Messreihe mit zwei Cliquen der Größe 30. Ferner sieht man, dass auch die Tests von benachbarten Messreihen stets das Ergebnis geliefert haben, dass mit steigender Cliquengröße die Erfolgswahrscheinlichkeit bei konstantem $q(m, n)$ sinkt. Für konzentriert gewählte Brückenkanten ist letzteres in jedem Falle äußerst signifikant, für zufällig und gleichmäßig gewählte Brückenkanten gibt es einige Paare von Messreihen, bei denen das

Ergebnis weniger bis gar nicht signifikant ist.

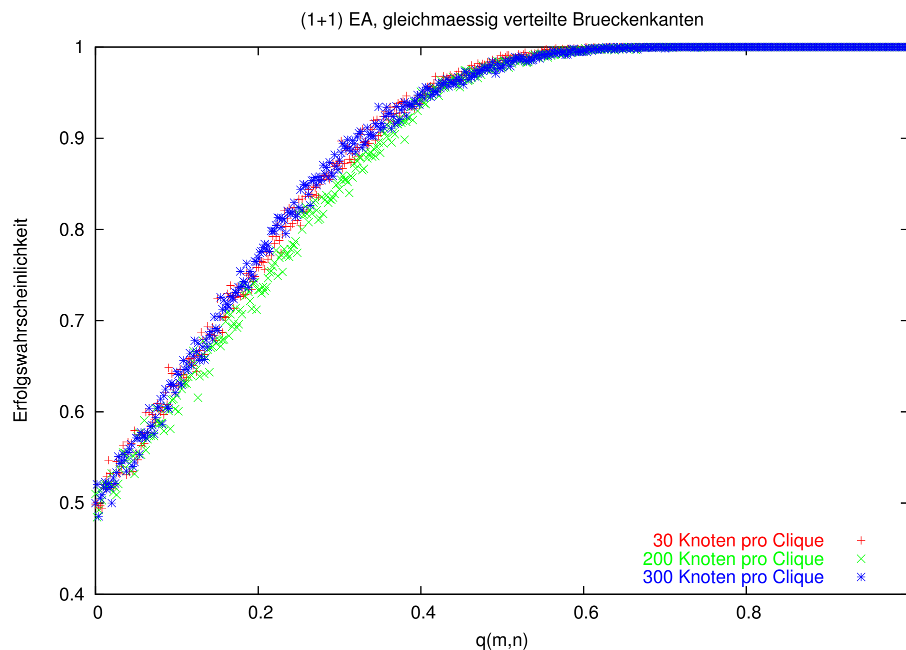


Abbildung 9.1: Entwicklung der Erfolgswahrscheinlichkeit für steigende Cliquengrößen.

Beim (1+1) EA bietet sich bei zufällig gewählten Brückenkanten ein sehr ähnliches Bild zu den beschriebenen Ergebnissen der (1+1) RLS, denn auch hier konnte bis auf zwei Messreihen signifikant nachgewiesen werden, dass die durchschnittliche Erfolgswahrscheinlichkeit mit steigender Cliquengröße in dem von uns untersuchten Bereich fällt.

Bei gleichmäßig verteilten Brückenkanten gilt die gleiche Aussage, solange die Cliquengröße kleiner oder gleich 200 bleibt. Das Überraschende ist jedoch, dass beim Vergleich der Messreihen für je 200 und je 300 Knoten pro Clique gezeigt wurde, dass die Messreihe mit Cliquen der Größe 300 äußerst signifikant eine größere durchschnittlich Erfolgswahrscheinlichkeit zeigt als die Messreihe mit Cliquen der Größe 200. Der Anstieg der Erfolgswahrscheinlichkeit in diesem Bereich ist sogar so groß, dass kein signifikanter Unterschied mehr zwischen der Messreihe mit Cliquen der Größe 30 und der Messreihe mit Cliquen der Größe 300 gezeigt werden konnte (siehe Abbildung 9.1).

Bei konzentriert gewählten Brückenkanten zeigt sich zwar, dass die Messreihen mit höchstens 150 Knoten pro Clique zu signifikant kleineren Schätzwerten als die Messreihe mit 30 Knoten geführt haben, jedoch werden die Abstände von der Messreihe mit 30 Knoten pro Clique zu den beiden letzten Messreihen mit 200 bzw. 300 Knoten pro Clique auch hier wieder so klein, dass mit dem Wilcoxon-Rangsummentest keine signifikante Aussage getroffen werden kann. Außerdem scheinen die Unterschiede zwischen benachbarten Messreihen bis auf eine Ausnahme so klein zu sein, dass keine signifikanten Aussagen über ihr Verhalten gemacht werden können.

Test von Hypothese 3: Es macht keinen Unterschied, ob die Brückenkanten zufällig oder gleichmäßig verteilt werden.

Um diese Hypothese zu bestätigen, haben wir die Messreihen mit zufällig gewählten Brückenkanten mit denen mit gleichmäßig verteilten Brückenkanten paarweise verglichen. Als erstes haben wir auch hier Wilcoxon-Rangsummentests durchgeführt und stellten zu unserer Überraschung fest, dass sie für viele Messreihen einen signifikanten Unterschied zwischen den beiden Strategien zur Brückenkantenwahl zeigen. Die Ergebnisse finden sich in Tabelle 9.3.

Die (1+1) RLS hat also für alle von uns getesteten Cliquengrößen größer als 30 eine signifikant höhere Erfolgswahrscheinlichkeit, wenn die Brückenkanten gleichmäßig verteilt sind. Beim (1+1) EA ist für kleine Cliquen ebenfalls die Erfolgswahrscheinlichkeit bei gleichmäßigen Brückenkanten signifikant größer, bei großen Cliquen ist es jedoch umgekehrt.

	Richtung der Abweichung und Signifikanz			
Cliquengröße	(1+1) RLS		(1+1) EA	
30	−	0,568	−	0,000
40	−	0,000	−	0,016
50	−	0,001	−	0,006
60	−	0,000	−	0,781
70	−	0,000	+	0,562
80	−	0,000	−	0,800
90	−	0,000	+	0,274
100	−	0,000	+	0,296
150	−	0,000	+	0,011
200	−	0,000	+	0,000
300	−	0,000	+	0,000

Tabelle 9.3: Der Wilcoxon-Rangsummentest wurde angewendet, um zwei Messreihen mit der gleichen Cliquengröße, der gleichen Brückenkantenzahl und dem gleichen Algorithmus, aber unterschiedlichen Strategien zur Anordnung der Brückenkanten zu vergleichen. Ein + bedeutet, dass die Messwerte bei zufällig gewählten Brückenkanten größer sind als die bei gleichmäßig verteilten. Ein − bedeutet das Gegenteil. Die Signifikanzen sind auf drei Stellen gerundet.

Da dieses Ergebnis dem optischen Eindruck (Abbildung 9.2), den man durch Betrachtung der Kurven erhält, zum Teil widerspricht, haben wir zusätzlich den Abstand der Kurven für gleichmäßige und zufällige Brückenkanten näher untersucht. Dazu haben wir ausgenutzt, dass wir bei einem komplett festgelegten Szenario einen Bernoulli-Versuch mit einer festen Erfolgswahrscheinlichkeit vorliegen haben. Vergleicht man für eine Cliquengröße die beiden Kurven, die sich für die beiden Strategien zur Kantenwahl ergeben, so hat man 501 Paare von Schätzwerten vorliegen. Diese haben wir jedes für sich, unabhängig von den anderen, miteinander verglichen. Wir haben eine Signifikanz von 0,05 vorgegeben und das kleinste $\varepsilon > 0$ gesucht, so dass ein Binomialtest ergibt, dass jedes der 501 Paare von Erfolgswahrscheinlichkeiten mit einer Signifikanz von 0,05 um höchstens ε voneinander abweicht. Der Nachteil dieses Verfahrens ist jedoch, dass bereits ein Ausreißer dafür sorgt, dass ε größer wird als für die Mehrheit der Paare von Schätzwerten gebraucht. Deswegen haben wir mit $\varepsilon = 0{,}01$ begonnen, es in

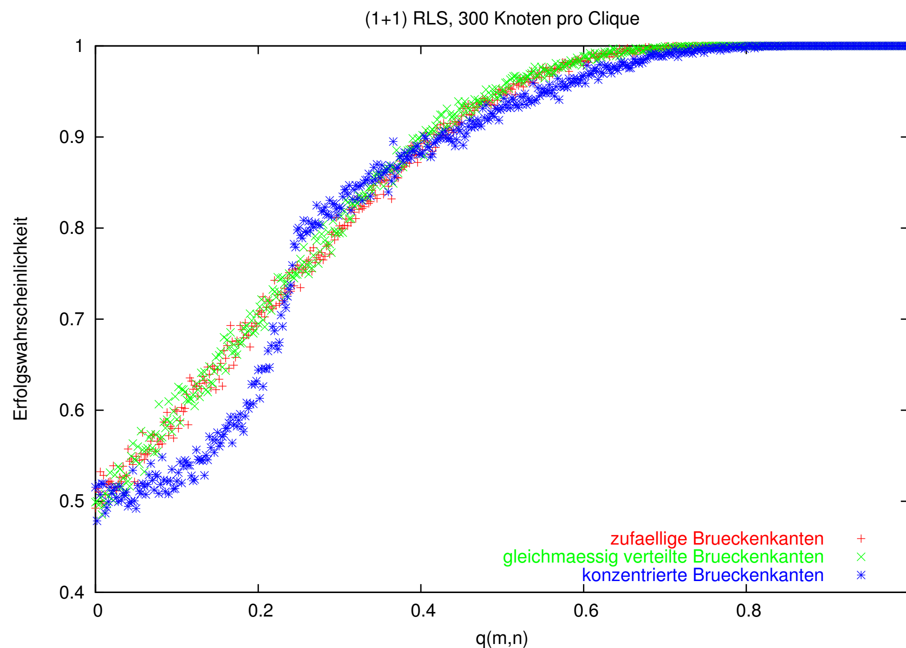


Abbildung 9.2: Vergleich der Erfolgswahrscheinlichkeiten für die drei verschiedenen Strategien zur Verteilung der Brückenkanten.

jedem Schritt um 0,01 erhöht und jeweils getestet, wie viele Paare bereits einen signifikant kleineren Abstand haben. Die Ergebnisse sind in Tabelle 9.4 dargestellt.

Wir können also festhalten, dass in jeder Messreihe mit einer Ausnahme für mindestens 95,8% der 501 Paare von Erfolgswahrscheinlichkeiten mit einer Signifikanz von 0,05 gezeigt werden konnte, dass sie um höchstens $\varepsilon = 0{,}05$ voneinander abweichen. Die Ausnahme ist der (1+1) EA auf Cliquen mit jeweils 300 Knoten. Dort ist die Abweichung größer, was sich auch in einer größeren durchschnittlichen Abweichung der Schätzwerte widerspiegelt.

Test von Hypothese 4: Konzentrierte Brückenkanten führen beim (1+1) EA zu einer größeren Erfolgswahrscheinlichkeit als gleichmäßig oder zufällig verteilte.

Auch hier haben wir mit dem Wilcoxon-Rangsummentest die Kurven paarweise verglichen. Die Ergebnisse dabei waren sehr überzeugend, denn für jede Cliquengröße ergab sich mit einer auf drei Stellen gerundeten Signifikanz von 0,000, dass die Kurve der Erfolgswahrscheinlichkeiten des (1+1) EA bei konzentrierten Brückenkanten im Durchschnitt oberhalb der Kurven für zufällig und gleichmäßig verteilte Brückenkanten verläuft.

Algorithmus		Erlaubte Abweichung ε										$\emptyset$
Cliquengröße		0,01	0,02	0,03	0,04	0,05	0,06	0,07	0,08	0,09	0,1	Abw.
(1+1) RLS	30	240	296	381	449	486	497	500	501			0,006
	40	221	286	363	446	483	494	497	501			0,006
	50	228	288	384	458	487	498	500	501			0,005
	60	218	282	372	443	481	494	500	500	501		0,006
	70	222	281	376	443	480	496	501				0,006
	80	213	277	369	450	483	497	501				0,006
	90	215	287	362	446	486	497	501				0,006
	100	218	274	364	445	487	500	501				0,006
	150	215	271	371	447	487	499	501				0,006
	200	211	275	362	452	483	499	501				0,006
	300	206	269	352	440	480	492	500	501			0,007
(1+1) EA	30	255	321	393	467	494	499	501				0,005
	40	257	325	400	465	493	499	501				0,005
	50	258	320	391	461	488	499	501				0,005
	60	258	322	404	465	491	499	501				0,005
	70	257	320	394	458	485	499	500	501			0,005
	80	257	318	408	469	492	499	501				0,004
	90	251	316	390	456	488	500	501				0,005
	100	247	314	388	453	488	498	501				0,005
	150	251	310	387	458	490	498	501				0,005
	200	253	303	377	448	485	498	501				0,006
	300	241	274	314	361	400	440	477	494	500	501	0,013

Tabelle 9.4: Die Ergebnisse der Binomialtests. Die Zahlen geben an, von wie vielen Paaren von Erfolgswahrscheinlichkeiten der entsprechenden Versuchsreihen mit einer Signifikanz von 0,05 gezeigt werden konnte, dass sie um höchstens ε voneinander abweichen. Außerdem ist die durchschnittliche Abweichung der Schätzwerte auf drei Stellen gerundet angegeben.

Test von Hypothese 5: Konzentrierte Brückenkanten führen bei der (1+1) RLS bei sehr wenigen und sehr vielen Brückenkanten zu einer kleineren Erfolgswahrscheinlichkeit als gleichmäßig oder zufällig verteilte und sonst zu einer größeren.

Vergleicht man eine Messreihe mit konzentriert gewählten Kanten mit einer Messreihe mit gleichmäßig oder zufällig gewählten Kanten, so wird durch die Hypothese eine Aufteilung der Messreihen in vier Bereiche nahe gelegt. Im ersten Abschnitt liegt die Kurve mit konzentrierten Brückenkanten unterhalb der anderen Kurve, im zweiten Abschnitt dreht sich dieses Verhalten um und im dritten Abschnitt ist das Verhalten wieder wie im ersten. Der vierte Abschnitt, der für uns am wenigsten interessant ist, besteht aus den Messwerten, die zu den Brückenkantenzahlen gehören, für die bei beiden Messreihen immer eine Gleichfärbung der Cliquen beobachtet wurde (siehe Abbildung 9.2).

Da die Grenzen dieser Bereiche sich für verschiedene Cliquengrößen verschieben, mussten wir zunächst eine Regel aufstellen, nach der wir die Einteilung in Bereiche vornehmen. Für

den Vergleich zweier Messreihen mit gleich großen Cliquen, wobei die Brückenkanten bei der einen konzentriert und bei der anderen gleichmäßig oder zufällig verteilt wurden, haben wir zunächst jedes der 501 Paare von Schätzwerten unabhängig von den anderen mit einem Binomialtest verglichen und getestet, ob mit einer Signifikanz von 0,05 die Aussage getroffen werden kann, dass entweder die eine Erfolgswahrscheinlichkeit größer ist als die andere oder umgekehrt. Um die Diskussion im Folgenden etwas zu vereinfachen stellen wir uns das Ergebnis dieser 501 Binomialtests als einen String der Länge 501 vor, in dem jedes Zeichen für das Ergebnis eines Tests steht. Ein + bedeutet, dass der Test ergeben hat, dass die Erfolgswahrscheinlichkeit für gleichmäßig bzw. zufällig gewählte Kanten größer ist als für konzentrierte, ein − bedeutet das Gegenteil und eine 0 bedeutet, dass keine der beiden Aussagen mit einer Signifikanz von 0,05 mit einem Binomialtest bestätigt werden kann.

Nahe liegend wäre es nun gewesen, das Ende des ersten Bereiches durch das erste − in dem String zu bestimmen. Leider ist diese Art der Einteilung zu sensibel für Messreihen mit leichten Schwankungen, deswegen haben wir uns folgende Heuristik zur Einteilung der Kurven in vier Bereiche überlegt.

- Der erste Abschnitt ist beendet, sobald das erste Mal drei − in dem String vorkommen, zwischen denen sich kein + befindet.
- Der zweite Abschnitt ist beendet, sobald nach dem ersten Bereich das erste Mal zwei + in dem String vorkommen, zwischen denen sich kein − befindet. Ist diese Bedingung nicht erfüllbar, so definieren wird den dritten Abschnitt als leer und der zweite ist beendet, sobald der vierte beginnt.
- Der vierte Abschnitt beginnt, sobald für eine Brückenkantenzahl das erste Mal in beiden Messreihen immer eine Gleichfärbung beobachtet wurde.

Nachdem wir diese Einteilung definiert haben, haben wir jede Messreihe mit konzentrierten Brückenkanten jeweils mit der entsprechenden Messreihe mit zufälligen und der Messreihe mit gleichmäßig verteilten Brückenkanten verglichen. Dazu haben wir zunächst jeweils die entsprechende Einteilung berechnet und die ersten drei Abschnitte der Kurven jeweils mit einem Wilcoxon-Rangsummentest verglichen. Die Ergebnisse finden sich in Tabelle 9.5.

Wir können festhalten, dass die Hypothese mit hoher Signifikanz bestätigt wurde. Die dritte Phase, in der die konzentrierten Brückenkanten wieder zu kleineren Erfolgswahrscheinlichkeiten führen als die gleichmäßig bzw. zufällig gewählten, ist umso ausgeprägter, je größer die Cliquen werden.

Hinter diesen Ergebnissen für konzentrierte Brückenkanten vermuten wir folgende Ursachen.

- Bei einer kleinen Anzahl von Brückenkanten sind die Brückenkanten auf sehr wenige Knoten beschränkt. In den meisten Schritten der Algorithmen werden diese Knoten nicht geflippt, so dass die Brückenkanten in diesen Schritten keinerlei Einfluss auf der Verhalten der Algorithmen haben.

Strategie		Ende der Phasen			Abweichung, Signifikanz					
Cliquengröße		Phase1	Phase 2	Phase 3	Phase 1		Phase 2		Phase 3	
Gleichm.	30	99	341	341	+	0,000	–	0,000		
	40	97	358	358	+	0,000	–	0,000		
	50	91	348	358	+	0,000	–	0,000	–	0,250
	60	87	351	351	+	0,000	–	0,000		
	70	92	293	361	+	0,000	–	0,000	+	0,003
	80	89	280	375	+	0,000	–	0,000	+	0,000
	90	88	254	364	+	0,000	–	0,000	+	0,000
	100	87	259	361	+	0,000	–	0,000	+	0,000
	150	80	225	385	+	0,000	–	0,000	+	0,000
	200	79	200	392	+	0,000	–	0,000	+	0,000
	300	124	199	400	+	0,000	–	0,000	+	0,000
Zufällig	30	93	328	328	+	0,000	–	0,000		
	40	89	348	351	+	0,000	–	0,000	0	1,000
	50	89	352	352	+	0,000	–	0,000		
	60	87	331	357	+	0,000	–	0,000	+	0,820
	70	93	297	362	+	0,000	–	0,000	+	0,000
	80	86	280	375	+	0,000	–	0,000	+	0,000
	90	92	274	368	+	0,000	–	0,000	+	0,000
	100	87	259	361	+	0,000	–	0,000	+	0,000
	150	79	240	369	+	0,000	–	0,000	+	0,000
	200	77	211	385	+	0,000	–	0,000	+	0,000
	300	124	218	400	+	0,000	–	0,000	+	0,000

Tabelle 9.5: Die Ergebnisse der Wilcoxon-Rangsummentests. Ein + bedeutet, dass konzentrierte Kanten zu einer kleineren Erfolgswahrscheinlichkeit geführt haben als gleichmäßig bzw. zufällig gewählte Kanten. Ein – bedeutet das Gegenteil. Die 0 bedeutet, dass keine Abweichung festgestellt wurde. Die Signifikanzen sind auf 3 Stellen gerundet.

- Bei einer mittleren Anzahl von Brückenkanten ist die konzentrierte Verteilung besser als die zufällige oder die gleichmäßige Verteilung, da hier ähnlich wie beim Treppengraphen die Knoten mit vielen Brückenkanten den Anstoß dafür geben können, Knoten umzufärben und sie der Farbe der anderen Clique anzupassen. Die Knoten mit vielen Brückenkanten wirken also quasi als „Eisbrecher“, die eine hohe Tendenz haben, sich der Färbung der anderen Clique anzuschließen und dadurch anderen Knoten in der eigenen Clique den Farbwechsel zu erleichtern.

- Bei einer hohen Anzahl von Brückenkanten vermuten wir jedoch, dass es dann wenig Sinn macht, Knoten, die bei der zufälligen oder gleichmäßigen Verteilung im Durchschnitt bereits recht viele Brückenkanten haben, in der konzentrierten Verteilung noch mehr Brückenkanten zu geben. Die Kehrseite der Medaille ist dann nämlich, dass einige Knoten der Clique überhaupt keine Brückenkanten besitzen.

Test von Hypothese 6: Der (1+1) EA hat eine größere Erfolgswahrscheinlichkeit als die (1+1) RLS.

Die Hypothese, dass der (1+1) EA für alle Cliquengrößen, alle Zahlen an Brückenkanten und alle Strategien zur Wahl der Brückenkanten eine durchschnittlich größere Erfolgswahrscheinlichkeit hat als die (1+1) RLS, konnte mit dem Wilcoxon-Rangsummentest bestätigt werden. Für alle 33 Kombinationen aus Algorithmus und Strategie zur Kantenwahl wurde dies mit einer auf drei Stellen gerundeten Signifikanz von 0,000 bestätigt.

Eine mögliche Erklärung für dieses Phänomen ist der Effekt von k-Bit-Mutationen für gerades k. Wenn am Anfang eines Laufs in beiden Cliquen ungefähr gleich viele Knoten mit Nullen und mit Einsen gefärbt sind, ist die Wahrscheinlichkeit relativ groß, dass eine Mutation eine gerade Anzahl von Bits flippt und dabei die Zahl 0-gefärbter Knoten in beiden Cliquen konstant bleibt.

Ein solcher Schritt wird genau dann akzeptiert, wenn dabei die Färbung der Brückenkanten nicht verschlechtert wird. Hier wirken die Brückenkanten als Zünglein an der Waage; sie haben in diesem Fall einen entscheidenden Einfluss auf die Fitness.

Während sich bei der randomisierten lokalen Suche in jedem akzeptierten Schritt in einer Clique die Farbmehrheit verändert, ergibt sich beim (1+1) EA die Möglichkeit, die Färbung der Brückenkanten zu verbessern, ohne dass die Farbmehrheiten sich verändern. Auch wenn es den Anschein hat, als würde der Optimierungsprozess in solchen Schritten langsamer laufen, könnten diese Schritte doch helfen, die Färbung der Brücken zu verbessern und dadurch die Erfolgswahrscheinlichkeit des (1+1) EA gegenüber der RLS zu erhöhen.

In weiteren Erkundungsexperimenten wurde der (1+1) EA unter diesen Gesichtspunkten weiter beobachtet, und zwar bei einer Brückenkantendichte von $q(m, n) = 1/4$, bei der große Unterschiede zwischen dem (1+1) EA und der (1+1) RLS festgestellt worden waren. Es zeigte sich bei verschiedenen Knotenzahlen, dass unter allen Generationen, in denen eine echte Fitnessverbesserung auftrat, in etwa 1/10 der Fälle die Anzahl 0-gefärbter Knoten in beiden Cliquen konstant blieb. Dabei waren die Generationen, in denen ein solches Ereignis auftrat, über den gesamten Lauf relativ gleichmäßig verteilt. Wir vermuten daher, dass der (1+1) EA bei entsprechenden Brückenkantendichten in einem konstanten Anteil der Schritte ausschließlich die Färbung der Brücken verbessert.

Test der Hypothesen 7 und 8 zur Erfolgsvorhersage mittels Potenzialfunktionen

Für den Test der Hypothesen 7 und 8 zur Erfolgsvorhersage mittels Potenzialfunktionen wurden gemeinsame Experimente durchgeführt, deren Anordnung im Folgenden beschrieben werden soll.

Der untersuchte Algorithmus ist ein (1+1) EA mit einem Stoppkriterium, das den Lauf stoppt, sobald alle Cliquen jeweils einheitlich gefärbt sind. Für eine feste Zahl von Knoten und Brückenkanten wurden jeweils 1000 unabhängige Läufe durchgeführt und für die jeweiligen Tests mit den Potenzialfunktionen φ und ψ die Rate der korrekt vorhergesagten Läufe aufgezeichnet. Die Knotenzahl der Graphen wurde auf einer linearen Skala variiert mit Werten $n = 128, n = 256, \ldots, n = 1024$. Die Verteilung der Brückenkanten wurde für neue Werte

für Knoten und Kanten jeweils zufällig erstellt und blieb dann für alle 1000 Läufe dieses Experiments fest.

Da nach den Vorbetrachtungen kleine Zahlen von Brückenkanten nur wenige Auswirkungen haben, wurden für die Zahl der Brückenkanten quadratische Werte gewählt. Statt die Zahl der Brückenkanten für jedes n anzugeben, sprechen wir im Folgenden von der Dichte der Brückenkanten $q(m,n)$, dem Anteil an Brückenkanten bezogen auf den maximal möglichen Wert $n^2/4$. Um trotz Einschränkung auf quadratische Werte einen möglichst großen Bereich abzudecken, wurden die Konstanten in der O-Notation nach einem exponentiellen Schema variiert: getestet wurden Dichten von $q(m,n) = 1, 1/2, 1/4, \ldots, 1/512$.

Test von Hypothese 7 zur knotenbasierten Erfolgsvorhersage

In Tabelle 9.6 sind die gemessenen Raten an korrekt vorhergesagten Läufen aufgeführt abhängig von der Knotenzahl des Graphen und der Dichte der Brückenkanten.

	Dichte der Brückenkanten									
Knoten	1/512	1/256	1/128	1/64	1/32	1/16	1/8	1/4	1/2	1
128	0,750	0,767	0,763	0,787	0,808	0,812	0,849	0,914	0,951	0,999
256	0,797	0,788	0,807	0,824	0,820	0,808	0,845	0,930	0,975	1,000
384	0,826	0,829	0,779	0,810	0,843	0,842	0,876	0,938	0,968	1,000
512	0,807	0,822	0,818	0,827	0,833	0,856	0,871	0,945	0,968	1,000
640	0,835	0,831	0,844	0,869	0,850	0,852	0,891	0,940	0,977	1,000
768	0,853	0,812	0,854	0,854	0,853	0,854	0,882	0,959	0,980	1,000
896	0,854	0,837	0,821	0,860	0,854	0,876	0,888	0,960	0,984	1,000
1024	0,863	0,864	0,861	0,875	0,870	0,875	0,912	0,960	0,985	1,000

Tabelle 9.6: Die gemessenen Raten der mit knotenbasierter Erfolgsvorhersage korrekt vorhergesagten Läufe des (1+1) EA.

Man kann aus der Tabelle ablesen, dass die Erfolgsvorhersage um so besser ist, je mehr Brückenkanten vorhanden sind und je größer die Knotenzahl ist, da die Werte sowohl in den Zeilen als auch in den Spalten annähernd monoton steigen. Letzteres entspricht unserer Hypothese und soll im Folgenden mit statistischen Tests untermauert werden.

Zur Auswertung wurden Variablen für die verschiedenen Knotenzahlen n gebildet. Die Daten jeder Variablen entsprachen der gemessenen Rate korrekt vorhergesagter Läufe über die verschiedenen Brückenkantendichten $q(m,n) = 1/512, 1/256, \ldots, 1$.

Dann wurden benachbarte Variablen auf signifikante Unterschiede überprüft; es gab also sieben Tests mit den Variablenpaaren $(n = 128, n = 256), (n = 256, n = 384), \ldots, (n = 896, n = 1024)$. Als Test wurde der Wilcoxon-Rangsummentest für verbundene Datenreihen verwendet, der die Werte beider Variablen paarweise für gleiche Brückenkantendichten miteinander vergleicht.

Leider konnte die Hypothese, dass alle benachbarten Variablen voneinander abweichen und eine monoton steigende Folge bilden, nicht mit ausreichender Signifikanz bestätigt werden.

Daher wurde eine Vergröberung der Daten vorgenommen: die Variablen wurden in vier Intervalle von je zwei benachbarten Variablen partitioniert und die Daten der Variablen eines Intervalls zu einer gemeinsamen Datenreihe zusammengefasst. So ergaben sich vier Variablen für $n \in \{128, 256\}, n \in \{384, 512\}, n \in \{640, 768\}, n \in \{896, 1024\}$. Der Wilcoxon-Rangsummentest vergleicht also nun wieder Trefferquoten für gleiche Brückenkantendichten, jedoch für Knotenzahlen, die eine Differenz von 256 (statt 128) haben.

Mit Hilfe dieser Vergröberung konnte Hypothese 7 für die betrachteten Szenarien bestätigt werden: die Variablenpaare $(n \in \{128, 256\}, n \in \{384, 512\})$, $(n \in \{384, 512\}, n \in \{640, 768\})$, $(n \in \{640, 768\}, n \in \{896, 1024\})$ wichen statistisch sehr signifikant voneinander nach oben ab. Die Signifikanzwerte der drei Tests, auf drei Nachkommastellen gerundet, lagen bei $p = 0{,}000, p = 0{,}001$ und $p = 0{,}008$.

Test von Hypothese 8 zur brückenbasierten Erfolgsvorhersage

In Tabelle 9.7 sind die gemessenen Raten korrekt vorhergesagter Läufen aufgeführt, abhängig von der Knotenzahl des Graphen und der Dichte der Brückenkanten.

	Dichte der Brückenkanten									
Knoten	1/512	1/256	1/128	1/64	1/32	1/16	1/8	1/4	1/2	1
128	0,548	0,582	0,659	0,659	0,716	0,791	0,864	0,902	0,883	0,963
256	0,593	0,609	0,681	0,721	0,787	0,830	0,920	0,930	0,929	0,985
384	0,618	0,648	0,690	0,758	0,828	0,877	0,929	0,955	0,923	0,996
512	0,620	0,643	0,740	0,777	0,861	0,922	0,950	0,959	0,936	1,000
640	0,608	0,663	0,743	0,801	0,884	0,920	0,956	0,961	0,959	1,000
768	0,624	0,670	0,757	0,823	0,880	0,933	0,973	0,971	0,959	1,000
896	0,652	0,686	0,738	0,835	0,892	0,943	0,979	0,988	0,956	0,999
1024	0,672	0,700	0,742	0,841	0,913	0,951	0,980	0,972	0,961	1,000

Tabelle 9.7: Die gemessenen Raten der mit brückenbasierter Erfolgsvorhersage korrekt vorhergesagten Läufe des (1+1) EA.

Die Auswertung verlief genau so wie bei der knotenbasierten Erfolgsvorhersage; auch die Vergröberung der Daten wurde genau so durchgeführt. Es zeigte sich, dass die Signifikanzen sowohl vor als auch nach der Vergröberung besser waren als die entsprechenden Signifikanzen bei der knotenbasierten Erfolgsvorhersage.

Die Tests der Paare $(n \in \{128, 256\}, n \in \{384, 512\})$, $(n \in \{384, 512\}, n \in \{640, 768\})$, $(n \in \{640, 768\}, n \in \{896, 1024\})$ mit dem Wilcoxon-Rangsummentest für die brückenbasierte Erfolgsvorhersage ergaben auf drei Nachkommastellen gerundete Signifikanzwerte von $p = 0{,}000, p = 0{,}000$ und $p = 0{,}002$.

Fazit zum Ising-Modell auf Cliquen

Wir haben die Wahrscheinlichkeiten untersucht, mit denen der (1+1) EA und die (1+1) RLS auf zwei durch Brückenkanten verbundenen Cliquen ein globales Optimum erreichen, ohne

vorher ein lokales Optimum zu durchlaufen. Wir haben die Rolle näher untersucht, die die Strategie zur Verteilung der Brückenkanten spielt. Als unser Ansicht nach interessantes Ergebnis können wir festhalten, dass der (1+1) EA in allen von uns getesteten Situationen eine größere Erfolgswahrscheinlichkeit hat als die (1+1) RLS. Wir haben eine Vermutung aufgestellt, die eine Intuition vermittelt, warum die Erfolgswahrscheinlichkeit des (1+1) EA größer ist. Eine genaue theoretische Analyse erscheint jedoch sehr kompliziert.

Die Verteilung der Brückenkanten im Treppengraphen stellt vermutlich ein Worst-Case-Beispiel dar für die Zeit, bis die RLS und der (1+1) EA den ersten lokal optimalen Punkt erreichen. Unter der Annahme, dass einer der beiden Pfade auf seiner ersten Hälfte erreicht wird, haben wir mit einer theoretischen Analyse untere Schranken von $\Omega(n^2)$ bewiesen. Unsere Experimente lassen vermuten, dass diese Annahme zumindest für die getesteten n erfüllt ist.

Weiterhin haben wir mit Hilfe zweier Maße φ und ψ für Szenarien mit einer zufällig gewählten Menge von $\Theta(n^2)$ Brückenkanten gezeigt, dass bereits nach einer kleinen Anfangsphase von $n^{3/4}$ Generationen fest steht, ob der (1+1) EA ein globales Optimum erreicht oder in ein lokales Optimum läuft. Die Richtung, in die der Optimierungsprozess läuft, wird also in den ersten $n^{3/4}$ Schritten entschieden, wenn sich noch keine großen Mehrheiten an Farben heraus gebildet haben. Die große Herausforderung einer umfassenden Analyse des Ising-Modells auf Cliquen besteht also unserer Meinung nach darin, die Vorgänge in der Anfangsphase zu verstehen und zu erfassen, da sich hier Mehrheiten herauskristallisieren und sich der Ausgang des Laufs entscheidet.

Die Beschränkung auf die Zahl 1-gefärbter Knoten, wie sie in φ vorkommt, hat sich aus theoretischer Sicht für die (1+1) randomisierte lokale Suche als hilfreich erwiesen, wenn die maximale Zahl adjazenter Brückenkanten klein ist. Der Anteil ψ korrekt gefärbter Brücken gibt nicht nur Hinweise auf den Fitnesswert, sondern auch auf die Ähnlichkeit beider Cliquen. Auch mit dieser Potenzialfunktion lässt sich für die betrachteten Szenarien schon früh feststellen, ob ein Lauf in einem globalen oder einem lokalen Optimum enden wird.

Die Potenzialfunktionen φ und ψ können bei einer theoretischen Analyse hilfreich sein. Alle möglichen Verteilungen von Brückenkanten sind nur schwer überschaubar und die Betrachtung der konkreten Verteilung aller Brückenkanten ist ohne massive Einschränkungen an die Graphen wenig sinnvoll. Ein erstes Ziel einer theoretischen Analyse kann daher sein, die Menge an Informationen zu reduzieren. Beide Potenzialfunktionen bieten Anregungen in diese Richtung.

9.1.2 Das Ising-Modell auf Tori

Um unsere in Abschnitt 8.1.2 aufgestellten Hypothesen testen zu können, haben wir mit FrEAK den (1+1) EA auf dem Torus vielfach simuliert. Damit wir stabile Ringe erkennen und den aktuellen Lauf dann abbrechen konnten, wurde ein neues Stoppkriterium namens *Ising on Torus* hinzugefügt.

Nun war es uns möglich, die Suchraumdimension langsam von 10×10 Knoten auf 50×50 Knoten zu steigern. In jeder Dimension wurde der (1+1) EA dann laufen gelassen (je nach

Dimension zwischen 250 und 2000 Simulationen) und für jeden Lauf die Generationenanzahl bis zum Ring bzw. Optimum und ggf. die Ringbreite bestimmt.

Dabei wurden dieselben Dimensionen und Laufanzahlen jeweils für Tori mit und ohne Diagonalkanten benutzt, um die Ergebnisse miteinander vergleichen zu können.

Anschließend haben wir mittels SPSS die Mittelwerte und Standardabweichungen dieser Werte für jede Dimension bestimmt. Tabellen 9.8 und 9.9 zeigen die erhaltenen Ergebnisse.

Größe	Läufe	Gesamt		Optimum		Ring		Ringbreite		rel. H'keit
		Mittel	SD	Mittel	SD	Mittel	SD	Mittel	SD	
10x10	2000	3290	1507	2870	1081	4342	1868	4,02	0,82	29%
15x15	2000	17572	8692	14179	5323	24718	9989	6,00	1,06	32%
20x20	2000	57041	31547	45798	25087	79661	31084	8,07	1,47	33%
25x25	1000	147100	77419	116591	52781	207663	82805	10,09	1,80	34%
30x30	1000	316118	218452	262326	227984	423864	147889	12,14	2,10	33%
35x35	500	584771	423834	482323	442851	808590	266761	13,98	2,59	31%
40x40	250	1073743	886631	903425	975494	1416429	531767	16,11	3,15	33%
45x45	250	1559913	908596	1249493	716937	2227377	807835	18,12	3,60	32%
50x50	250	2344026	1440359	1967374	1367094	3372792	1097960	20,49	3,16	27%

Tabelle 9.8: Überblick über die Ergebnisse auf Tori ohne Diagonalkanten.

Größe	Läufe	Gesamt		Optimum		Ring		Ringbreite		rel. H'keit
		Mittel	SD	Mittel	SD	Mittel	SD	Mittel	SD	
10x10	2000	2461	1377	1967	678	3896	1819	4,13	0,80	26%
15x15	2000	13220	8384	9221	2940	21639	9777	6,09	1,02	32%
20x20	2000	42902	28046	28317	8640	70714	31072	8,35	1,33	34%
25x25	1000	106689	68686	68192	20286	174243	71502	10,43	1,54	36%
30x30	1000	219267	136719	137730	39259	364221	127866	12,54	2,00	36%
35x35	500	438206	295142	246021	76483	712490	274056	14,68	2,12	41%
40x40	250	730931	475571	428805	109360	1215593	433816	16,78	2,76	38%
45x45	250	1211809	770635	661101	191610	1924192	643033	18,26	3,00	44%
50x50	250	1660677	1167962	995223	294341	2929682	1160281	20,28	3,96	34%

Tabelle 9.9: Überblick über die Ergebnisse auf Tori mit Diagonalkanten.

Hypothese 9: Die Ringwahrscheinlichkeit ist konstant.

Die Abbildung 9.3 zeigt den beobachteten Verlauf der relativen Ringhäufigkeiten auf Tori der Größe 10,15,...,45,50 mit bzw. ohne Diagonalkanten. Betrachten wir zunächst den Fall ohne Diagonalkanten. Die Schwankungen liegen in einem Bereich von ca. 10 Prozent, wohingegen die Dimension des Suchraums von 100 auf 2500 Knoten wächst. Um die Hypothese nun testen zu können, haben wir mit SPSS eine lineare Regression durchgeführt. Die ermittelte Funktion lautet:

$$32{,}625242 - 0{,}001003 \cdot n.$$

Sie sieht also wie eine konstante Funktion aus, allerdings können wir das von SPSS berechnete Signifikanzniveau nicht verwenden, da es von einer zugrunde liegenden Normalverteilung der Messwerte ausgeht. Dementsprechend haben wir den in Abschnitt 9.1.1 erläuterten Test verwendet und mit einer Signifikanz von 0,042 festgestellt, dass die Ringwahrscheinlichkeit in das Intervall von 20% bis 40% fällt.

Dieses Intervall ist offensichtlich nicht gerade klein. Das hat mehrere Gründe, vor allem stört dabei die relativ geringe Anzahl an Testläufen in höherer Dimension. Daher haben wir den Test noch einmal durchgeführt, diesmal jedoch nur mit den Dimensionen, in denen wir jeweils 1000 Läufe zur Verfügung hatten.

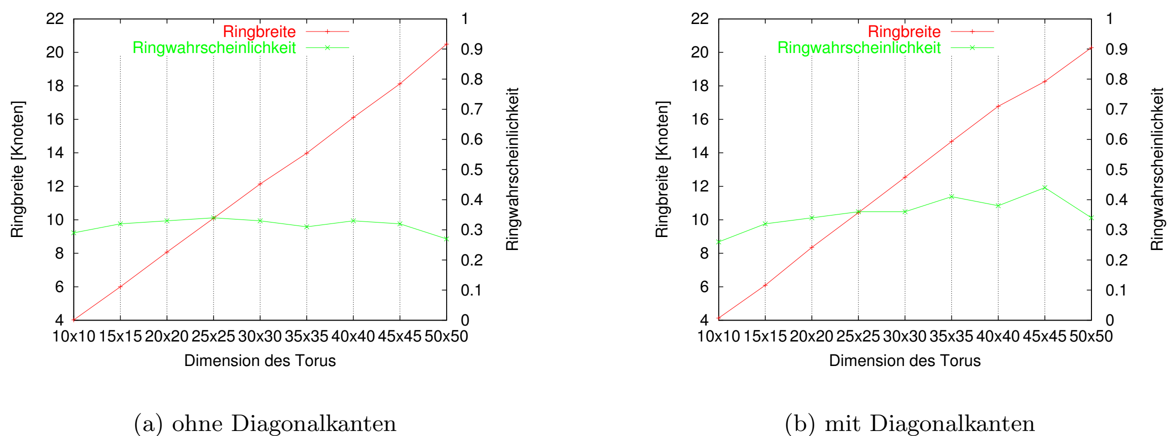
Nun beträgt die Signifikanz 0,030 bei einem Intervall von 24% bis 27%. In diesem Bereich ist also ein erheblich besseres Ergebnis zu erzielen.

Im Fall des Torus mit Diagonalkanten sieht die Situation etwas anders aus. Wie aus der Abbildung 9.3 erkennbar, schwanken die Ringwahrscheinlichkeiten nun sogar um 20 Prozent. Allerdings ist eine leicht steigende Tendenz erkennbar. Daher lautet die von SPSS berechnete Funktion:

$$31{,}656534 + 0{,}003759 \cdot n.$$

Wir gehen aber weiterhin von einer konstanten Wahrscheinlichkeit aus und testen darauf hin. Diesmal können wir ein Intervall von 20% bis 50% mit einer Signifikanz von 0,045 bei Verwendung aller Daten und von 23% bis 40% mit einer Signifikanz von 0,024 bei ausschließlicher Verwendung der Daten mit mindestens 1000 Testläufen berechnen.

Insgesamt kann man also sagen, dass sich unsere Hypothese im Fall ohne Diagonalkanten auf Tori der Größe 10×10 bis 30×30 auf jeden Fall bestätigen ließ. Darüber hinaus war die Anzahl an Testläufen zu gering. Auf Tori mit Diagonalkanten ließ sich die Hypothese im Rahmen der Tests leider nicht zeigen.



(a) ohne Diagonalkanten

(b) mit Diagonalkanten

Abbildung 9.3: Verlauf der Ringwahrscheinlichkeiten und -breiten auf den untersuchten Tori.

Hypothese 10: Die Breite der Ringe ist linear.

Die Untersuchung dieser Hypothese verlief weitaus erfreulicher als die der Hypothese 9. Alleine schon anhand der Abbildung 9.3 wird die Hypothese wenigstens intuitiv gestützt. Eine Regressionsanalyse mit SPSS ergab mit einer Signifikanz von $p < 0{,}001$ folgende Funktion:

$$\text{E(Ringbreite)} = \begin{cases} 0{,}407367 \cdot \sqrt{n} - 0{,}107667 & \text{ohne Diagonalkanten} \\ 0{,}407400 \cdot \sqrt{n} + 0{,}171333 & \text{mit Diagonalkanten} \end{cases}$$

Zusammenfassend wurde die Hypothese „Die Breite der Ringe ist linear in der Seitenlänge des quadratischen Torus“ (in dem untersuchten Rahmen) voll bestätigt.

Hypothese 11: Kleinere Wahrscheinlichkeit für diagonale Ringe.

Unsere Experimente ergaben die in Tabelle 9.10 gezeigten relativen Häufigkeiten für stabile Ringe.

Ringtyp	10x10	15x15	20x20	25x25	30x30	35x35	40x40	45x45	50x50
horizontal	48%	47%	46%	46%	46%	48%	48%	49%	42%
vertikal	50%	47%	46%	45%	48%	44%	46%	46%	42%
diagonal	2%	5%	8%	9%	7%	8%	6%	6%	16%

Tabelle 9.10: Relative Häufigkeiten der verschiedenen stabilen Ringe.

Wie nicht anders zu erwarten ergab der Wilcoxon-Rangsummentest, der auf Gleichheit der zugrunde liegenden Erwartungswerte testet, dass die beobachteten relativen Häufigkeiten für horizontale vs. diagonale, wie auch für vertikale vs. diagonale Ringe mit Signifikanz 0,001 nicht aus derselben Verteilung stammen. Dies ist trotz der geringen Stichprobenzahl eine hohe Signifikanz. Aufgrund der geringen Stichprobenzahl ergab der Test bei horizontalen vs. vertikalen Ringen nur eine Signifikanz von ca. 0,34. Horizontale und vertikale Ringe sind jedoch aufgrund theoretischer Überlegungen vollkommen äquivalent.

Unsere Hypothese 11 hat sich im untersuchten Bereich voll bestätigt.

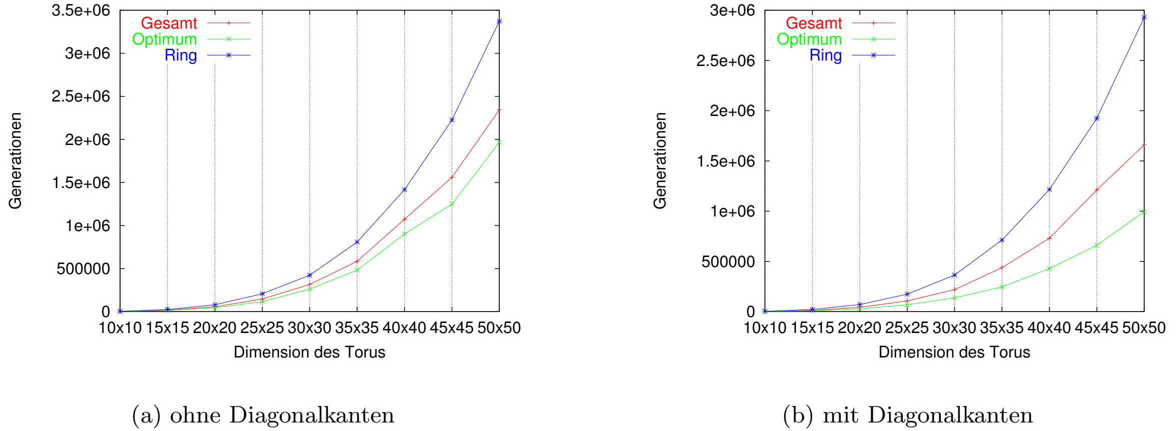
Hypothese 12: $O(n^{3/2})$ Generationen reichen aus.

Die in Tabellen 9.8 und 9.9 genannten Experimentdaten über die Laufzeit bis zum Erreichen eines globalen bzw. stabilen lokalen Optimums wurden in Abbildung 9.4 noch einmal dargestellt.

Dargestellt sind also die mittleren Laufzeiten bis zum Erreichen eine stabilen Rings, des globalen Optimums bzw. das arithmetische Mittel über alle gemessenen Laufzeiten.

Die Hypothese lautete, dass diese Laufzeiten im Wesentlichen durch ein Polynom von Grad 3 bezüglich der Kantenlänge des Torus beschrieben werden. Um dies zu bestätigen, wandten wir wiederum eine Regressionsanalyse an und erhielten als Ergebnis folgende Funktionen.

$$\text{E(Generationen)} \approx \begin{cases} 32n^{3/2} - 801n + 5734\sqrt{n} + 2581 & \text{ohne Diagonalkanten} \\ 20n^{3/2} - 227n - 4995\sqrt{n} + 64870 & \text{mit Diagonalkanten} \end{cases}$$



(a) ohne Diagonalkanten (b) mit Diagonalkanten

Abbildung 9.4: Laufzeiten bis zum Erreichen des globalen bzw. stabilen lokalen Optimums.

$$\text{E(Gen. bis Optimum)} \approx \begin{cases} 30n^3/2 - 901n + 10518\sqrt{n} - 42342 & \text{ohne Diagonalkanten} \\ 17n^3/2 - 634n + 9907\sqrt{n} - 52135 & \text{mit Diagonalkanten} \end{cases}$$

$$\text{E(Gen. bis Ring)} \approx \begin{cases} 60n^{3/2} - 2283n + 34279\sqrt{n} - 173380 & \text{ohne Diagonalkanten} \\ 53n^{3/2} - 2065n + 31730\sqrt{n} - 163709 & \text{mit Diagonalkanten} \end{cases}$$

Insgesamt liegen wir also im Bereich $O(n^{3/2})$ (wobei wir damit bewusst keine Aussage für Dimensionen n machen wollen, die nicht im untersuchten Bereich liegen). Damit hat sich unsere Hypothese voll und ganz bestätigt.

Weiterhin erkennt man, dass die Anzahl an Generationen bis zum Erreichen eines stabilen Rings durchschnittlich größer ist als die Anzahl an Generationen, bis das Optimum erreicht ist (natürlich gilt diese Aussage nur, falls das Optimum überhaupt in der betrachteten Zeit erreicht wird, also zwischenzeitlich kein stabiler Ring erreicht wurde).

Hierfür gibt es ein theoretisches Indiz. Nehmen wir an, eine Population (in unserem Fall ein Individuum) befindet sich „kurz“ vor dem Optimum. In diesem Fall sind fast alle Knoten gleich gefärbt und es gibt nur wenige „Inseln“, die eine andere Farbe haben. Diese können relativ leicht umgefärbt werden, da meist ein einziger anders gefärbter Knoten bereits eine Fitnessverbesserung nach sich zieht. Solche speziellen 1-Bit-Flips haben eine Wahrscheinlichkeit von $\Omega\left(\frac{1}{n}\right)$, sind also nicht zu unwahrscheinlich. Somit sollten wir das Optimum aus dieser Situation gut erreichen können.

Befinden wir uns hingegen „kurz“ vor dem Erreichen eines stabilen Rings, ist häufig bereits eine Seite des Ringes stabil (es sind keine Flips mit weniger als $\sqrt{n}$ Bits mehr möglich), wohingegen die andere Seite des Rings wie in Abbildung 9.5 aussieht. Dort sind noch 1-Bit-Flips möglich, die den Fitnesswert nicht mehr erhöhen, aber denselben Fitnesswert erzielen. Damit befinden wir uns also auf einem Plateau der Größe $\Omega(\sqrt{n})$, auf dem wir einen Random

Walk durchführen. Offensichtlich gibt es (unter der Annahme, dass wir nicht mehr als 4-Bit-Mutationen ausführen) nur eine konstante Anzahl von Flips, die wieder Individuen auf dem Plateau erzeugen. Dementsprechend ist es durchaus möglich, dass die Läufe, die in einem Ring enden, wesentlich länger dauern, als die Läufe, die in einem globalen Optimum enden.

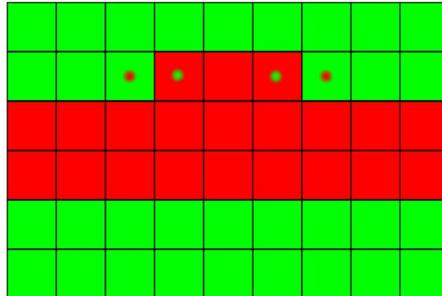


Abbildung 9.5: Mögliche 1-Bit-Flips in einer Situation „kurz“ vor dem stabilen Ring.

Fazit zum Ising-Modell auf Tori

Wir haben also mit Experimenten hauptsächlich die Wahrscheinlichkeit des Auftretens von stabilen Ringen und deren Größe untersucht. Dabei wurden die aufgestellten Hypothesen weitestgehend bestätigt. In dem von uns untersuchten Rahmen ist die Wahrscheinlichkeit des Auftretens stabiler Ringe auf Tori ohne Diagonalkanten konstant und liegt zwischen 20 und 40 Prozent. Unsere Tests waren für diagonale Ringe nicht aussagekräftig genug, um auch hier diese Annahme zu bestätigen.

Einwandfrei wurde jedoch experimentell gezeigt, dass die Ringwahrscheinlichkeit linear in der Kantenlänge des Torus wächst, und zwar sowohl für Tori mit als auch für Tori ohne Diagonalkanten.

Ebenso wurde nachgewiesen, dass die Wahrscheinlichkeit des Auftretens diagonaler Ringe maßgeblich geringer ist als die des Auftretens horizontaler bzw. vertikaler Ringe. Es ergab sich ein Verhältnis von ca. 1 zu 6.

Die Worst-Case-Abschätzung der zugrunde liegenden Diplomarbeit [10] für die erwartete Zeit bis zum Finden eines stabilen Rings bzw. eines Optimums inspirierte uns zu einer Auswertung unserer Läufe in dieser Richtung. Die Regression ergab, dass (in dem untersuchten Rahmen) eine in der Seitenlänge des Torus kubische Funktion die erwartete Laufzeit am besten beschreibt.

9.1.3 Das Ising-Modell auf booleschen Hypercubes

Für alle Tests wurden für jeden Hypercube der Dimension 1 bis 22 jeweils 1000 Läufe des (1+1) EA und 1000 Läufe der RLS durchgeführt. Ein Lauf wurde angehalten, sobald eine Färbung erreicht wurde, in der keine Fitness verbessernde 1-Bit-Mutation mehr möglich war. Da sich alle Untersuchungen auf Hypercubes der Dimension 22 oder weniger beziehen, können

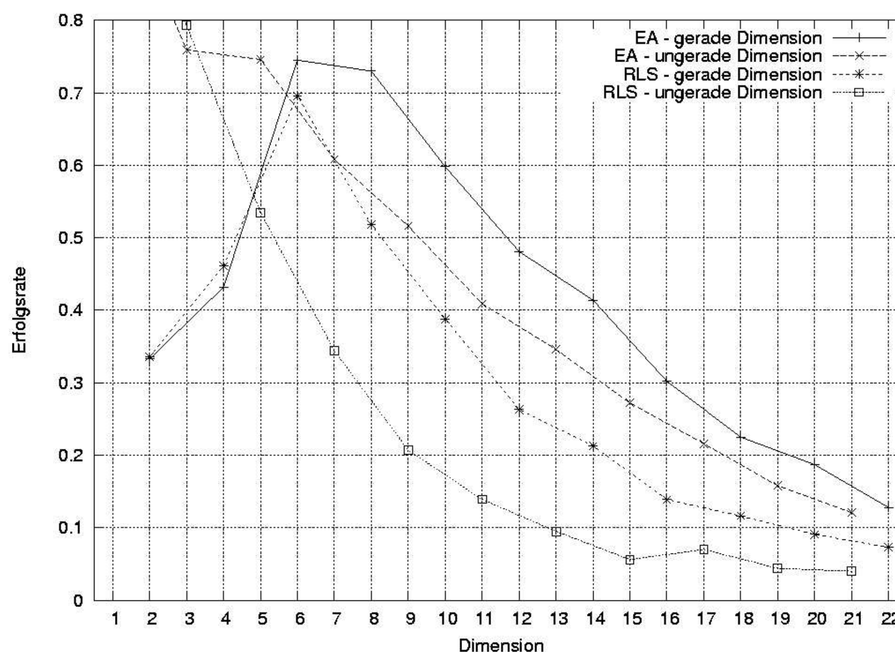


Abbildung 9.6: Gemessene Erfolgsraten der Algorithmen.

keinerlei Aussagen über größere Dimensionen getroffen werden. Es können jedoch Tendenzen aufgezeigt werden, die Anlass und Anhaltspunkte für allgemeinere theoretische Analysen geben können. Es besteht die Hoffnung, dass die gelieferten Anhaltspunkte nicht irreführend sind, da ein Hypercube der Dimension 22 bereits einer beträchtlichen Suchraumdimension von 2^{22} entspricht.

Hypothese 13: Die Wahrscheinlichkeit, ein globales Optimum zu erreichen, sinkt mit wachsender Dimension.

Diese Hypothese ist ungünstig formuliert. Durch die grundlegenden Unterschiede zwischen den Hypercubes gerader Dimension und den Hypercubes ungerader Dimension scheint es sinnvoller, diese getrennt zu betrachten. Dass dieses Vorgehen sinnvoll ist, wird sich durch die Tests zu Hypothese 15 („Die Wahrscheinlichkeit, ein globales Optimum zu erreichen, ist für ungerade Dimensionen geringer als für gerade Dimensionen.“) bestärkt.

Die Kurvenverläufe sind in Abbildung 9.6 dargestellt. Die Ergebnisse der Tests finden sich in Tabelle 9.11.

Für jeden Algorithmus und jede Dimension entstand ein Datensatz mit jeweils 1000 Werten. Geeignete Paare dieser Datensätze wurden dann mit dem Mann-Whitney-Test auf signifikante Unterschiede getestet. Dabei wurden (1+1) EA und RLS gänzlich getrennt behandelt und ein Datensatz zur Dimension i wurde mit dem Datensatz zur Dimension $i + 2$ verglichen.

Wie man sieht, sind die Signifikanzwerte vor allem im mittleren Bereich der gemessenen Dimensionen sehr klein. Gerade in den kleinen geraden Dimensionen steigt die Erfolgsrate

	Signifikanz	
Dimension	(1+1) EA	RLS
1 – 3	0,000	0,000
3 – 5	0,266	0,000
5 – 7	0,000	0,000
7 – 9	0,000	0,000
9 – 11	0,000	0,000
11 – 13	0,002	0,001
13 – 15	0,000	0,001
15 – 17	0,002	0,916
17 – 19	0,000	0,006
19 – 21	0,041	0,483
2 – 4	1,000	1,000
4 – 6	1,000	1,000
6 – 8	0,243	0,000
8 – 10	0,000	0,000
10 – 12	0,000	0,000
12 – 14	0,002	0,003
14 – 16	0,000	0,000
16 – 18	0,000	0,069
18 – 20	0,021	0,040
20 – 22	0,000	0,081

Tabelle 9.11: Die Ergebnisse der Mann-Whitney-Tests. In der Zeile mit den Dimensionen „$x - y$“ wurde jeweils für den (1+1) EA und die RLS getestet, mit welcher Signifikanz die Erfolgsrate für Dimension y kleiner ist als die für Dimension x. Die Signifikanzen sind auf drei Stellen gerundet.

jedoch zunächst stark an. Dies sind jedoch Graphen mit sehr wenigen Knoten, in denen es vermutlich noch zu einigen Randerscheinungen kommt. Dass die Signifkanzwerte in größeren Dimensionen schließlich wieder ansteigen leuchtet ein, da die Wahrscheinlichkeit das Optimum zu erreichen, hier vermutlich nur noch schwach fällt.

Hypothese 14: Die Wahrscheinlichkeit, ein globales Optimum zu erreichen, ist für die RLS geringer als für den (1+1) EA.

Zum Testen dieser Hypothese wurden die gleichen Datensätze wie für die vorangegangene Hypothese gebildet. Nun wurden aber immer Datensätze getestet, die zur gleichen Dimension aber zu verschiedenen Algorithmen gehören. Die Ergebnisse finden sich in Tabelle 9.12.

Auch hier wird deutlich, dass die Signifikanz in sehr kleinen Dimensionen ausgesprochen schlecht ist. Zwischen Dimension 5 und Dimension 22 hat sich diese Hypothese allerdings bestätigt. Auch hier scheinen also in kleinen Dimension noch spezielle Effekte aufzutreten.

Dimension	Signifikanz
1	1,000
2	0,575
3	0,970
4	0,914
5	0,000
6	0,010
7	0,000
8	0,000
9	0,000
10	0,000
11	0,000
12	0,000
13	0,000
14	0,000
15	0,000
16	0,000
17	0,000
18	0,000
19	0,000
20	0,000
21	0,000
22	0,000

Tabelle 9.12: Die Ergebnisse der Mann-Whitney-Tests. In jeder Zeile ist angegeben, mit welcher Signifikanz die RLS in der entsprechenden Dimension eine schlechtere Erfolgsrate als der (1+1) EA hat. Die Signifikanzen sind auf drei Stellen gerundet.

Hypothese 15: Die Wahrscheinlichkeit, ein globales Optimum zu erreichen, ist für ungerade Dimensionen i geringer als für die geraden Dimensionen $i-1$ und $i+1$.

Diese Hypothese lässt sich testen, indem man jede Dimension $2i+1$ gegen Dimension $2i$ und $2i+2$ testet. Liegt die Erfolgsrate für jede Dimension $2i+1$ signifikant unter der für Dimension $2i$ und $2i+2$, kann die Hypothese als bestätigt gelten. Genau dieser Test wurde sowohl für den (1+1) EA auch als für die RLS durchgeführt. Die Ergebnisse sind in Tabelle 9.13 dargestellt.

Wir sehen, dass wir in geringen Dimensionen kein signifikantes Ergebnis bekommen. Das ist nicht überraschend, da auch die Tests der beiden anderen Hypothesen zur Erfolgsrate ein abweichendes Verhalten in kleinen Dimensionen aufgezeigt haben. Für die Dimensionen zwischen 7 und 22 kann die Hypothese zumindest für die RLS bestätigt werden. Für die Dimensionen 15, 17 und 21 sind die Signifikanzwerte für den (1+1) EA allerdings zu schlecht um eine Aussage zu treffen. Hier scheint die angewendete Methode zu versagen.

	Signifikanz			
	(1+1) EA		RLS	
Dimension i	$i-1$	$i+1$	$i-1$	$i+1$
1	–	1,000	–	1,000
3	1,000	1,000	1,000	1,000
5	1,000	0,542	1,000	0,000
7	0,000	0,000	0,000	0,000
9	0,000	0,000	0,000	0,000
11	0,000	0,000	0,000	0,000
13	0,000	0,001	0,000	0,000
15	0,000	0,074	0,000	0,000
17	0,000	0,326	0,000	0,000
19	0,000	0,011	0,000	0,000
21	0,000	0,654	0,000	0,002

Tabelle 9.13: Die Ergebnisse der Mann-Whitney-Tests. Für jede ungerade Dimension i ist abzulesen, mit welcher Signifikanz die Erfolgsrate für Dimension i kleiner ist als die für Dimension $i-1$ bzw. $i+1$, und zwar jeweils für den (1+1) EA und die RLS. Die Signifikanzen sind auf drei Stellen gerundet.

Hypothese 16: Die Approximationsgüte steigt mit wachsender Dimension.

Zum Testen dieser Hypothese wurde wieder jede Dimension $i-2$ mit der Dimension i mit Hilfe des Mann-Whitney-Tests verglichen. Die Kurven zur Approximationsgüte sind in Abbildung 9.7 aufgeführt. Die Signifikanzwerte finden sich in Tabelle 9.14.

Die sich hier ergebenden Signifikanzwerte sind erstaunlich gut, allerdings gibt es wieder einige störende Einträge in hohen Dimensionen. Für den (1+1) EA ist das der Übergang zwischen Dimensionen 17 und 19 und zwischen 20 und 22, für die RLS lediglich der Übergang zwischen Dimensionen 18 und 20. Obwohl eine Betrachtung der Kurve nahe legt, dass die Hypothese richtig ist, können wir sie mit unserem Test nicht deutlich bestätigen. Die Gründe dafür dürften ähnlich den in Hypothese 13 aufgeführten Gründen sein. In hohen Dimensionen steigt die Approximationsgüte nur noch sehr schwach an, so dass vermutlich ein Vielfaches der Läufe durchgeführt werden müsste um signifikante Aussagen zu erhalten.

Hypothese 17: Die Approximationsgüte ist für die RLS geringer als für den (1+1) EA.

Die Tests zu dieser Hypothese wurden analog zu den Tests zu Hypothese 14 durchgeführt. Die Signifikanzwerte sind in Tabelle 9.15 enthalten.

Die Hypothese kann für die Dimensionen 11 bis 22 bestätigt werden. Interessant ist allerdings noch, dass die Signifkanzwerte für ungerade Dimensionen hier deutlich besser zu sein scheinen als die für gerade Dimensionen.

	Signifikanz	
Dimension	(1+1) EA	RLS
3 – 5	0,000	0,000
5 – 7	0,000	0,000
7 – 9	0,000	0,000
9 – 11	0,000	0,000
11 – 13	0,000	0,000
13 – 15	0,000	0,000
15 – 17	0,000	0,000
17 – 19	0,094	0,000
19 – 21	0,000	0,002
2 – 4	0,000	0,000
4 – 6	0,000	0,000
6 – 8	0,000	0,000
8 – 10	0,000	0,000
10 – 12	0,000	0,000
12 – 14	0,001	0,001
14 – 16	0,006	0,001
16 – 18	0,002	0,008
18 – 20	0,004	0,058
20 – 22	0,533	0,015

Tabelle 9.14: Die Ergebnisse der Mann-Whitney-Tests. In der Zeile mit den Dimensionen „$x - y$“ wurde jeweils für den (1+1) EA und die RLS getestet, mit welcher Signifikanz die Approximationsgüte für Dimension x kleiner ist als die für Dimension y. Die Signifikanzen sind auf drei Stellen gerundet. Es sind nur Läufe berücksichtigt, die kein Optimum gefunden haben.

Dimension	Signifikanz
1	1,000
2	1,000
3	1,000
4	0,643
5	0,399
6	0,998
7	0,144
8	0,688
9	0,000
10	0,318
11	0,000
12	0,015
13	0,000
14	0,004
15	0,000
16	0,006
17	0,000
18	0,000
19	0,000
20	0,000
21	0,000
22	0,001

Tabelle 9.15: Die Ergebnisse der Mann-Whitney-Tests. In jeder Zeile ist angegeben, mit welcher Signifikanz die RLS in der entsprechenden Dimension eine schlechtere Approximationsrate als der (1+1) EA hat. Die Signifikanzen sind auf drei Stellen gerundet. Es sind nur Läufe berücksichtigt, die kein Optimum gefunden haben.

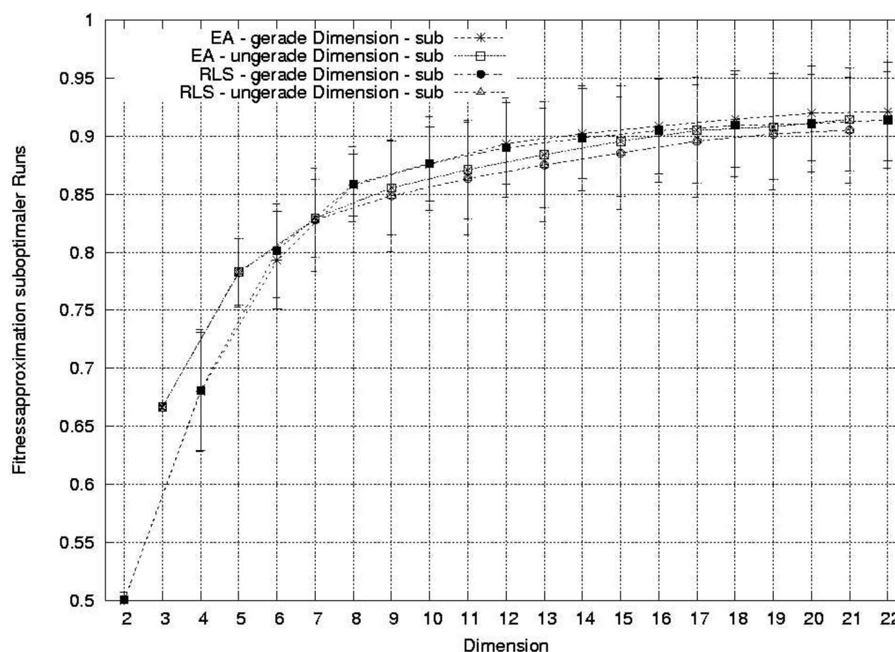


Abbildung 9.7: Durchschnittliches Verhältnis der Fitness suboptimaler stabiler Zustände zur optimalen Fitness.

Hypothese 18: Die Approximationsgüte ist für ungerade Dimensionen i geringer als für die geraden Dimensionen $i-1$ und $i+1$.

Hier können wir mit einigem Erfolg die gleiche Methode wie für Hypothese 15 anwenden. Sehen wir uns die Signifkanzwerte in Tabelle 9.16 an, stellen wir zunächst fest, das wir die Hypothese bis Dimension 7 (im Falle des (1+1) EA sogar bis Dimension 9) nicht nachweisen können. Für die restlichen untersuchten Dimensionen können wir die Hypothese jedoch bestätigen. Ein Blick auf die Kurve in Abbildung 9.7 verrät uns, dass die Hypothese bis Dimension 7 nicht nachweisbar ist, weil sie für diese Dimensionen wahrscheinlich falsch ist. So zeigt sich also auch wieder, dass in kleinen Dimensionen besondere Situationen auftreten können.

Hypothese 19: Zum Erreichen eines stabilen Zustands benötigt die RLS weniger Generationen als der (1+1) EA.

Diese Hypothese kann wie erwartet bestätigt werden. Der Mann-Whitney-Test liefert für jede betrachte Dimension eine Signifikanz von 0,000 (auf drei Stellen gerundet).

	Signifikanz			
	(1+1) EA		RLS	
Dimension i	$i-1$	$i+1$	$i-1$	$i+1$
1	–	1,000	–	1,000
3	1,000	0,000	1,000	0,000
5	1,000	0,016	1,000	0,000
7	1,000	0,000	1,000	0,000
9	0,176	0,000	0,000	0,000
11	0,001	0,000	0,000	0,000
13	0,000	0,000	0,000	0,000
15	0,000	0,000	0,000	0,000
17	0,008	0,000	0,000	0,000
19	0,000	0,000	0,000	0,000
21	0,001	0,000	0,000	0,000

Tabelle 9.16: Die Ergebnisse der Mann-Whitney-Tests. Für jede ungerade Dimension i ist abzulesen, mit welcher Signifikanz die Approximationsgüte für Dimension i kleiner ist als die für Dimension $i-1$ bzw. $i+1$, und zwar jeweils für den (1+1) EA und die RLS. Die Signifikanzen sind auf drei Stellen gerundet.

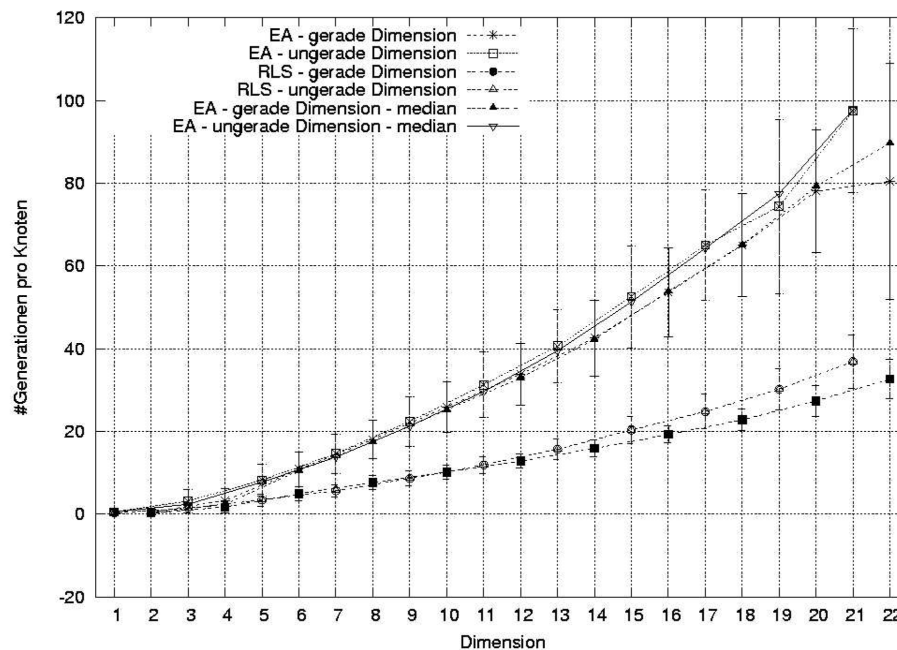


Abbildung 9.8: Anzahl der Generationen pro Knoten bis zum Erreichen eines stabilen Zustands.

9.2 Minimale Spannbäume

9.2.1 Experimente auf einem asymptotischen Worst-Case-Beispiel

Bei der Untersuchung von Ingo's Graph mit verschiedenen Verhältnissen zwischen Dreiecken und Clique ist uns aufgefallen, dass Ingo's Graph immer leichter zu optimieren war als zufällige Graphen gleicher Dimension. Wir wollten nun testen, ob dies auch für Graphen mit mehr oder weniger als 29 Knoten gilt.

Damit sich auch die Ergebnisse für verschiedene Dimensionen untereinander vergleichen lassen, haben wir die Verteilung von Knoten auf Dreiecke und Clique fest gehalten. Dies ermöglichte uns das Wachstum der Laufzeit mit steigender Knotenzahl ohne Seiteneffekte zu analysieren.

Bei einer gleichmäßigen Verteilung existiert eine entsprechende Instanz von Ingo's Graph alle vier Knoten, bei jeder anderen Verteilung sind passende Instanzen seltener, daher haben wir die gleichmäßige Verteilung gewählt.

Unsere Hypothese ist: Die erwartete Laufzeit des (1+1) EA bis zur Berechnung eines minimalen Spannbaums ist auf Ingo's Graph (mit gleichmäßiger Verteilung der Knoten auf Dreiecke und Clique) kleiner als auf zufälligen zusammenhängenden Graphen mit gleicher Knoten- und Kantenzahl und gleichem Maximalgewicht.

Wir haben nun versucht, diese Hypothese für möglichst viele Dimensionen statistisch zu bestätigen. Dazu wurden die 10 Instanzen von Ingo's Graph mit 5 bis 41 Knoten getestet. Für Ingo's Graph und die zufälligen Graphen wurden pro Dimension jeweils 1000 Läufe gestartet, wobei der zufällige Graph für jeden Lauf neu gewählt wurde. Außerdem wurde, wie bei allen folgenden Experimenten, die natürliche Fitnessfunktion w' benutzt.

Knoten	Kanten	Ingo's Graph	Zufällige Graphen
5	6	43,98	70,80
9	16	456,04	1062,92
13	30	1901,28	4659,35
17	48	5444,47	14139,30
21	70	12113,63	31927,86
25	96	24191,28	65674,34
29	126	43132,79	119995,47
33	160	73378,60	197078,25
37	198	116879,16	318341,41
41	240	176590,61	478054,23

Tabelle 9.17: Vergleich der Laufzeiten des (1+1) EA auf Ingo's Graph und zufälligen Graphen bei steigender Dimension.

Es wurden jeweils die Mittelwerte der 1000 Läufe berechnet (Tabelle 9.17) und miteinander mit dem 2-Stichproben-Gaußtest verglichen. Für alle 10 Experimente wird die Hypothese auf einem Signifikanzniveau von 0,001 bestätigt.

Für höhere Dimensionen waren die Experimente leider zu langwierig. Mit entsprechendem Aufwand wäre es natürlich möglich, noch ein Stück weiter zu rechnen. Insbesondere die für jede Generation vollständig neu erfolgende Fitnessauswertung ließe sich durch einen spezialisierten Algorithmus stark beschleunigen. Da aber die Laufzeit des EA mit $\Theta(n^4 \log n)$ steigt, sind die Grenzen des Erreichbaren recht eng gesteckt.

Um aus den vorhandenen Daten einige Anhaltspunkte zu gewinnen, haben wir eine Regressionsanalyse mit einigen typischen Laufzeitfunktionen ohne Terme niedrigerer Ordnungen auf den Daten durchgeführt. Wir betrachten die Hypothese dabei als bestätigt, wenn unter allen getesteten Funktionen $m^2 \log n$ die Daten für zufällige Graphen mit dem geringsten quadratischen Fehler approximiert.

Die verwendeten Funktionen waren m, $m \log m$, m^2, $m^2 \log(\log m)$, $m^2 \log n$, und m^3, jeweils mit einem konstanten Vorfaktor als einzigem freien Parameter für die Regression. Die obere Schranke $m^2 \log n$ lässt sich als $m^2 \log(\sqrt{8m+16} - 3) = \Theta(m^2 \log m)$ beschreiben, daher haben wir auf zusätzliche Funktionen zur Unterscheidung zwischen $\log n$ und $\log m$ verzichtet. Die letzte Funktion wächst in jedem Fall zu schnell und diente nur als Vergleich.

Die Analyse wurde mit dem Marquardt-Levenberg Algorithmus (Gnuplot) durchgeführt, der Initialwert für a wurde in allen Fällen auf 1 gesetzt.

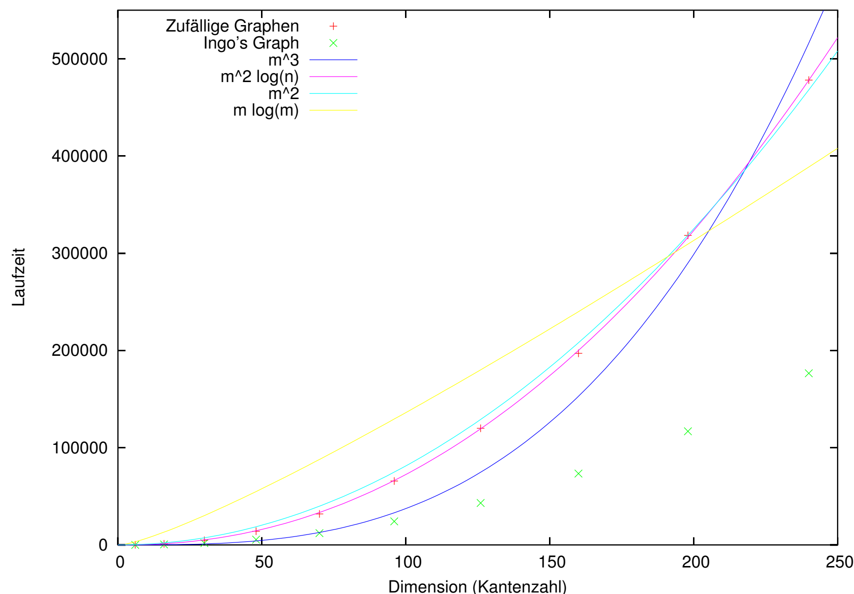


Abbildung 9.9: Regressionsanalyse auf zufälligen Graphen für MST.

Die Funktion $m^2 \log n$ erreichte wie erwartet die geringste quadratische Abweichung, wobei allerdings der Unterschied zu $m^2 \log(\log m)$ sehr gering war. Am Plot der Funktion in Abbildung 9.9 erkennt man leicht, dass die Funktion die Daten tatsächlich gut beschreibt.

Zuletzt wollten wir uns als weiteren Anhaltspunkt für unsere Hypothese einen Vergleich zwischen beiden Datenreihen unternehmen. Wenn die Hypothese korrekt ist, muss der Quotient der tatsächlichen Laufzeitfunktionen durch zwei Konstanten beschränkt sein. Vermutlich

wird der Quotient sogar gegen eine Konstante konvergieren, da die Probleme keine erkennbare Struktur aufweisen, die sich periodisch mit steigender Dimension ändert.

Diese Aussage lässt sich schlecht durch systematische Analysen prüfen, daher beschränken wir uns auf ein Diagramm (Abbildung 9.10). Sollten zufällige Graphen asymptotisch leichter zu optimieren sein als Ingo's Graph, müsste der Quotient mit der Dimension steigen. Im betrachteten Bereich verhält sich der Quotient jedoch offensichtlich nicht so.

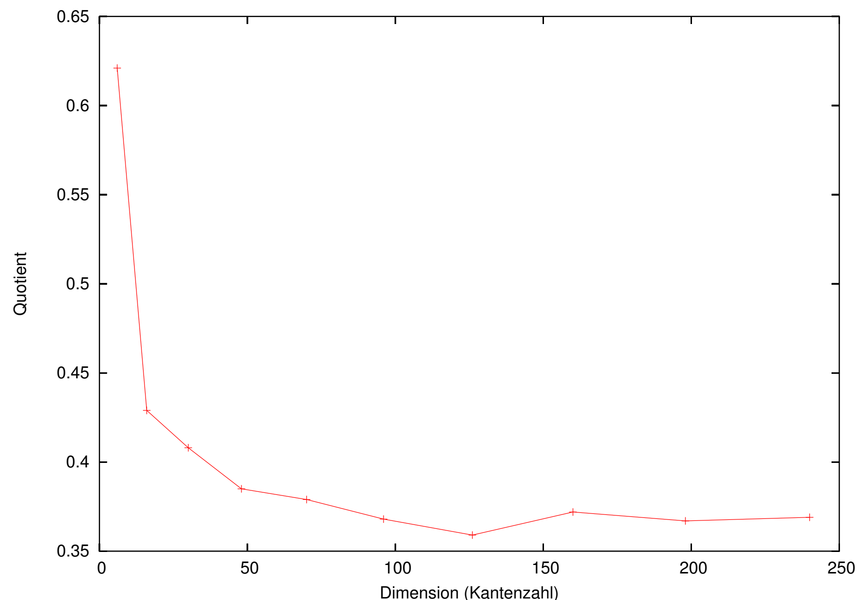


Abbildung 9.10: Quotient der Laufzeiten des (1+1) EA auf Ingo's Graph und zufälligen Graphen.

9.2.2 Experimente zu Kantengewichten

Da die Binary-Value-Gewichte aus den ersten Experimenten nicht ohne weiteres zu untersuchen waren (siehe Abschnitt 8.2.2), stellten wir eine andere, aber doch ähnliche Vermutung auf: Mit der Anzahl der möglichen Kantengewichte erhöht sich die Schwierigkeit für den (1+1) EA, einen minimalen Spannbaum zu finden. Die Wahrscheinlichkeit, dass es sehr viele minimale Spannbäume mit gleicher Fitness gibt, ist bei einer kleinen Auswahl von Kantengewichten deutlich höher als bei sehr großen. Da lediglich ein einziger minimaler Spannbaum vom (1+1) EA gefunden werden muss, sollte das Problem einfacher sein, wenn viele Lösungen möglich sind. Insofern haben wir unser eigentliches Vorgehen, den Unterschied zwischen exponentiellen und polynomiellen Gewichten nachweisen zu wollen, dahingehend geändert, nun die Schwierigkeit vieler verschiedener Kantengewichte gegenüber wenigen zu bestätigen.

Untersucht wurden Graphen mit Knotenzahlen von 5 bis 40. Die Größe der untersuchten Graphen wurde durch die vorhandene Rechenzeitkapazität beschränkt. Die repräsentativste Auswahl von zufälligen Graphen scheint uns unter den gegebenen Bedingungen die Folgende

zu sein: Bei gegebener Knotenmenge wird für jede potentielle Kante mit Wahrscheinlichkeit 1/2 ausgewürfelt, ob die Kante vorhanden ist oder nicht. Für jede gewählte Kante wird uniform zufällig aus der Menge der möglichen Kantengewichte eines ausgewählt. Für die Beschränkung der möglichen Kantengewichte haben wir folgende Mengen gewählt:

1. $\{1\}$
2. $\{1, 2\}$
3. $\{1, \ldots, \log \mathrm{E}(m)\}$
4. $\{1, \ldots, \sqrt{\mathrm{E}(m)}\}$
5. $\{1, \ldots, \mathrm{E}(m)\}$
6. $\{1, \ldots, (\mathrm{E}(m))^2\}$

Dabei ist $\mathrm{E}(m)$ die erwartete Anzahl von Kanten im Graphen. Also ist unter den gegeben Umständen

$$\mathrm{E}(m) = \frac{1}{2} \cdot \binom{\text{Knotenzahl}}{2}.$$

Im ersten Fall ist jeder Spannbaum optimal, im zweiten Fall gibt es zumindest noch sehr viele optimale Spannbäume. Wenn die Kantengewichte aus dem Bereich 1 bis $(\mathrm{E}(m))^2$ stammen, ist die Wahrscheinlichkeit, dass es keine zwei Kanten mit dem selben Gewicht gibt, bereits größer als $1/e$ und wir sehen keine Veranlassung zu vermuten, dass noch größere Mengen von potenziellen Kantengewichten eine weitere großartige Verschlechterung der Performance nach sich ziehen. Wenn alle Kantengewichte unterschiedlich sind, so ist der minimale Spannbaum eindeutig.

Die restlichen Intervallgrenzen der Kantenmengen scheinen uns repräsentativ zu sein, hätten aber auch anders gewählt werden können.

Für jede Knotenzahl mit jeder Kantengewichtsmenge haben wir jeweils 1000 Läufe auf jeweils neu ausgewürfelten Graphen analysiert. Gemessen wurde die Anzahl der Fitnessauswertungen, bis das Optimum erreicht wurde. Untersucht wurde dabei der (1+1) EA mit der natürlichen Fitnessfunktion w', die im Gegensatz zur Fitnessfunktion w keine weiteren Hinweise gibt, einen gültigen Spannbaum zu erreichen (siehe Abschnitt 7.1).

Auswertung

Für die statistische Auswertung wurden zusätzlich zur Berechung der Mittelwerte der jeweils 1000 Läufe unsere Hypothese „Je größer die Auswahl an Kantengewichten, desto länger benötigt der (1+1) EA zur Optimierung“ getestet. Dazu wurde die obige Hypothese aufgespalten und für eine feste Knotenzahl und jede mögliche Kombination der Kantengewichte i mit j $(1 \leq i < j \leq 6)$ folgende Hypothese überprüft:

Hypothese 27. *Die Verteilung der Laufzeiten bei Graphen mit n Knoten sind für die beiden Kantengewichte i und j verschieden.*

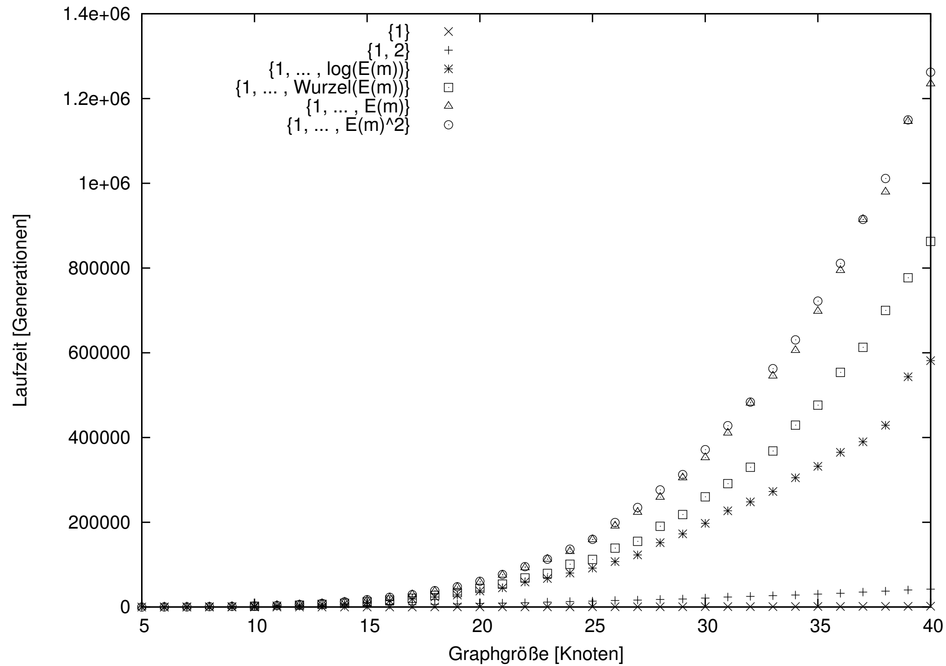


Abbildung 9.11: Die Mittelwerte der Laufzeiten des (1+1) EA über die Graphgröße aufgetragen.

Diese Hypothese wurde mit dem nichtparametrischen Mann-Whitney-Test durchgeführt. Dieser Test ist ein so genannter Rangsummentest, der die jeweils 1000 Messwerte von zwei verschiedenen Stichproben vergleicht und ihnen Ränge zuordnet. Abhängig von den Rängen entscheidet der Test, mit welcher Wahrscheinlichkeit die beiden Stichproben derselben Verteilung unterliegen. Das Ergebnis unserer Tests sieht folgendermaßen aus: Fast alle Tests waren statistisch hoch signifikant, mit Ausnahme der Vergleiche zwischen den Kantengewichten $\mathrm{E}(m)$ (Stichprobe 5) und $(\mathrm{E}(m))^2$ (Stichprobe 6). Die statistische Signifikanz dieses Tests zeigt Tabelle 9.18 in Abhängigkeit der Knotenzahl. Bei kleinen Knotengrößen liegen außerdem $\log \mathrm{E}(m)$ und $\sqrt{\mathrm{E}(m)}$ zu nahe zusammen, als dass die Ergebnisse statistisch sehr signifikant sein könnten (siehe Tabelle 9.19). Bei weniger als 12 Knoten waren diese Werte gerundet meistens sogar gleich.

Die Verteilungen sind nicht nur nach den statistischen Auswertungen verschieden, sondern die Mittelwerte waren bei den Experimenten auf einer Graphgröße, die aus einer größeren Menge von Kantengewichten wählen durften, tatsächlich auch immer größer. In dieser Hinsicht konnten die Hypothesen bis auf den Vergleich von Experiment 5 mit Experiment 6 ab einer Knotenzahl von 12 voll bestätigt werden.

Die Abbildungen 9.11 und 9.12 zeigen zudem die Entwicklung der Mittelwerte der Laufzeiten über die Graphgröße.

Knotenzahl	Stat. Signifikanz	
40	0,214	-
39	0,538	-
38	0,110	-
37	0,393	-
36	0,517	-
35	0,068	-
34	0,081	-
33	0,027	*
32	0,959	-
31	0,010	**
30	0,003	**
29	0,151	-
28	0,002	**
27	0,007	**
26	0,053	-
25	0,757	-
24	0,016	*
23	0,435	-
22	0,181	-
21	0,459	-
20	0,183	-
19	0,197	-
18	0,010	**
17	0,439	-
16	0,065	-
15	0,001	***
14	0,021	*
13	0,046	*
12	0,010	**
11	0,029	*
10	0,000	***
9	0,005	**
8	0,001	***
7	0,394	-
6	0,010	**
5	0,000	***

Tabelle 9.18: Die statistische Signifikanz der Vergleichstests zwischen Kantengewichten aus der Menge $\{1, \ldots, \mathrm{E}(m)\}$ und der Menge $\{1, \ldots, (\mathrm{E}(m))^2\}$ in Abhängigkeit der Knotenzahl.

Knotenzahl	Stat. Signifikanz	
15	0,003	**
13	0,019	*
12	0,003	**
10	0,454	-

Tabelle 9.19: Die statistische Signifikanz der Vergleichstests zwischen Kantengewichten aus der Menge $\{1, \ldots, \log \mathrm{E}(m)\}$ und der Menge $\{1, \ldots, \sqrt{\mathrm{E}(m)}\}$ in Abhängigkeit der Knotenzahl.

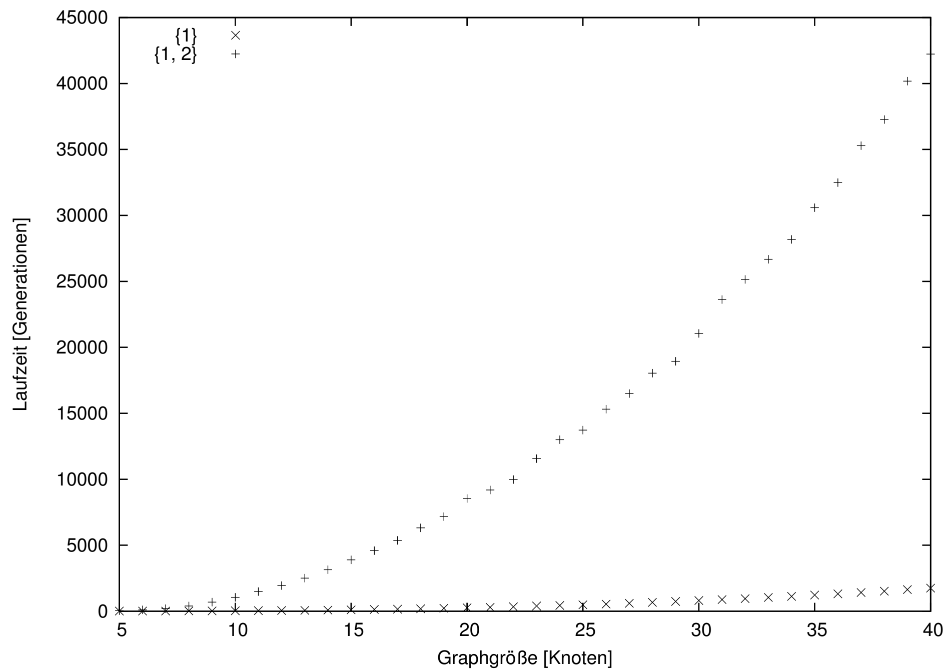


Abbildung 9.12: Die Mittelwerte der Laufzeiten des (1+1) EA über die Graphgröße aufgetragen. Hier nur die Kantengewichte aus den Mengen $\{1\}$ und $\{1, 2\}$.

Regression

Zusätzlich haben wir auch eine Regression über die Kurvenverläufe der Mittelwerte durchgeführt. Dabei haben wir uns aber nach ersten Testregressionen mit komplizierten Funktionenklassen mit mehreren Variablen entschlossen, als Funktionenklassen nur die folgenden zu betrachten:

1. $f(x) = a \cdot x$
2. $g(x) = b \cdot x^2$
3. $h(x) = c \cdot x^3$
4. $i(x) = d \cdot x^4$
5. $j(x) = e \cdot x^4 \cdot \log(x)$

Die Ergebnisse der Regression stehen in Tabelle 9.20.

Fazit zu Kantengewichten in minimalen Spannbäumen

Je kleiner die Menge ist, aus der die Kantengewichte gewählt werden können, desto besser ist die Performance des (1+1) EA.

9.2.3 Experimente zum Vergleich der Algorithmen (1+1) EA und RLS

Nach den Vorexperimenten haben wir folgende Hypothesen aufgestellt:

Hypothese 28. *Die RLS mit 2-Bit-Flip-Wahrscheinlichkeit* $1/2$ *ist etwa um den Faktor* e *schneller als der (1+1) EA.*

Hypothese 29. *Bei hoher 2-Bit-Flip-Wahrscheinlichkeit ist die RLS fast um den Faktor* $2e$ *schneller als der (1+1) EA.*

Als repräsentative Graphen haben wir bei gegebener Knotenzahl einen Graphen gewählt, bei dem für jede mögliche Kante mit Wahrscheinlichkeit 1/2 entschieden wird, ob die Kante zum Graphen gehört oder nicht. Jeder Kante wird ein Gewicht zugeteilt, uniform zufällig aus dem Intervall [1,Knotenzahl] gewählt. Untersucht wurden Graphen mit Knotenzahlen 5 bis 36.

Folgende Algorithmen wurden untersucht, wobei n die Knotenzahl und m die Kantenzahl bezeichnet:

- (1+1) EA
- (1+10) EA
- (1+100) EA

	$\{1\}$	$\{1,2\}$	$\{1,\ldots,\log m\}$
SQ(f)f	2,8157e+06	1,35472e+09	3,8757 e+11
a	28,3341	734,464	7531,7
SQ(g)	359144,0	7,77761e+07	1,21367e+11
b	0,963188	24,6214	264,087
SQ(h)	30731,0	1,22115e+08	2,23412e+10
c	0,0283752	0,718305	7,96493
SQ(i)	543763,0	6,29436e+08	4,67561e+09
d	0,000784743	0,0197267	0,224297
SQ(j)	764539,0	8,10862e+08	8,167 e+09
e	0,000217294	0,00545313	0,0623859

	$\{1,\ldots,m\}$	$\{1,\ldots,\sqrt{m}\}$	$\{1,\ldots,m^2\}$
SQ(f)	2,03312e+12	9,11672e+11	2,08784e+12
a	15385,4	10909,2	15749,9
SQ(g)	7,55266e+11	3,421 e+11	7,6161e+11
b	546,768	380,92	559,193
SQ(h)	1,97336e+11	8,53513e+10	1,90163e+11
c	16,6487	11,5684	17,0131
SQ(i)	1,36028e+10	4,03883e+09	9,11519e+09
d	0,472319	0,327828	0,482312
SQ(j)	2,7527 e+09	4,90025e+08	6,05109e+08
e	0,13161	0,0913257	0,134369

Tabelle 9.20: Die Summe der quadratischen Abweichungen (SQ) und die Koeffizienten der Regression zu den Experimenten bei unterschiedlichen Kantengewichten.

- $(1+n^2)$ EA
- $(1+n^3)$ EA
- $(1+\frac{n^2}{\log(n)})$ EA
- RLS mit 2-Bit-Flip-Wahrscheinlichkeit 50%
- RLS mit 2-Bit-Flip-Wahrscheinlichkeit 90%
- RLS mit 2-Bit-Flip-Wahrscheinlichkeit 99%
- RLS mit 2-Bit-Flip-Wahrscheinlichkeit $1 - 1/m \cdot \log(n)$

Hypothese 30. *Die genannten Algorithmen unterscheiden sich bei fester Graphgröße paarweise voneinander bezüglich der Anzahl der Fitnessauswertungen, bis ein Optimum gefunden wird.*

Jeweils 1000 Läufe wurden auf jeweils neu ausgewürfelten Graphen untersucht. Gemessen wurde die Anzahl der Fitnessauswertungen bis ein Optimum erreicht wurde.

Der letzte Algorithmus schien uns interessant, da wir von einer erwarteten Optimierungszeit von $O(m^2 \log n)$ ausgehen (siehe auch Abschnitt 7.1). In dieser Zeit kann es einen nötigen 1-Bit-Flip geben, der an genau einer von m verschiedenen Stellen auftreten muss. Damit wir diesen einen 1-Bit-Flip in unserer erwarteten Optimierungszeit auch mit hoher Wahrscheinlichkeit durchführen, müssen wir mit Wahrscheinlichkeit $1/(m \cdot \log(n))$ einen 1-Bit-Flip durchführen.

Auswertung

Wie man an den Mittelwerten der Laufzeiten der einzelnen Algorithmen (Abbildungen 9.13 und 9.14) zum Teil deutlich sehen kann, gibt es Unterschiede in der Performanz der Algorithmen, gemessen in der Anzahl der Fitnessauswertungen. Wir haben statistische Tests auf jeder untersuchten Graphgröße durchgeführt und dabei jeden der zehn Algorithmen gegen die anderen getestet. Verwendet wurde der Mann-Whitney-Test, der lediglich die Ränge der Stichproben betrachtet. Es ergaben sich folgende Ergebnisse: Der (1+1) EA ist schneller als der $(1+\lambda)$ EA. Allerdings waren für $\lambda = 10$ die Unterschiede kaum messbar und auch nicht signifikant. Bis zu einer Knotenzahl von 20 Knoten ist der (1+1) EA bei allen Knotenzahlen signifikant schneller als der (1+100) EA, meistens sogar sehr signifikant schneller ($p \leq 0{,}001$). Danach sind die Stichproben mit einer Größe von 1000 offensichtlich nicht mehr groß genug, um die Unterschiede als statistisch signifikant einstufen zu können. Der (1+1) EA ist auch auf fast allen untersuchten Graphgrößen schneller als der $(1+\frac{n^2}{\log(n)})$ EA, allerdings sind die Unterschiede nicht signifikant. In Abbildung 9.14 liegen die Kurven der $(1+\lambda)$ EAs sehr nahe beisammen für $1 \leq \lambda \leq n^2$, und vermutliche Unterschiede sind bei der gewählten Experimentgröße ebenfalls nicht signifikant.

Der $(1+n^2)$ EA unterscheidet sich von allen anderen Algorithmen sehr signifikant, außer vom (1+100) EA, bei dem der Test bei manchen Knotenzahlen keine statistische Signifikanz

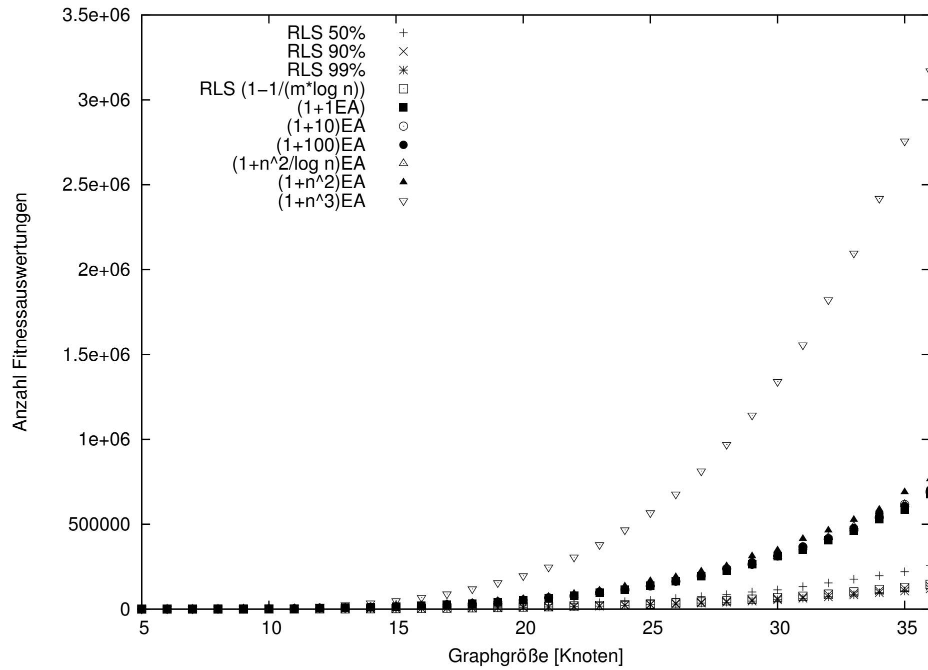


Abbildung 9.13: Die Mittelwerte der Laufzeiten der untersuchten Algorithmen über die Graphgröße aufgetragen.

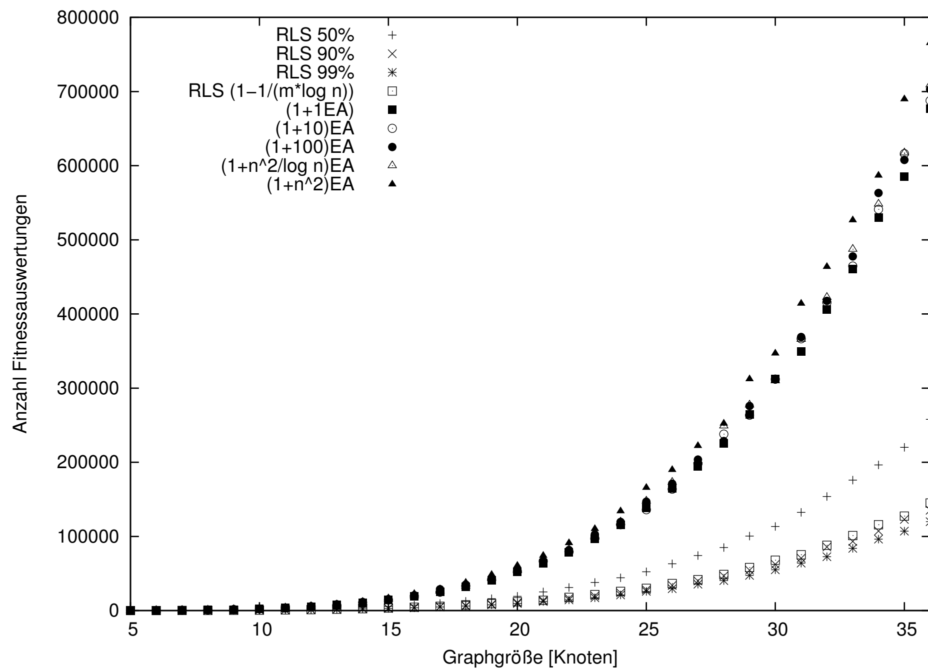


Abbildung 9.14: Die Mittelwerte der Laufzeiten der untersuchten Algorithmen (mit Ausnahme des $(1+n^3)$ EA) über die Graphgröße aufgetragen.

Knotenzahl	Signifikanz		Signifikanz	
5	0,845			
6	0,001	***		
8	0,003	**		
13	0,045	*	0,799	
14	0,039	*	0,394	
15	0,011	*	0,144	
16	0,061		0,016	*
17	0,337		0,001	***
18			0,004	**
19	0,187			
20	0,212			
21	0,031	*		

Tabelle 9.21: Links der Vergleich der RLS mit 90% 2-Bit-Flip-Wahrscheinlichkeit mit der RLS mit $(1 - \frac{1}{m \cdot \log(n)})$ 2-Bit-Flip-Wahrscheinlichkeit. Bei höheren Graphgrößen sind die Unterschiede nicht mehr signifikant. Rechts der Vergleich der RLS mit 90% 2-Bit-Flip-Wahrscheinlichkeit mit der RLS mit 99% 2-Bit-Flip-Wahrscheinlichkeit. Alle nicht aufgeführten Signifikanzen waren kleiner als 0,001.

feststellen konnte. Eine Ausnahme bei 28 Knoten gegenüber dem $(1+\frac{n^2}{\log(n)})$ EA werten wir als zufällige Schwankung.

Der $(1+n^3)$ EA und die RLS mit 50% 2-Bit-Flip-Wahrscheinlichkeit unterscheiden sich bei allen untersuchten Graphen signifikant von allen anderen Algorithmen, bei kleinen Graphgrößen (Knotenzahl ≤ 10) sogar sehr signifikant. Auch die RLS mit 90% und mit 99% 2-Bit-Flip-Wahrscheinlichkeit unterschieden sich fast immer sehr signifikant von den anderen Algorithmen. Die Ausnahmen sind in der Tabelle 9.21 zu sehen. Wir vermuten, dass die Ergebnisse für größere Stichproben oder größere Graphen noch deutlicher sind. Allerdings ließen sich diese Experimente bei den bestehenden Rechnerkapazitäten nicht in einem größeren Maßstab durchführen.

Wir vermuten, dass die RLS mit $(1 - \frac{1}{m \cdot \log(n)})$ 2-Bit-Flip-Wahrscheinlichkeit schlechter abgeschnitten hat als zuvor vermutet, weil bei den betrachteten Graphgrößen $1 - \frac{1}{m \cdot \log(n)}$ zu groß ist. Die 1-Bit-Flip-Wahrscheinlichkeit ist in diesem Falle zu klein und wir müssen in der Anfangsphase des Optimierungsprozesses zu lange auf den eventuell nötigen Flip warten.

Die Mehrbitflips des (1+1) EA bieten beim Problem der minimalen Spannbaumberechnung anscheinend keinen Vorteil. Das Problem MST lässt sich lokal optimieren und die RLS, die „lokaler“ als der (1+1) EA operiert, ist im Vorteil, da sie Schritte mit Mehrbitflips, die anscheinend selten akzeptiert werden, vermeidet. Auch wenn die Unterschiede bei dem „kleinen“ Experimentaufbau (mit immerhin mehreren 100000 Experimenten) nicht immer statistisch signifikant sind, so scheinen große Populationen doch ein Nachteil zu sein in Bezug auf die Anzahl der Fitnessauswertungen bis zur Optimierung.

Eine Regressionsanalyse führte mit der Funktion $f(n) = a \cdot n^4$ zu den besten Ergebnissen. In der Tabelle 9.22 sind die Parameter für die einzelnen Algorithmen aufgeführt.

Algorithmus	Koeffizient	Summe der Fehlerquadrate
(1+1) EA	0,38814	2,66793e+09
(1+10) EA	0,397386	3,5335e+09
(1+100) EA	0,403801	3,32219e+09
(1+$\frac{n^2}{\log(n)}$) EA	0,406219	2,95285e+09
(1+n^2) EA	0,444049	3,19669e+09
(1+n^3) EA	1,76059	2,19876e+11
RLS 50%	146531	3,80647e+08
RLS 90%	0,0792883	9,97879e+07
RLS 99%	0,0700224	6,90116e+07
RLS (1-$\frac{1}{m \cdot \log(n)}$)	0,0843154	9,55293e+07

Tabelle 9.22: Das Ergebnis der Regressionsanalyse. Die Werte bei der RLS stehen für die jeweilige 2-Bit-Flip-Wahrscheinlichkeit.

Untersuchung des Faktors zwischen RLS und (1+1) EA

Die Untersuchung des Faktors, um den sich der (1+1) EA und die RLS in der Laufzeit unterscheiden, ergab folgendes Ergebnis: Alle Quotienten der (1+1) EA-Laufzeiten und der RLS-Laufzeiten zeigt Abbildung 9.15. Statistisch untersucht wurden aber nur die beiden RLS mit 50% bzw. 99% 2-Bit-Flip-Wahrscheinlichkeit (Abbildungen 9.16 und 9.17).

Der Faktor, um den die RLS mit 50% 2-Bit-Flip-Wahrscheinlichkeit schneller ist als der (1+1) EA, liegt zwischen 2,4 und 2,8. Diese Aussage ist ab einer Graphgröße von 15 Knoten statistisch signifikant. Eine Ausnahme hinsichtlich der Signifikanz bildet der Faktor bei einer Graphgröße von 30 Knoten.

Der Faktor, um den die RLS mit 99% 2-Bit-Flip-Wahrscheinlichkeit schneller ist als der (1+1) EA, liegt zwischen 5,2 und 5,9. Diese Aussage ist erst ab einer Graphgröße von 22 Knoten statistisch signifikant. Eine Ausnahme hinsichtlich der Signifikanz bildet hier der Faktor bei einer Graphgröße von 36 Knoten.

Fazit zum Vergleich der Algorithmen auf minimalen Spannbäumen

Die untersuchten Algorithmen unterscheiden sich zum Teil erheblich. Zur Berechnung minimaler Spannbäume sind Algorithmen im Vorteil, die lokal operieren, wie die RLS. Je wahrscheinlicher 2-Bit-Flips sind, desto besser ist die Performanz. Große Populationen bieten keinen Vorteil. Der Geschwindigkeitsvorteil der RLS mit 50% 2-Bit-Flip-Wahrscheinlichkeit gegenüber dem (1+1) EA liegt etwa bei dem Faktor e. Bei der RLS mit 99% 2-Bit-Flip-Wahrscheinlichkeit ist der Faktor fast $2e$.

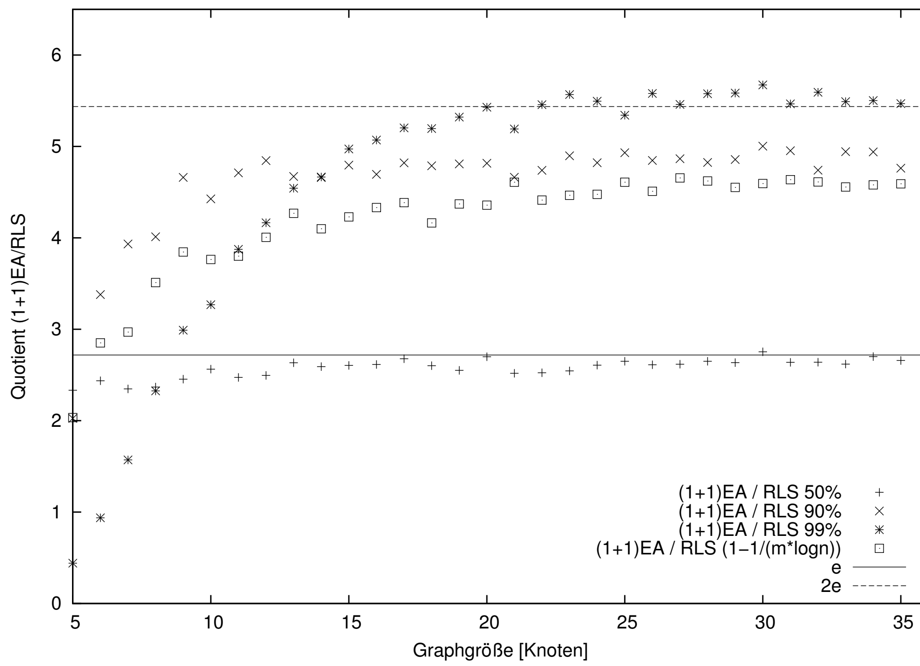


Abbildung 9.15: Die Quotienten aus den Mittelwerten der Laufzeiten des (1+1) EA und der verschiedenen RLS.

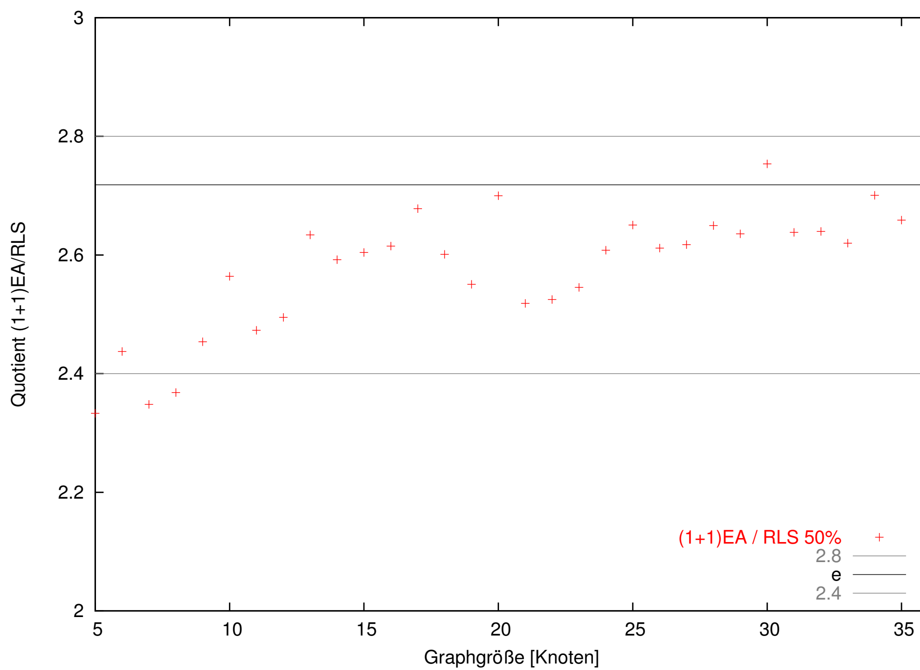


Abbildung 9.16: Die Quotienten aus den Mittelwerten der Laufzeiten des (1+1) EA und der RLS mit 50% 2-Bit-Flip-Wahrscheinlichkeit.

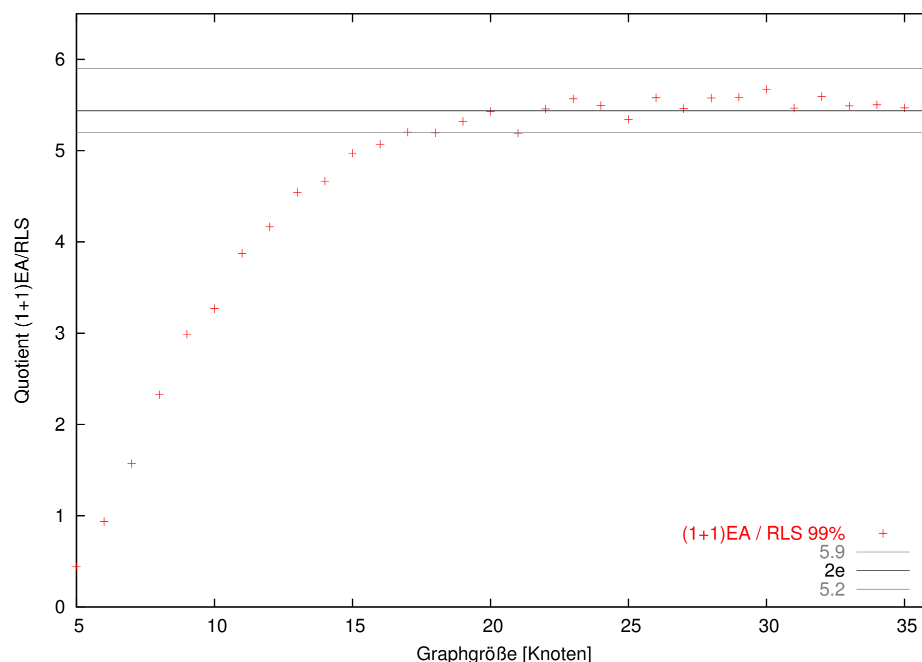


Abbildung 9.17: Die Quotienten aus den Mittelwerten der Laufzeiten des (1+1) EA und der RLS mit 99% 2-Bit-Flip-Wahrscheinlichkeit.

9.2.4 Experimente zur Darstellung von Spannbäumen als Prüfernummern

Hypothese 20: Prüfernummer vs. Bitstring auf zufälligen vollständigen Graphen

Wie in Kapitel 8 beschrieben, wollten wir zunächst testen, wie sich die Darstellung von Spannbäumen vollständiger Graphen als Prüfernummer im Vergleich zu einer Darstellung als Bitstring schlägt. Die Versuchsparameter waren dabei wie folgt definiert:

Graph-Typ:	vollständig
Anzahl Knoten:	10
Anzahl versch. Kantengewichte:	2 bis 50 in Zweier-Schritten
Graphen:	10000

Jede Konfiguration wurde hierbei an 10000 verschiedenen Graphen getestet. Gestartet wurde bei der Bitstring-Darstellung mit einer zufälligen Kantenbelegung und mit einer zufälligen (gültigen) Prüfernummer bei der Prüfernummer-Darstellung. Wie erwähnt wurde dabei zuerst aus den jeweils 10000 Läufen in Bitstring-Darstellung ein Mittelwert über die Laufzeit gebildet, dieser verzehnfacht und als obere Schranke für die maximale Laufzeit des gleichen Experiments für die Prüfernummer-Darstellung gesetzt. Ein Lauf mit Prüfernummer-Darstellung wurde somit beendet, sobald er entweder das Optimum fand oder aber er die soeben festgelegte Generationenanzahl überschritt. Im letzten Fall galt der Lauf als nicht erfolgreich.

Das Verhältnis der Anzahl erfolgreicher Läufe zu der Anzahl von Läufen insgesamt (10000) ergab letztendlich die Erfolgsrate. Ein Beispiel:

- Die maximale Anzahl Kantengewichte sei auf 6 festgelegt.
- Der (1+1) EA auf Bitstrings ermittelt die durchschnittliche Laufzeit auf 10000 zufälligen vollständigen Graphen. Diese sei $G_{avg} = 6919{,}8$.
- Dieser Wert wird mit 10 multipliziert und liefert so die maximale Anzahl von Generationen, die ein Prüfernummer-Lauf laufen darf. $G_{\max} = 69198$.
- Nun werden 10000 Läufe des (1+1) EAs auf Prüfernummern durchgeführt, wobei jeder Lauf gestoppt wird, wenn entweder das Optimum gefunden wurde oder die Anzahl Generationen $G_{\max}$ übersteigt.
- Von diesen 10000 Läufen waren 3967 nicht erfolgreich, d. h. nicht innerhalb von $G_{\max}$ Generationen zum Optimum gelangt.
- Die Erfolgswahrscheinlichkeit liegt somit bei 0,6033.

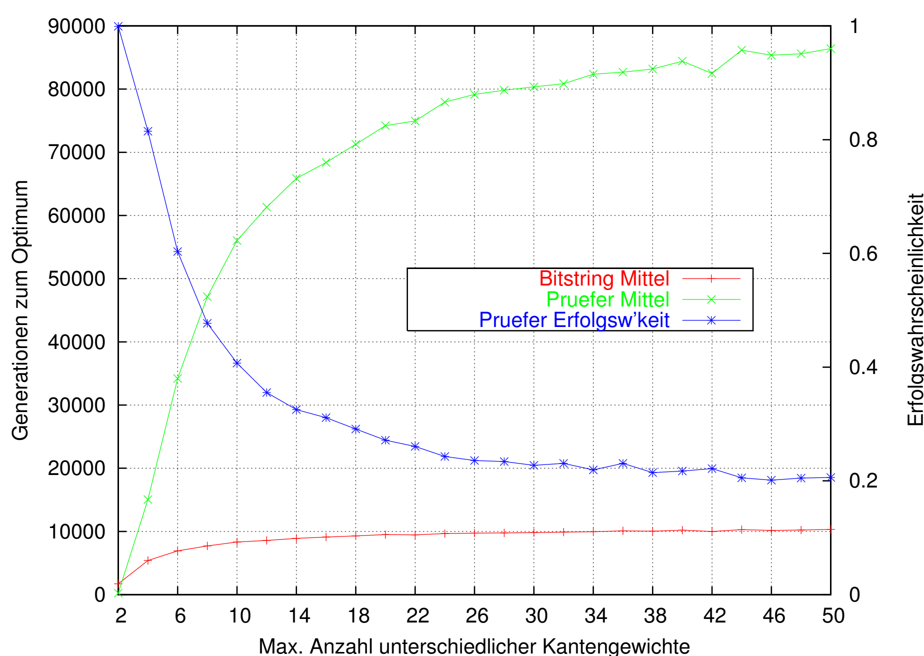


Abbildung 9.18: Durchschnittliche Laufzeiten und Erfolgswahrscheinlichkeiten des (1+1) EA mit Bitstring- und Prüfernummer-Darstellung auf dem MST-Problem.

Es sei bei den Abbildungen 9.18 und 9.19 angemerkt, dass die Kurve für die durchschnittliche Laufzeit und den Median bei Prüfernummer-Darstellung im Normalfall deutlich schneller und steiler ansteigen, da bei unserer Experimentdurchführung ein Lauf nach einer maximalen

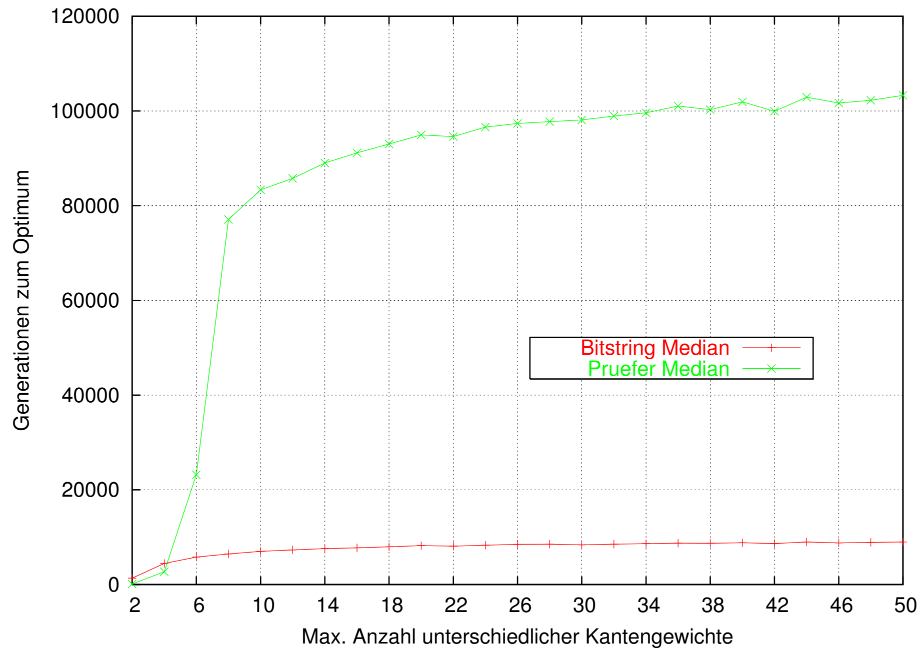


Abbildung 9.19: Median-Werte der Laufzeiten des (1+1) EA mit Bitstring- und Prüfernummer-Darstellung auf dem MST-Problem.

Anzahl Generationen gestoppt wurde und somit für die Anzahl der Generationen bis zum Optimum höchstens das Zehnfache des Durchschnitts der Bitstring-Variante ermittelt wurde.

Man erkennt deutlich, dass lediglich bei zwei möglichen Kantengewichten die Prüfernummer-Darstellung mit der Bitstring-Darstellung mithalten kann. Tatsächlich erweist sich die Prüfernummer-Darstellung bei nur zwei Kantengewichten sogar als schneller als die Bitstring-Darstellung. Bei einem Blick auf die Rohdaten erkennt man eine durchschnittliche Laufzeit von ungefähr 1714 Generationen bei der Bitstring-Darstellung im Vergleich zu lediglich 237 Generationen im Durchschnitt bei der Prüfernummer-Darstellung. Auch bei vier Kantengewichten gibt es noch sehr viele Läufe, die schneller sind als die Bitstring-Darstellung; allerdings häufen sich dort bereits die Ausreißer nach oben. Eine niedrigere Erfolgswahrscheinlichkeit und ein höherer Durchschnitt sind die Anzeichen dafür. Dieses Phänomen setzt sich entsprechend fort. Es gibt mehr und mehr lange Läufe und erfolgreiche Läufe werden immer unwahrscheinlicher.

Schaut man sich die Arbeitsweise des (1+1) EA auf Prüfernummern an, ist dieses Resultat keineswegs verwunderlich. Die Laufzeit des (1+1) EA auf Prüfernummern schwankt nämlich sehr stark. Das liegt daran, dass der Algorithmus teilweise bei recht guten Spannbäumen für lange Zeit verweilt. Aus abstrakter Sicht müssen nur noch wenige Kanten bis zum Optimum flippen, jedoch entspricht das in einer Prüfernummer-Darstellung oft einem Flippen mehrerer Stellen auf den korrekten Wert. So werden aus Mutationen, die in Bitstring-Darstellung einfache 2-Bit-Flips sind, Flips, die 4, 5 oder mehr Stellen auf den korrekten Wert zwischen 1 und n setzen müssen.

Interessant ist außerdem zu sehen, dass die Kurve der Erfolgswahrscheinlichkeiten mit mehreren möglichen Kantengewichten stark abflacht. Daraus könnten Schlussfolgerungen gezogen werden, dass auch bei vielen unterschiedlichen Kantengewichten nicht wesentlich weniger als 20% der Läufe das Optimum finden. Allerdings sind dafür unsere gesammelten Daten nicht ausreichend.

Wie die Kurve der Erfolgswahrscheinlichkeiten flacht auch die Kurve der durchschnittlichen Laufzeiten für die Bitstring-Darstellung ab, woraus man ähnliche Schlussfolgerungen ziehen könnte.

Um unsere Vermutungen statistisch abzusichern, wurde bei diesem Experiment mit SPSS ein Mann-Whitney-Test durchgeführt, der alle gesammelten Daten nach maximaler Anzahl von Kantengewichten gruppiert und auf signifikante Unterschiede hin untersuchte. Dieser Test ergab eine durchgehend hohe Signifikanz von gerundet 0,000 bei allen untersuchten Gruppen.

Hypothese 21: Prüfernummer vs. Bitstring auf vollständigen Graphen mit sternförmigem MST

Nun sollte untersucht werden, ob die Bitstring-Darstellung auf einem vollständigen Graphen, der einen einzigen, sternförmigen minimalen Spannbaum besitzt, eventuell bessere Laufzeiten erreichen kann. Die sternförmigen Graphen wurden so implementiert, dass in einem vollständigen Graphen ein Knoten ausgewählt wird und dessen inzidente Kanten das Gewicht 1 bekommen. Alle anderen Kanten im Graphen bekommen ein Gewicht größer als 1. Solch ein Graph hat genau einen minimalen Spannbaum mit einem zentralen Knoten vom Grad $n-1$.

Der Versuchsaufbau war der folgende:

Graph-Typ:	vollst. Graph mit Stern-MST
Anzahl Knoten:	10 bis 15
Graphen:	1000

Wie aus der Tabelle ersichtlich, wurden Knotenzahlen von 10 bis 15 getestet und obgleich dieser geringen Menge an unterschiedlichen Knotenzahlen lässt sich ein deutliches asymptotisches Verhalten erkennen. Es wurden jeweils 1000 Graphen, die die oben genannte Stern-Eigenschaft besitzen, ausgewürfelt und die Laufzeit in Generationen bis zum Optimum gemessen.

Betrachten wir zunächst einmal die Median-Werte der Laufzeiten etwas näher.

In Abbildung 9.20 könnte man zu dem erfreulichen Schluss kommen, dass die Prüfernummer-Darstellung auf sternförmigen minimalen Spannbäumen deutlich bessere Ergebnisse erzielt als die Bitstring-Darstellung. Doch der Schein trügt. So gut die Prüfernummer-Darstellung auch in den meisten Fällen arbeitet, so sind es doch die Ausnahmen, die die durchschnittliche Laufzeit enorm in die Höhe treiben. In Abbildung 9.21 erkennt man beispielsweise, dass die Standardabweichung in der Bitstring-Darstellung verhältnismäßig moderat ausfällt, wohingegen Abbildung 9.22 die Probleme der Prüfernummer-Darstellung deutlich macht.

In manchen Läufen wartet der Algorithmus, wie auch schon im Experiment zuvor, auf den entscheidenden Flip, um einen besseren Spannbaum zu erreichen. Da es nur einen einzigen,

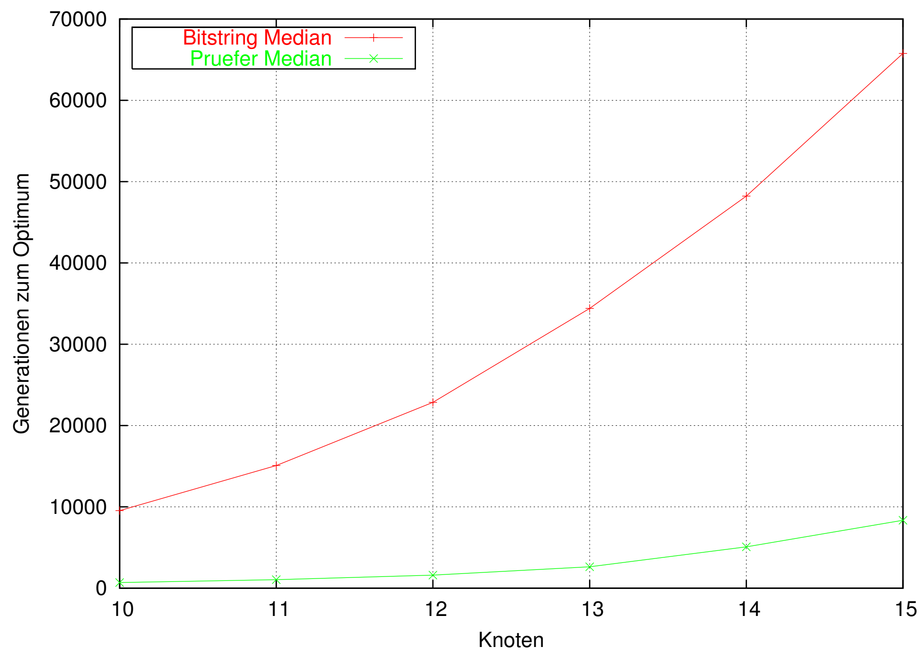


Abbildung 9.20: Median-Werte der Laufzeiten des (1+1) EA mit Bitstring- und Prüfernummer-Darstellung auf dem MST-Problem auf vollständigen Graphen mit sternförmigem MST.

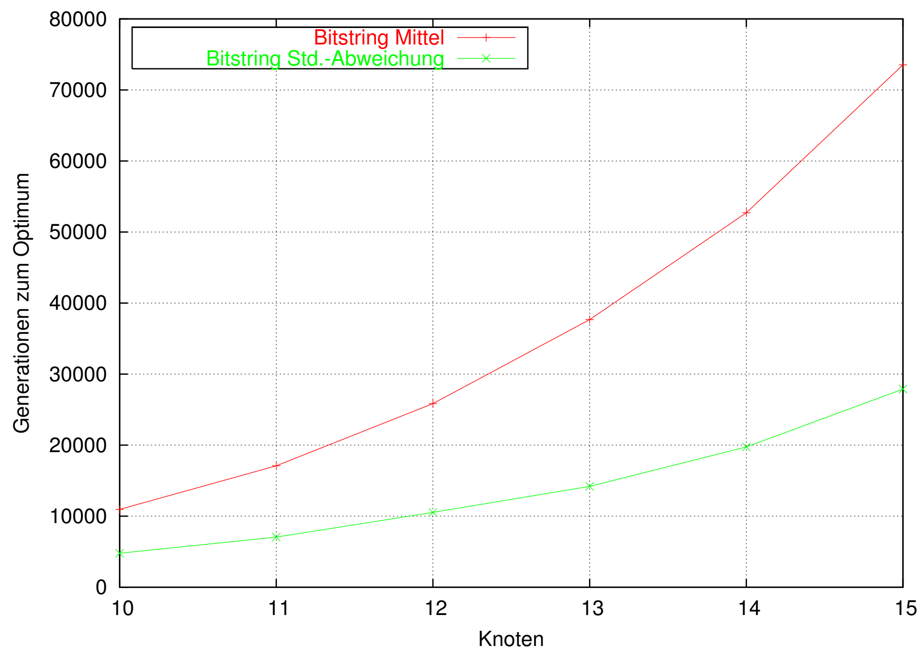


Abbildung 9.21: Durchschnittliche Laufzeiten und Standardabweichungen des (1+1) EA mit Bitstring-Darstellung auf dem MST-Problem auf vollständigen Graphen mit sternförmigem MST.

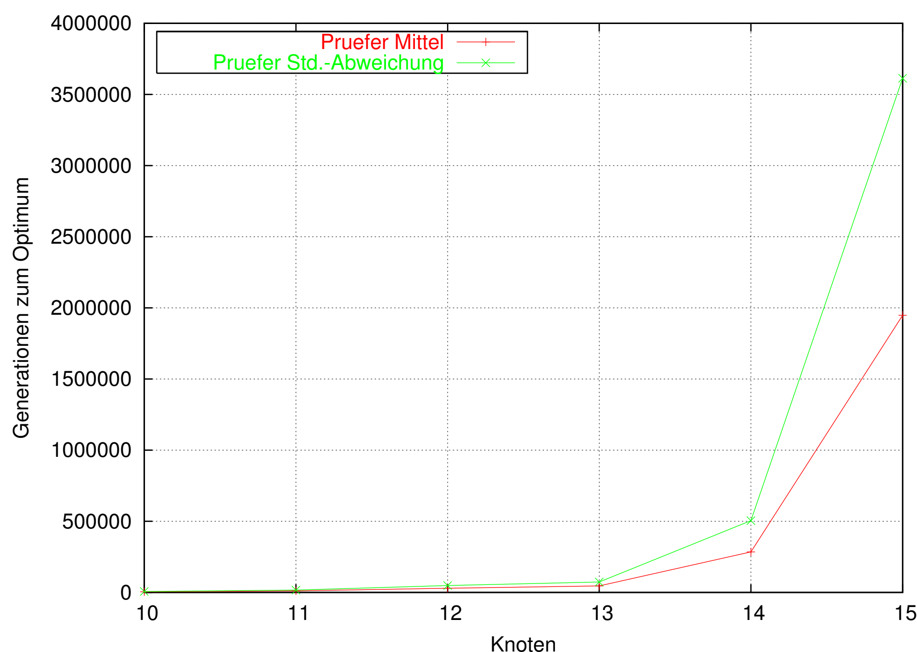


Abbildung 9.22: Durchschnittliche Laufzeiten und Standardabweichungen des (1+1) EA mit Prüfernummer-Darstellung auf dem MST-Problem auf vollständigen Graphen mit sternförmigem MST.

eindeutigen minimalen Spannbaum gibt, wird dieser Flip wahrscheinlich so lange dauern, wie es die Diagramme verdeutlichen.

Auch diese Ergebnisse wurden auf ihre statistische Signifikanz hin mit dem Mann-Whitney-Test abgesichert und auch hier ergab sich eine hohe Signifikanz von gerundet 0,000 bei allen untersuchten Gruppen.

Fazit zur Darstellung von Spannbäumen als Prüfernummern

Diese Experimente lassen uns zu einem zweideutigen Schluss kommen. Einerseits arbeitet die Prüfernummer-Darstellung auf wenigen Kantengewichten und geringer Knotenzahl hervorragend und übertrifft somit die Bitstring-Darstellung, andererseits kommt es bei allgemeineren Graphen oft zu langen Blockaden. Der Algorithmus läuft dort in eine Falle, aus der er nur schwer wieder heraus kommt. Einfache Kantenflips werden zu komplexen Mutationen, auf die lange gewartet werden muss.

9.2.5 Vergleich zweier Fitnessfunktionen

Es sollten die beiden Fitnessfunktionen aus [33] miteinander verglichen werden. Die Funktionen unterscheiden sich in der Bewertung von Suchpunkten darin, dass die eine Funktion einen

Parameter mehr bewertet als die andere. Bei der ersten Funktion, im Folgenden w genannt, ergibt sich der Funktionswert eines Individuums s als

$$w(s) = (c(s) - 1) \cdot w_{\text{ub}}^2 + (e(s) - (n - 1)) \cdot w_{\text{ub}} + \sum_{i|s_i=1} w_i$$

wobei $c(s)$ die Zahl der Zusammenhangskomponenten ist, die sich ergibt, wenn man nur die in s ausgewählten Kanten betrachtet, und $e(s)$ die Anzahl ausgewählter Kanten in s ist. Die Zahl w_{ub} ist definiert als $n^2 \cdot w_{\max}$, also als „n^2 mal maximales Kantengewicht“. In der ganz rechts stehenden Summe werden die Gewichte der ausgewählten Kanten aufsummiert. Diese Funktion gewichtet also nach fallender Wichtigkeit die Zahl der Zusammenhangskomponenten, die Zahl ausgewählter Kanten und das Gewicht aller ausgewählten Kanten, und zwar dominiert jeder der drei Parameter alle folgenden Parameter.

Die zweite Funktion, im Folgenden w' genannt, unterscheidet sich von der ersten in der Tatsache, dass die Zahl der ausgewählten Kanten nicht einbezogen wird. Sie lautet also

$$w'(s) = (c(s) - 1) \cdot w_{\text{ub}} + \sum_{i|s_i=1} w_i$$

Es sollte untersucht werden, ob das „mehr an Information“, welches in die Bewertung bei der Funktion w eingeht, dazu führt, dass der (1+1) EA auf dieser Funktion schneller zum Optimum findet als auf w'. In der Analyse in [33] wurden gleiche asymptotische Schranken für die Laufzeit ermittelt, aber es könnten zum Beispiel konstante Faktoren auftreten, um die sich die Laufzeiten unterscheiden.

Experiment-Aufsatz

Es wurden für Knotenzahlen n von 12 bis 48 in Viererschritten Experimente durchgeführt. Für jede Knotenzahl wurden 500 verschiedene Graphen zufällig erzeugt. Dabei fand die Erzeugung folgendermaßen statt: jede mögliche Kante wurde mit Wahrscheinlichkeit 1/2 in den Graph eingefügt. Die Kantengewichte wurden für die dann im Graphen existierenden Kanten gleichverteilt aus dem Bereich der ganzen Zahlen zwischen 1 und der Knotenzahl des Graphen gewählt.

Für jeden Graphen wurde der (1+1) EA jeweils einmal auf der Funktion w und auf der Funktion w' laufen gelassen und die Zahl der bis zum Erreichen des Optimums benötigten Generationen gezählt. Es ergaben sich also pro Funktion jeweils 500 verschiedene Generationenzahlen, die miteinander verglichen werden sollten.

Durchschnittswerte

Die durchschnittlichen Generationenzahlen und die zugehörigen Standardabweichungen sind in Tabelle 9.23 und in Abbildung 9.23 dargestellt.

Knotenzahl	Gen. w	SD w	Gen. w'	SD w'
12	5229,824	3362,538	5184,308	3542,468
16	19935,242	10999,353	18847,326	10449,463
20	54212,848	27580,127	52489,358	28151,130
24	118147,448	59777,327	119065,294	57109,945
28	231668,192	111842,460	230031,016	107505,002
32	420134,234	183605,113	401641,870	191265,719
36	696232,320	299072,518	680354,872	322534,090
40	1076293,230	482799,986	1031521,834	456461,405
44	1623908,200	659464,523	1630993,338	755150,627
48	2325065,904	994925,988	2266461,556	1043354,941

Tabelle 9.23: Die mittleren Generationenzahlen und die Standardabweichung für die Funktionen w und w'.

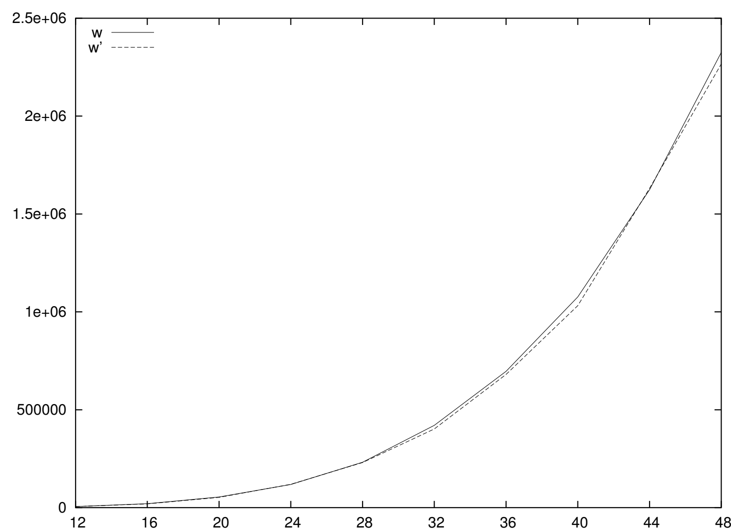


Abbildung 9.23: Die mittleren Generationenzahlen für die Funktionen w und w'.

Knotenzahl	durchschn. Rang w	durchschn. Rang w'	Signifikanz
12	508,41	492,59	0,387
16	514,97	486,03	0,113
20	514,58	486,42	0,123
24	496,65	504,35	0,673
28	501,58	499,42	0,906
32	519,33	481,67	0,039
36	514,75	486,26	0,119
40	512,68	488,32	0,182
44	506,36	494,64	0,521
48	514,59	486,41	0,123

Tabelle 9.24: Die Rangsummentests zu den Generationenzahlen für die Funktionen w und w'.

Wie man sieht, ist der Algorithmus auf der Funktion w' in den meisten Fällen schneller als auf der Funktion w. Das bereits erwähnte „mehr an Informationen“ das die Funktion w liefert, hat also in diesen Experimenten nicht dazu geführt, dass der Algorithmus schneller das Optimum findet. Eine Erklärung dafür könnten Mutationen sein, die bei der Funktion w' akzeptiert werden, bei der Funktion w aber nicht. Nach der theoretischen Analyse wissen wir, dass asymptotisch die meiste Zeit dafür verwendet wird, einen bereits gefundenen Spannbaum zu einem minimalen Spannbaum zu machen. „Gute“ Mutationen, die teure Kanten durch billigere Kanten ersetzen, dabei aber zusätzliche billige Kanten hinzu nehmen, könnten bei der Funktion w' akzeptiert werden, bei der Funktion w aber nicht, weil dort die Kantenzahl die Kantengewichte dominiert.

Rangsummentests

Es wurden dann Mann-Whitney-Tests (Rangsummentests) mit den Daten durchgeführt, um die statistische Signifikanz eines Unterschiedes in den Generationenzahlen zu ermitteln. Die Ergebnisse sind in Tabelle 9.24 aufgeführt.

Die Signifikanzen sind zu schlecht, als dass man nach den üblichen Konventionen von signifikanten Unterschieden sprechen könnte. Dennoch war in der überwiegenden Zahl der betrachteten Fälle der Algorithmus gemessen an den mittleren Laufzeiten auf der Funktion w' schneller als auf der Funktion w. Zudem bewegten sich viele der Signifikanzen immerhin im Bereich zwischen 0,1 und 0,2. Daher blieb die Vermutung bestehen, dass der Algorithmus auf der Funktion w' schneller arbeitet als auf der Funktion w.

Es wurden deshalb weitere Experimente durchgeführt. Aufgrund zu knapper Zeit konnten nicht mehr alle Experimente mit größeren Laufzahlen durchgeführt werden. Deswegen wurde das Experiment für die Knotenzahl 24 wiederholt, eine Knotenzahl, die klein genug war um die Durchführung vieler Läufe zu ermöglichen und für die im ersten Experiment eine schlechtere Signifikanz von 0,673 ermittelt wurde. Die Zahl der Läufe wurde von 500 auf 25350 erhöht. Das Ergebnis ist in Tabelle 9.25 zu sehen. Das Ergebnis ist also hoch signifikant und es kann

durchschn. Rang w	durchschn. Rang w'	Signifikanz
25871,81	24829,19	$< 0{,}001$

Tabelle 9.25: Der Rangsummentest zu 25350 Läufen mit Knotenzahl 24.

vermutet werden, dass die geringe Zahl an Läufen von 500 für die schlechteren Signifikanzen bei den ersten Experimenten verantwortlich war.

Insgesamt wurde als Ergebnis erhalten, dass entgegen der ersten Intuition der (1+1) EA mit der Funktion w' das Optimum schneller gefunden hat als mit der Funktion w.

Regression

Es stellte sich noch die Frage, wie die ermittelten mittleren Generationenzahlen für zufällige Graphen sich zur in der Analyse ermittelten erwarteten Generationenzahl für Worst-Case-Graphen von $\Theta(m^2 \cdot (\log n + \log w_{\max}))$ verhalten. Deswegen wurde mit dem Marquardt-Levenberg-Algorithmus im Programm Gnuplot eine Regressionsanalyse durchgeführt.

Wie erwähnt, wurde jede mögliche Kante bei der zufälligen Erzeugung der Graphen mit Wahrscheinlichkeit 1/2 eingefügt. Man kann sich überlegen, dass wegen der Unabhängigkeit der Zufallsexperimente die erwartete Zahl von Kanten in einem Graphen $1/2 \cdot n(n-1)/2 = \Theta(n^2)$ ist. Dies bedeutet wiederum $m^2 = \Theta(n^4)$. Es wurde deshalb die Funktionenklasse $a \cdot n^4 \cdot \log n$ für die Regression ausgewählt.

Als Ergebnis wurde ein Wert von 0,113752 für a erhalten, was zu einem durchschnittlichen Fehler von 7126,11 führt. Die entstandene Kurve ist zusammen mit den Durchschnittswerten in Abbildung 9.24 dargestellt. Wie man sieht, lässt sich die experimentell erhaltene Generationenzahl gut mit dieser Funktionenklasse approximieren. Man könnte also vermuten, dass die erzeugten durchschnittlichen Graphen „genau so schwer“ sind wie die Worst-Case-Graphen.

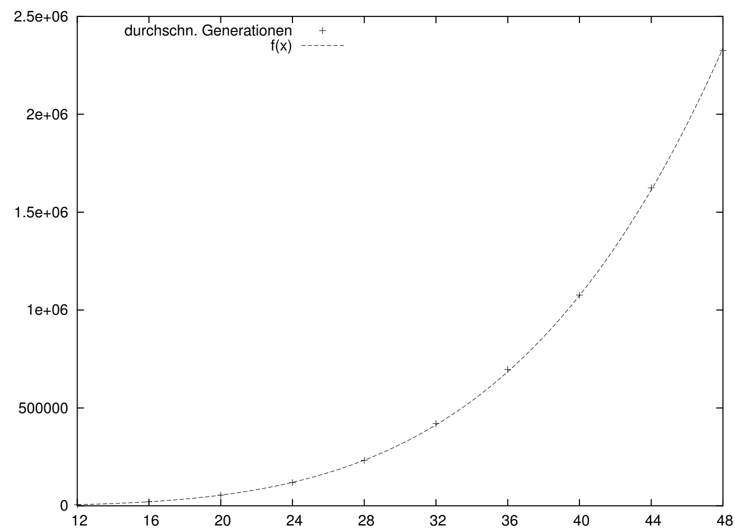


Abbildung 9.24: Die Laufzeit des (1+1) EA auf zufälligen Graphen und die mit Regression erhaltene Funktion.

9.3 Maximale Matchings

9.3.1 Matchings auf Bäumen und Pfaden

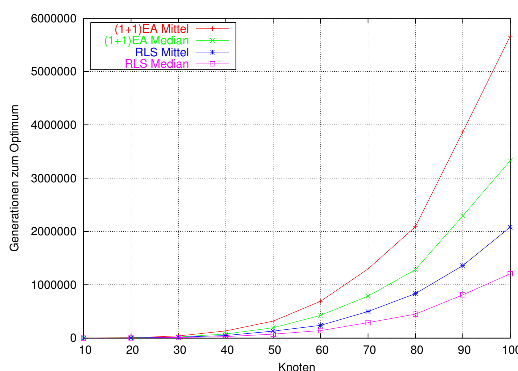
Hypothese 22: Vergleich des (1+1) EA und der RLS auf Pfaden.

Zunächst ging es also darum, das Laufzeitverhalten des (1+1) EA und der RLS auf maximalen Matchings von Pfaden zu vergleichen. Der offensichtliche Weg dies zu tun ist, etliche Läufe zu starten und Mittelwerte und Median-Werte zu bilden. Dabei sah der Versuchsaufbau wie folgt aus:

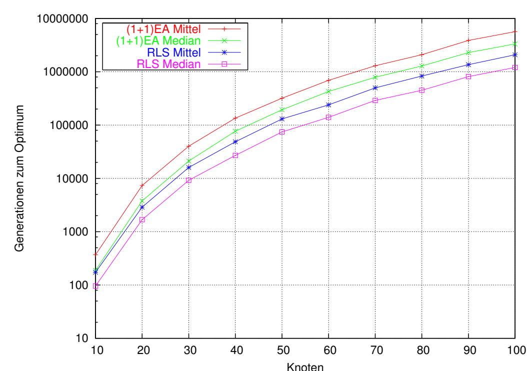
Graph-Typ:	Pfad
Anzahl Knoten:	10 bis 100 in Zehnerschritten
Läufe:	1000
Algorithmus:	(1+1) EA und RLS mit 1- und 2-Bit-Flips

Da ein Pfad auf n Knoten stets eindeutig ist, brauchten wir keine neuen zufälligen Graph-Instanzen für jeden Lauf auszuwürfeln. Es wurden auf demselben Graphen lediglich neue Läufe gestartet. Außerdem wurden aufgrund der doch recht hohen Laufzeit bei ansteigender Knotenzahl nur 1000 Läufe durchgeführt, was die Aussage über Durchschnittswerte aber nicht stark beeinflusst hat.

Die randomisierte lokale Suche beschränkte sich wie schon erwähnt auf 1- und 2-Bit-Flips – aus dem Grund, dass dies die wohl entscheidenden Flips sind, um von einem beliebigen Matching zu einem maximalen Matching zu gelangen. Dabei wurde in jeder Generation jeweils mit Wahrscheinlichkeit 1/2 eine 1-Bit- bzw. 2-Bit-Mutation durchgeführt. Wie bei einer RLS üblich wurde aber in jedem Fall mutiert. Der (1+1) EA arbeitete hingegen mit der allseits bekannten Mutationswahrscheinlichkeit von $1/n$ für jedes Bit.



(a) normale Skalierung



(b) logarithmische Skalierung

Abbildung 9.25: Laufzeiten des (1+1) EA und der RLS auf Pfaden.

Die in Abbildung 9.25 (a) angegebenen Laufzeiten bestätigen zunächst einmal den Verdacht, dass die RLS signifikant schneller das Optimum findet als der (1+1) EA. Dieser signifikante Unterschied wurde uns auch durch einen Mann-Whitney-Test in SPSS bestätigt: die durchgeführten Signifikanztests ergaben alle ein Signifikanzniveau von gerundet 0,000. Offensichtlich ist es zumindest für Knotenzahlen zwischen 10 und 100 tatsächlich ein Vorteil, sich auf 1- und 2-Bit-Flips zu beschränken.

Natürlich stellten wir uns die Frage, inwiefern sich nun die Kurven unterscheiden. Dazu betrachteten wir uns diese in einer logarithmischen Skala (Abbildung 9.25 (b)) und stellten fest, dass die Messwerte im Wesentlichen den gleichen Abstand in der logarithmischen Skala haben. Vermutlich liegen sie also um einen konstanten Faktor auseinander. Um dies zu testen, haben wir die jeweiligen Mittelwerte und Median-Werte dividiert und das Ergebnis aus Abbildung 9.26 erhalten. Zum Vergleich ist die eulersche Zahl e ebenfalls im Diagramm zu sehen. Man könnte behaupten, beide Algorithmen liegen in etwa um Faktor e auseinander, aber dazu sind unsere Ergebnisse sicher nicht aussagekräftig genug.

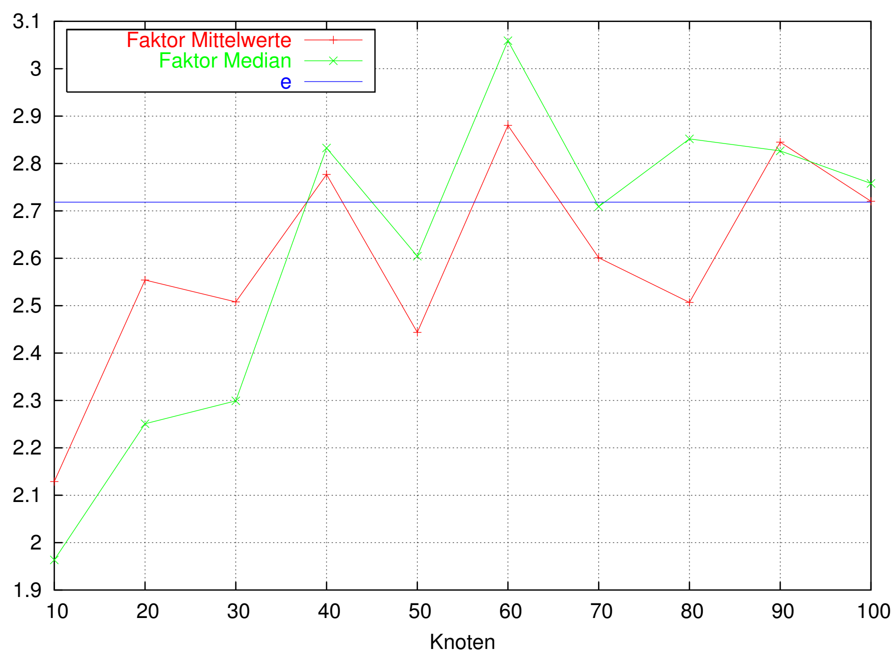


Abbildung 9.26: Faktorieller Unterschied zwischen (1+1) EA und RLS.

Hypothese 23: Der (1+1) EA optimiert maximale Matchings in Zeit $\Theta(n^4)$.

Im Seminarvortrag über maximale Matchings wurde erwähnt, dass die erwartete Laufzeit des (1+1) EA auf Pfaden vermutlich $\Theta(n^4)$ beträgt. Um dies zu überprüfen, approximierten wir die Mittelwerte aus Abbildung 9.25 (a) des (1+1) EA durch Polynome. Wir gaben dabei Funktionenklassen vor, die wir mit dem Programm Gnuplot einer Regressionsanalyse unterzogen. Die zwei von uns am besten empfundenen Regressionsergebnisse sind in Abbildung 9.27 zu erkennen. Die entsprechenden Funktionen sind dabei die folgenden:

1. $ax^4 + bx^3 + cx^2 + dx$
   Dabei ergab sich für die Koeffizienten:

a	=	0,061511
b	=	0,110115
c	=	-72,6084
d	=	1814,67

2. ax^4
   Dabei ergab sich für die Koeffizienten:

a	=	0,0565228

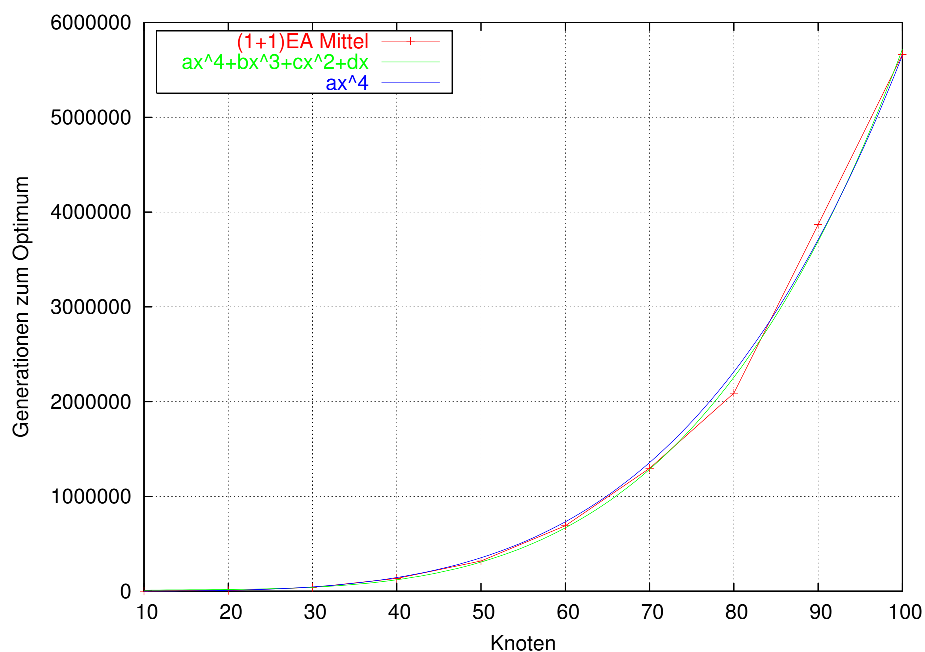


Abbildung 9.27: Regressionsanalyse des (1+1) EA auf Pfaden.

Die so gewonnenen Funktionen approximieren die Messwerte hinreichend gut, jedoch fällt in beiden Fällen stets der kleine Vorfaktor vor dem x^4 auf. Ein Versuch, diesen Term auszulassen und eine Regressionsanalyse mit einer Funktionenklasse vom Grad 3 durchzuführen, brachte allerdings ein unbefriedigendes Ergebnis.

Neugierigerweise führten wir dieselbe Regressionsanalyse mit den Mittelwerten der RLS durch. Die Funktionen sind in Abbildung 9.28 zu sehen. Folgende Funktionenklassen wurden untersucht:

1. $ax^4 + bx^3 + cx^2 + dx$
   Dabei ergab sich für die Koeffizienten:

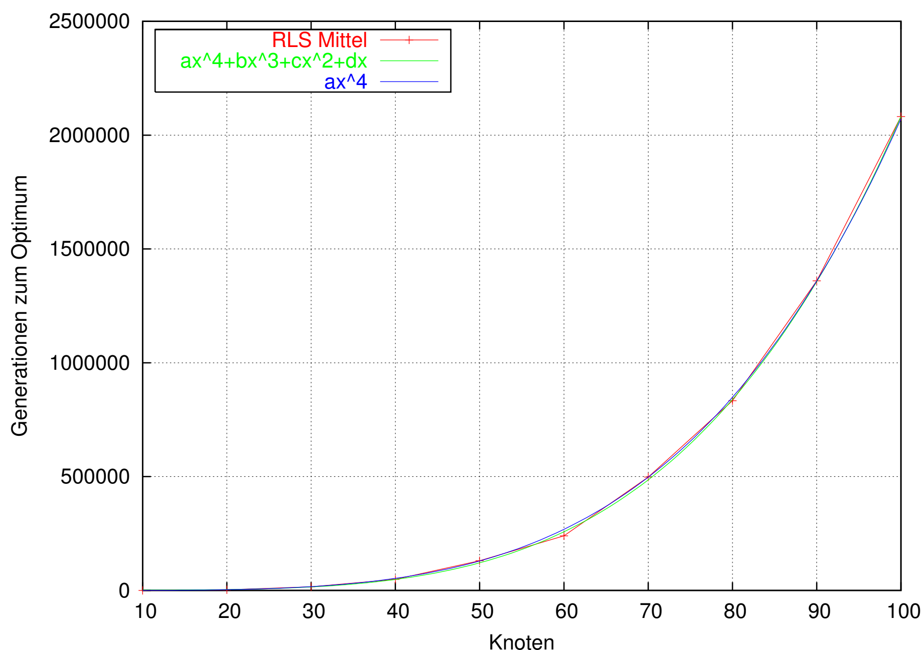


Abbildung 9.28: Regressionsanalyse der RLS auf Pfaden.

a	=	0,0215092
b	=	0,0151366
c	=	-10,7037
d	=	234,337

2. ax^4
   Dabei ergab sich für die Koeffizienten:

a	=	0,0207183

Auch hier wunderten wir uns über die kleinen Vorfaktoren vor x^4 und auch hier waren wir mit den daraus resultierenden Regressionskurven eigentlich recht zufrieden. Ein Weglassen des x^4-Terms führte auch hier zu schlechteren Ergebnissen.

Wir konnten also unsere Vermutung untermauern, dass der (1+1) EA maximale Matchings in erwarteter Zeit $\Theta(n^4)$ bestimmt, können uns allerdings die niedrigen Faktoren nicht weiter erklären.

Hypothese 24: Der (1+1) EA ist auf Pfaden langsamer als auf zufälligen Bäumen.

Aus dem Vortrag war uns bekannt, dass Pfade in der Tat schwer zu optimieren sind, da verbessernde Teil-Pfade unter Umständen lang sein können und komplett geflippt werden müssen, um tatsächlich eine Verbesserung herbeizuführen. Man vermutet also, dass Pfade als entartete Bäume signifikant schlechter durch den (1+1) EA zu optimieren sind als beliebige

zufällige Bäume. Schon bei der Hypothesenfindung stellten wir fest, dass diese Vermutung wahrscheinlich korrekt ist. So entschlossen wir uns, Bäume mit bis zu 200 Knoten zu testen, wohingegen wir bei Pfaden bis maximal 100 Knoten gehen konnten, ohne dass wir die Befürchtung haben mussten, mit den Experimenten nicht fertig zu werden. Der Versuchsaufbau war der folgende:

Graph-Typ:	Pfad / zufälliger Baum
Anzahl Knoten:	10 bis 100 in Zehnerschritten bei Pfaden
	10 bis 200 in Zehnerschritten bei Bäumen
Läufe bzw. Bäume:	1000
Algorithmus:	(1+1) EA

Um ein möglichst großes Spektrum von Bäumen in den Test mit einzubeziehen, haben wir bei den zufälligen Bäumen für jeden Lauf einen neuen zufälligen Baum ausgewürfelt. Das Ergebnis war ein sehr deutlicher Vorsprung der Läufe auf den zufälligen Bäumen gegenüber den Läufen auf den Pfaden, wie Abbildung 9.29 zeigt. (Zum Vergleich wurden in Abbildung 9.29 ebenfalls die Mittelwerte der RLS eingetragen.) Mit SPSS wurde dieser Unterschied als hoch signifikant herausgestellt (Signifikanzniveau $p \leq 0{,}001$).

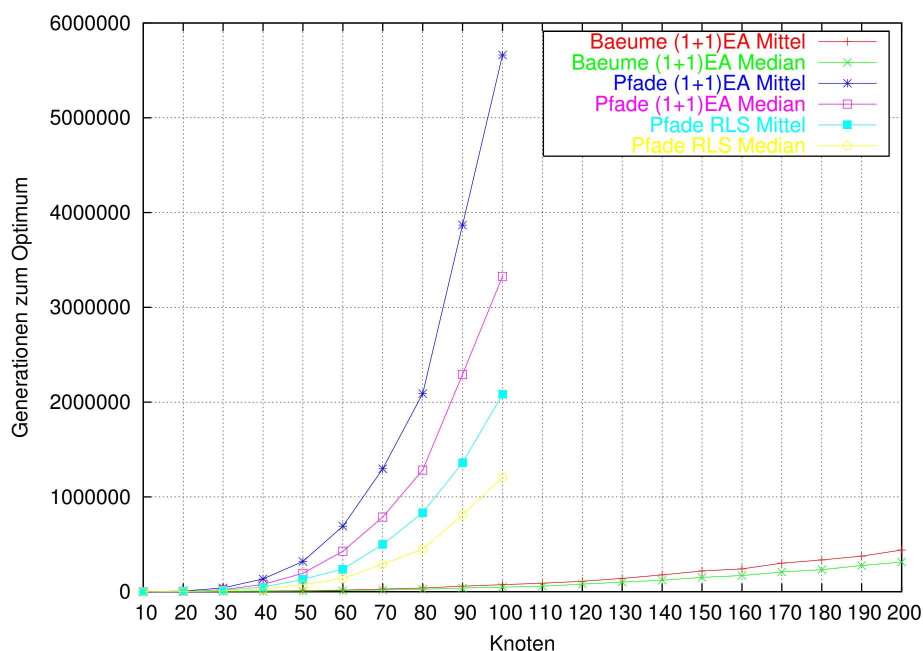


Abbildung 9.29: Vergleich der Laufzeiten auf Pfaden und zufälligen Bäumen.

Interessant ist hierbei die asymptotische Laufzeit, die wir hier ebenfalls wieder mit einer Regressionsanalyse versuchten zu bestimmen. Es stellte sich dabei eine Funktion vom Grad 3 als gute Näherung heraus. Die entsprechende Funktion ist in Abbildung 9.30 zu sehen.

Die betrachtete Funktionenklasse lautet:

- $ax^3 + bx^2 + cx$
  Dabei ergab sich für die Koeffizienten:

a	=	0,0271795
b	=	6,48705
c	=	-192,146

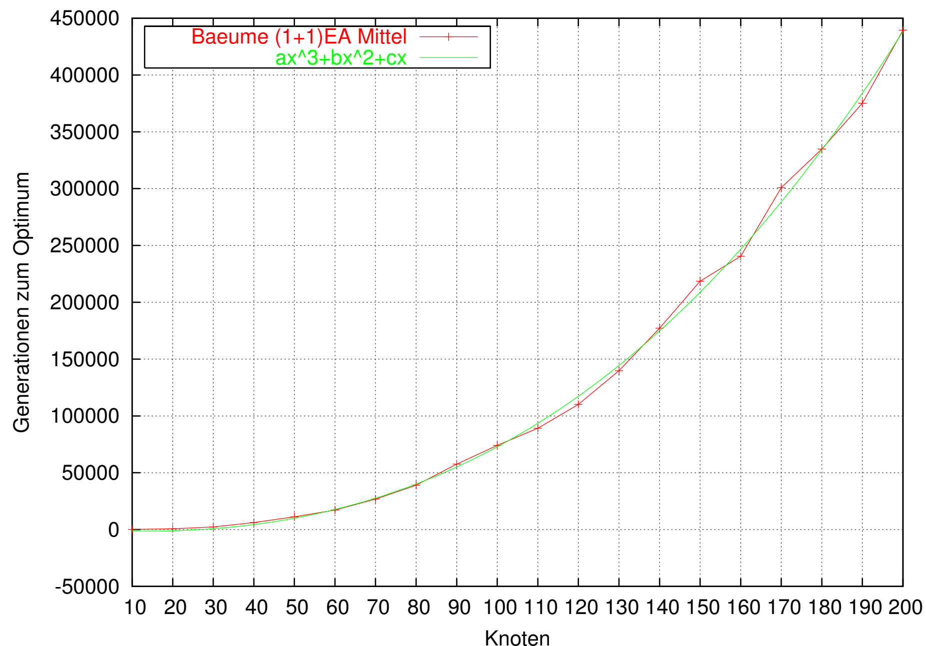


Abbildung 9.30: Regressionsanalyse der Laufzeit des (1+1) EA auf Bäumen.

Eine mögliche Schlussfolgerung hierzu könnte sein, dass zufällige Bäume um den Faktor n schneller optimiert werden als Pfade derselben Länge.

Beobachtung: k-äre Bäume

Die im Seminarvortrag angesprochenen k-ären Bäume betrachteten wir als Letztes. Als direkten Vergleich wählten wir die Optimierung von zufälligen Bäumen auf n Knoten. Der Versuchsaufbau ist der gleiche wie in Abschnitt 9.3.1. Das Ergebnis, welches in Abbildung 9.31 dargestellt wird, verwunderte uns jedoch. Im Wesentlichen blieben die Laufzeiten zwar unter denen der zufälligen Bäume, allerdings tauchten an scheinbar willkürlichen Stellen plötzlich starke Einbrüche der Laufzeit auf.

Da wir spontan keine Erklärung für die merkwürdigen Schwankungen in der Laufzeit hatten, beschlossen wir, eine solche Anomalie auszuwählen und Experimente durchzuführen, die das Ergebnis verfeinern sollten. Wir wählten den Laufzeit-Einbruch zwischen 150 und 170 Knoten und stellten folgendes Experiment zusammen:

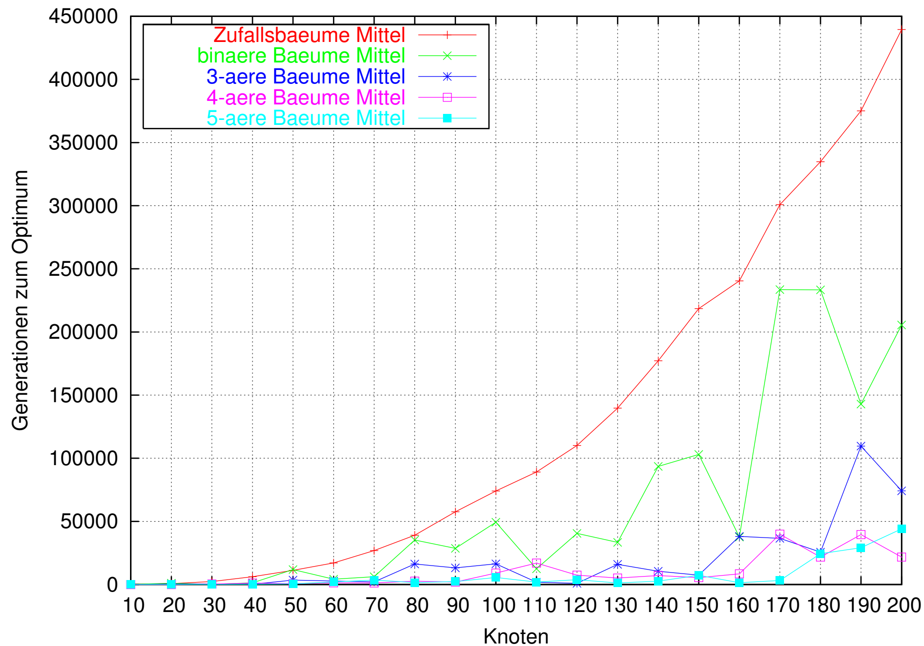


Abbildung 9.31: Laufzeiten des (1+1) EA auf zufälligen und k-ären Bäumen.

Graph-Typ:	binärer Baum
Anzahl Knoten:	150 bis 170
Läufe:	2000
Algorithmus:	(1+1) EA

Die Anzahl der Knoten wurde diesmal in Einerschritten erhöht und die Anzahl der Läufe verdoppelten wir auf 2000, um etwaige Abweichungen zu relativieren. Das Ergebnis bestätigte unsere vorherige Beobachtung und warf mehr Fragen als Antworten auf. Abbildung 9.32 zeigt im Bereich von 150 bis 170 Knoten wiederum starke Schwankungen von einer Knotenzahl zur nächsten. Als einziger Erklärungsansatz diente die Tatsache, dass bei unserer bislang benutzten Implementierung von vollständigen k-ären Bäumen stets die unterste Ebene soweit möglich von links nach rechts mit Blättern gefüllt wurde. Dadurch ergeben sich bei der Bestimmung eines maximalen Matchings andere Verhältnisse als wenn die Positionen der Blätter auf der untersten Ebene zufällig gewählt worden wären.

Also entschlossen wir uns, auch die zweite Implementierung (mit zufällig angeordneter untersten Ebene) zu testen und siehe da, Abbildung 9.33 zeigt ein halbwegs monotones Verhalten bei der Benutzung dieser Implementierung. Interessanterweise liegen dabei die Laufzeiten deutlich höher als bei einer linear von links nach rechts gefüllten untersten Ebene. Betrachtet man die Funktionswerte der gefundenen Optima, liegen diese im Schnitt bei zufällig angeordneter unterster Ebene auch einige Zähler höher. Es gibt also mehr Kanten, die zu einem optimalen Matching gehören.

Die letzte Frage, die sich uns in diesem Zusammenhang stellte war, wie sich wohl die Implementierung der k-ären Bäume mit zufällig angeordneter unterster Ebene auf dem gesamten

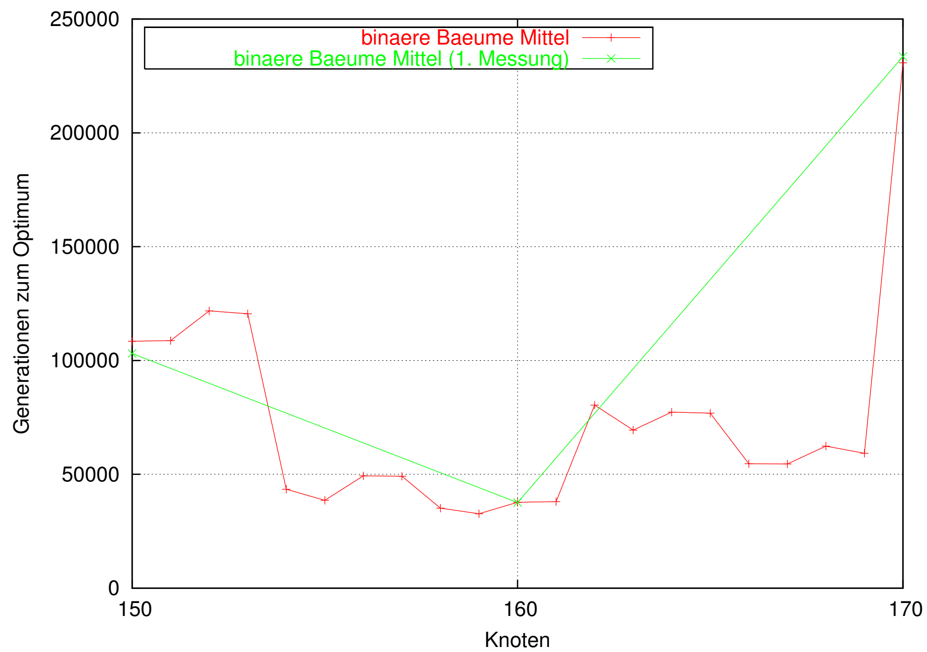


Abbildung 9.32: Laufzeiten des (1+1) EA auf binären Bäumen im Bereich von 150 bis 170 Knoten. Die andere Linie (grün) repräsentiert die ursprünglich bei 150, 160 und 170 Knoten gewonnenen Messwerte.

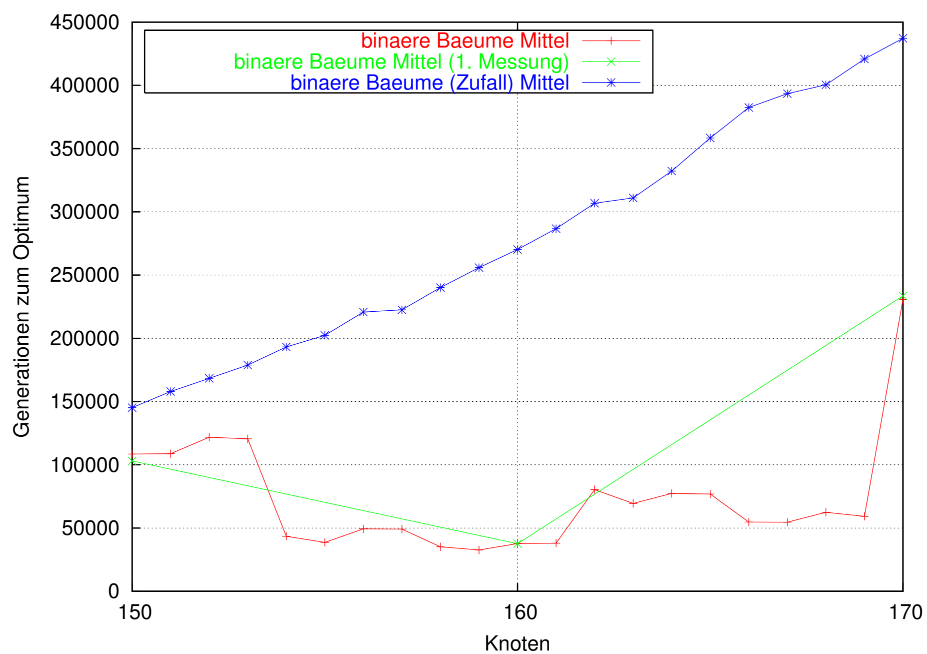


Abbildung 9.33: Laufzeiten des (1+1) EA auf binären Bäumen mit und ohne zufällig angeordneter unterster Ebene im Bereich von 150 bis 170 Knoten.

Experimentbereich von 10 bis 200 Knoten darstellt. Leider erwies sich dabei unsere Hoffnung, eine halbwegs monotone Messkurve zu erhalten, als falsch. Ein zusammenfassendes Diagramm über die Laufzeiten aller betrachteten Baumklassen zeigt Abbildung 9.34.

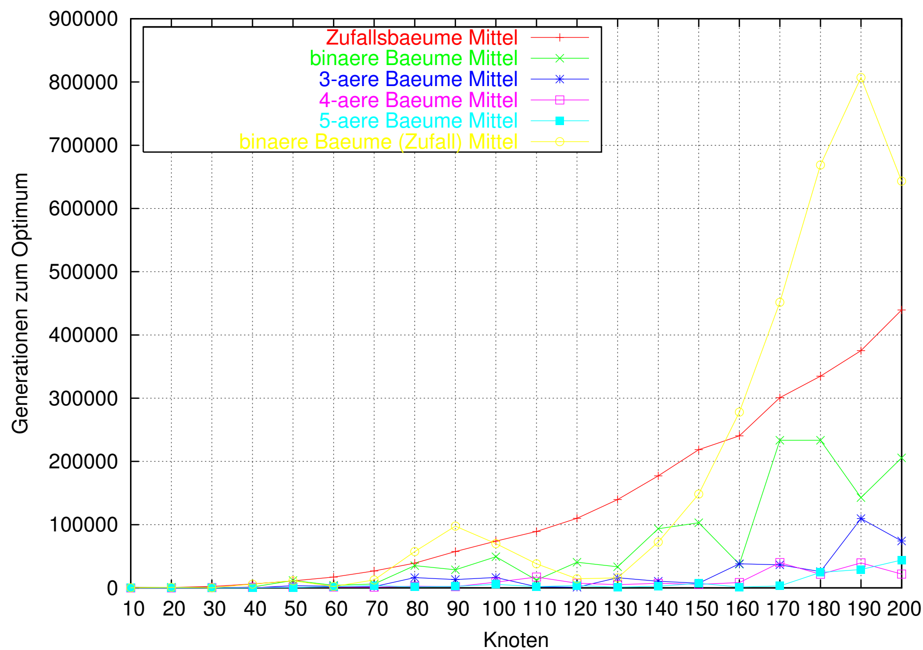


Abbildung 9.34: Laufzeiten des (1+1) EA auf allen betrachteten Baumklassen.

Fazit zu Matchings auf Bäumen und Pfaden

Zusammenfassend lässt sich sagen, dass sich alle aufgestellten Hypothesen experimentell und statistisch bewahrheitet haben. Pfade sind in der Tat, zumindest im Vergleich mit Bäumen, schwer zu optimierende Graphen, ihre erwartete Optimierungszeit liegt bei $\Theta(n^4)$ und randomisierte lokale Suche ist möglicherweise in etwa um den Faktor e schneller. Zufällige Bäume sind im Vergleich dazu nochmal um den Faktor n schneller zu optimieren, so dass eine erwartete Laufzeit von $O(n^3)$ nicht auszuschließen ist. Im Vergleich dazu liefern die Experimente zu k-ären Bäumen stark schwankende Ergebnisse, die stark von der Anordnung der Blätter auf der untersten Ebene abhängen. Genauere Analysen hierzu sind sicherlich noch eine lohnenswerte Aufgabe.

9.3.2 Nutzen von Uniformem Crossover bei Matchings auf Pfaden

Hypothese 25: die (2+3) RLS mit uniformem Crossover hat eine kleinere Laufzeit als die (2+2) RLS.

Es wurden jeweils 1000 Läufe zwischen 10 und 100 Knoten in 10er-Schritten durchgeführt. Dabei haben wir sehr starke Schwankungen zwischen den Laufzeiten festgestellt, z. B. gab

es bei 100 Knoten bei der (2+3) RLS mit uniformem Crossover einen Lauf mit 19.850.006 Generationen, aber auch einen sehr kurzen Lauf mit nur 18.689 Generationen.

	Mittelwert		Median	
Dimension	(2+3) RLS+Uni.	(2+2) RLS	(2+3) RLS+Uni.	(2+2) RLS
10	103,86	93,52	46	47
20	2393,09	2104,34	1207	1064
30	15273,53	12040,84	8300	6478
40	50758,24	41465,51	28169	22205
50	120782,42	101134,87	70111	55145,5
60	263777,95	226099,11	155067,5	132194,5
70	454342,74	439254,62	267179	236458,5
80	806102,91	775879,13	434662	440796,5
90	1303242,12	1304635,14	749650,5	766449,5
100	2145063,4	1971263,5	1199882	1127631

Tabelle 9.26: Die Tabelle zeigt den Mittelwert und den Median der Laufzeiten von (2+2) RLS und (2+3) RLS mit uniformem Crossover über die verschiedenen Dimensionen (= Knotenzahl).

Anhand der Daten in Tabelle 9.26 kann man sehen, dass uniformes Crossover eher schadet als hilft. Aus diesem Grund haben wir eine gegenteilige Hypothese aufgestellt:

Hypothese 31. *Die (2+3) RLS mit uniformem Crossover benötigt für maximale Matchings auf Pfaden im Durchschnitt mehr Generationen als die (2+2) RLS.*

Um diese zu bestätigen haben wir den Mann-Whitney-Test angewendet. Die erhaltenen Signifikanzen sind in Tabelle 9.27 aufgeführt. Wie man dort erkennen kann, lassen die Ergebnisse keine eindeutige Schlussfolgerung zu. Im Bereich von 30 bis 60 Knoten schadet uniformes Crossover mit statistisch hoher Signifikanz. Für die anderen Knotenzahlen lässt sich kein signifikanter Schaden bestätigen.

Bei der Optimierung von Pfaden vollführen die Algorithmen die meiste Zeit einen Random Walk. Eine Erklärung für das schadhafte Verhalten von uniformen Crossover könnte sein, dass die durch uniformes Crossover erzeugten Individuen eher in der Mitte eines Plateaus landen, es aber besser ist, an eines der Enden des Plateaus zu kommen. Somit verdrängt das neue Individuum einen seiner Eltern, der sich eventuell näher an einem Ende des Plateaus befunden hat.

Fazit zum Nutzen von Uniform Crossover bei Matchings auf Pfaden

Auf jeden Fall konnte eines gezeigt werden: Uniformes Crossover ist zumindest für diesen Versuchsaufbau nicht geeignet, maximale Matchings auf Pfaden schneller zu optimieren.

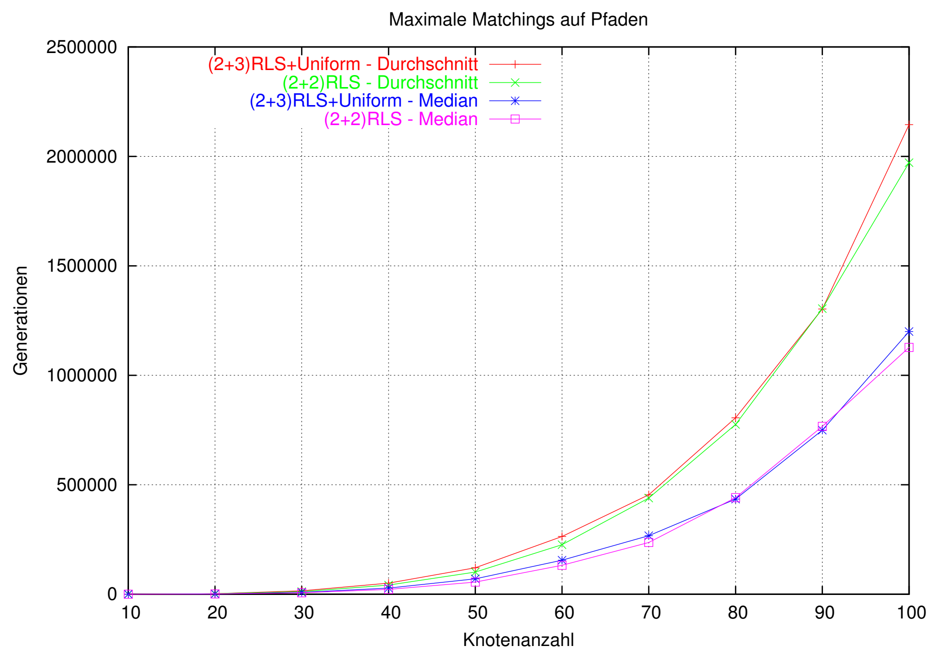


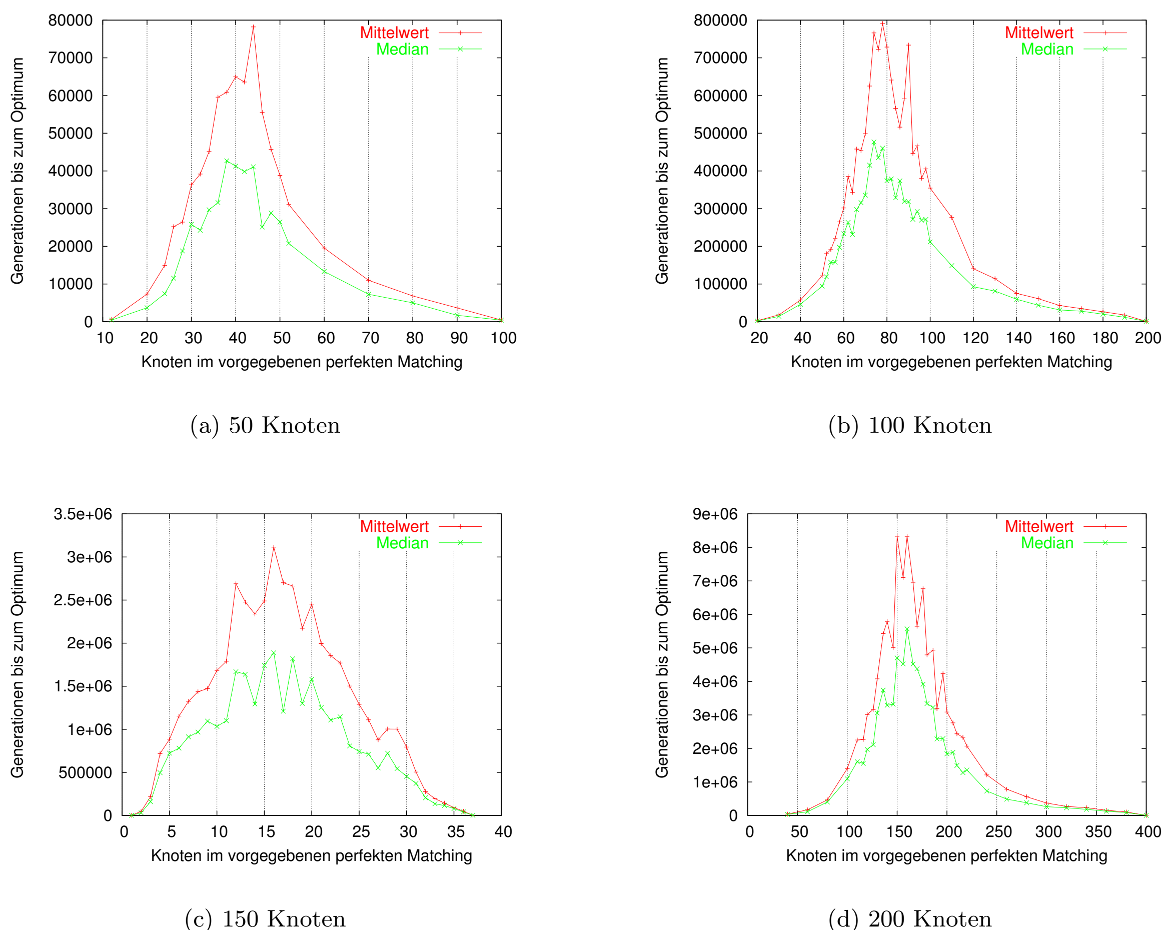
Abbildung 9.35: Vergleich der Algorithmen (2+3) RLS+Uniform und (2+2) RLS auf maximalen Matchings auf Pfaden.

Knotenzahl	Stat. Signifikanz
10	0,61532
20	0,18708
30	0,00018
40	0,00050
50	0,00008
60	0,00084
70	0,07945
80	0,61594
90	0,86889
100	0,17904

Tabelle 9.27: Ergebnis des Mann-Whitney-Test auf den gesammelten Daten von (2+3) RLS+Uniform und (2+2) RLS.

9.3.3 Matchings auf Semi-Random-Graphen

Wie schon in Abschnitt 8.3.3 erläutert, wollen wir uns den Einfluss von störenden Kanten ansehen, wenn ein perfektes Matching bereits vorgegeben war. Dementsprechend haben wir zwei Parameter, die eingestellt werden müssen: die Dimension des Suchraums, also die Anzahl an Kanten im Graph, und die Größe des (vorgegebenen) perfekten Matchings.



(a) 50 Knoten

(b) 100 Knoten

(c) 150 Knoten

(d) 200 Knoten

Abbildung 9.36: Laufzeiten des (1+1) EA auf Semi-Random-Graphen mit verschiedenen vorgegebenen perfekten Matchings.

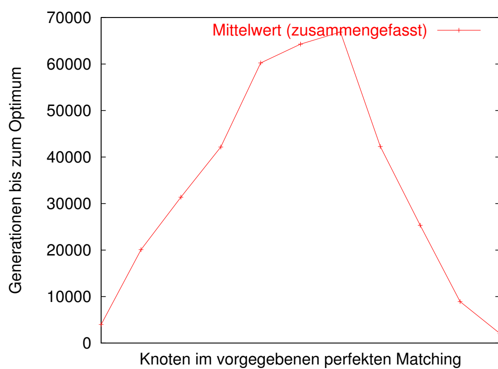
Wir haben uns dazu entschieden mehr Wert auf die Größe des perfekten Matchings zu legen und daher nur vier verschiedene Suchraumgrößen untersucht: 50, 100, 150 und 200 Kanten bzw. Bits. In jeder Suchraumdimension haben wir die Größe des Matchings langsam von n_0 bis n_1 erhöht und bei jeder Konfiguration 100 Läufe durchgeführt. Die dabei entstandenen Laufzeitkurven sind in Abbildung 9.36 gezeigt.

Die Verläufe entsprechen (vermutlich durch statistische Abweichungen) nicht ganz dem in Hypothese 26 angenommenen Aussehen. Die Kurven sind zwar im Wesentlichen im ersten

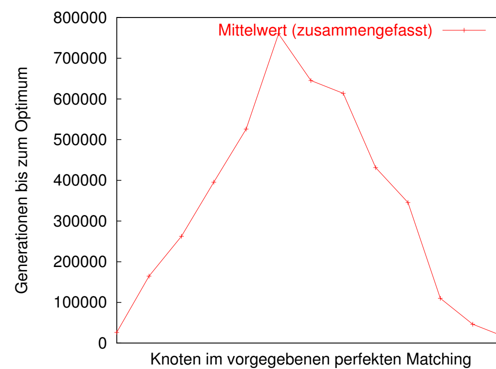
Teil steigend und danach fallend, allerdings eben nicht ganz.

Dies führte dazu, dass wir die Tests etwas abschwächen mussten, um statistisch signifikante Ergebnisse zu erhalten. Wir konnten nicht jeweils zwei benachbarte Punkte in der Kurve mittels eines Wilcoxon-Rangsummentests miteinander vergleichen, sondern mussten die Kurven etwas vergröbern.

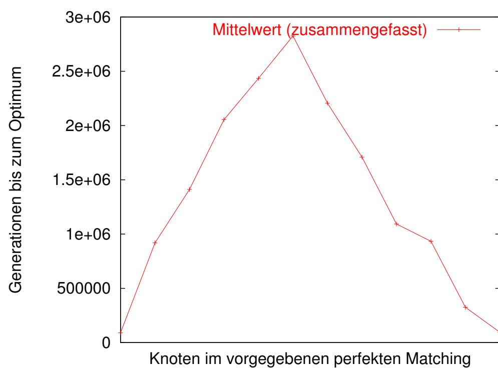
Dabei haben wir (siehe auch Abschnitt 9.1.1) jeweils mehrere benachbarte Konfigurationen zusammengefasst, und dann darauf die Tests angewendet. Die nun entstandenen Kurven sind in Abbildung 9.37 darstellt, die dazu gehörigen Testergebnisse in Tabelle 9.28.



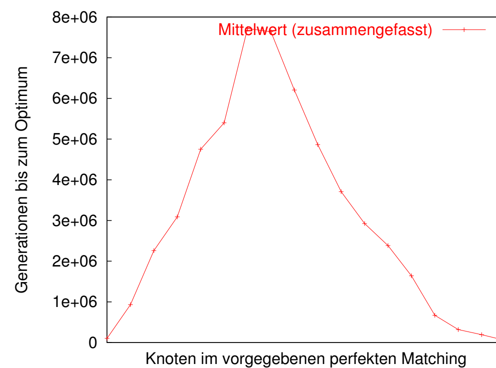
(a) 50 Knoten



(b) 100 Knoten



(c) 150 Knoten



(d) 200 Knoten

Abbildung 9.37: Zusammengefasste Laufzeiten des (1+1) EA auf Semi-Random-Graphen mit verschiedenen vorgegebenen perfekten Matchings.

Wie man der Tabelle entnehmen kann, wurden die nun abgeschwächten Tests voll und ganz bestätigt. Die Kurven sind erst monoton steigend und anschließend monoton fallend. Lediglich in der Nähe des Maximums wurden unsere Tests nicht erfüllt, was aber nicht verwunderlich ist und zu erwarten war (die rot gekennzeichneten Werte).

50 Knoten		100 Knoten		150 Knoten		200 Knoten	
Test	Signifikanz	Test	Signifikanz	Test	Signifikanz	Test	Signifikanz
1-2	0,001	1-2	0,001	1-2	0,001	1-2	0,001
2-3	0,001	2-3	0,001	2-3	0,001	2-3	0,001
3-4	0,017	3-4	0,001	3-4	0,001	3-4	0,009
4-5	0,008	4-5	0,001	4-5	0,001	4-5	0,001
5-6	0,334	5-6	0,001	5-6	0,001	5-6	0,240
6-7	0,854	6-7	0,013	6-7	0,012	6-7	0,005
7-8	0,018	7-8	0,909	7-8	0,908	7-8	0,663
8-9	0,001	8-9	0,004	8-9	0,003	8-9	0,047
9-10	0,001	9-10	0,003	9-10	0,002	9-10	0,026
10-11	0,001	10-11	0,001	10-11	0,001	10-11	0,008
		11-12	0,001	11-12	0,001	11-12	0,028
		12-13	0,001	12-13	0,001	12-13	0,011
						13-14	0,002
						14-15	0,001
						15-16	0,001
						16-17	0,001
						17-18	0,001

Tabelle 9.28: Ergebnisse der Tests auf strenge Monotonie.

Fazit zu Matchings auf Semi-Random-Graphen

Unsere Hypothese konnte also teilweise bestätigt werden. Es existiert ein Punkt, bis zu dem die Kurve steigend und anschließend fallend ist. Jedoch können wir dies nur nachweisen, wenn wir unsere Daten vergröbern. Dementsprechend steigt und fällt die Kurve nur „im Wesentlichen“. Auch der erwähnte Punkt ist eher ein kleiner Bereich.

9.4 Kürzeste Wege in Graphen

Motiviert durch den Vortrag über multikriterielle Optimierung (siehe Abschnitt 7.9.2) haben wir uns entschlossen, die dort definierte einkriterielle Fitnessfunktion für das Single Source Shortest Path Problem mit der multikriteriellen Variante zu vergleichen. Wir haben ausschließlich Experimente auf vollständigen Graphen durchgeführt, da es sonst ohne eine entsprechende Anpassung der einkriteriellen Fitnessfunktion bei vielen Probleminstanzen zu Situationen kommen kann, in denen es genauso wie bei NEEDLE sehr große Plateaus gibt.

9.4.1 Suchraum, Fitnessfunktionen und Algorithmus

Wir gehen also davon aus, dass die Probleminstanzen durch $n \times n$-Distanzmatrizen $D = (d_{ij})$ mit $d_{ij} \in \mathbb{N}$ gegeben sind und dass wir den Suchraum

$$S = \{(x_1, \dots, x_{n-1}) \in \{1, \dots, n\}^{n-1} \mid \forall i : x_i \neq i\}.$$

vorliegen haben. Es sei noch einmal darauf hingewiesen, dass nicht jedes Element $x \in S$ einen Baum repräsentiert, sondern dass es vorkommen kann, dass einige Knoten auf Kreisen liegen, die nicht vom Knoten n aus erreicht werden können. Das müssen wir bei der Definition der Fitnessfunktionen beachten. Weiterhin gilt o. B. d. A., dass der Knoten, von dem aus die kürzesten Wege berechnet werden sollen, der Knoten n ist. Wir definieren nun für jeden Knoten $i \in \{1, \dots, n-1\}$ eine Funktion $l_i(x) : S \to \mathbb{N} \cup \{\infty\}$. Dabei sind zwei Fälle zu unterscheiden: entweder x ist ein Suchpunkt, der einen Graph repräsentiert, der einen Weg von n nach i enthält oder nicht. Im ersten Fall ist $l_i(x)$ einfach die Länge dieses Pfades. Diese ist durch $n \cdot d^*$ beschränkt, wenn d^* der größte Eintrag der Matrix D ist. Im zweiten Fall definieren wir $l_i(x)$ als ∞.

Damit können wir die einkriterielle Fitnessfunktion $f_s : S \to \mathbb{N}$, die wir getestet haben, definieren als

$$f_s : x \mapsto \sum_{i, l_i(x) \neq \infty} l_i(x) + n \cdot d^* \cdot \sum_{i, l_i(x) = \infty} 1.$$

Wir haben also die Strafkosten für nicht verbundene Knoten von ∞ auf $n \cdot d^*$ reduziert, weil es sonst trotz der vollständigen Graphen zu großen Plateaus mit unendlich großer Fitness gekommen wäre.

Die multikriterielle Fitnessfunktion $f_m : S \to \mathbb{N}^{n-1}$ lässt sich nun einfach definieren als

$$f_m : x \mapsto (l_1(x), \dots, l_{n-1}(x)).$$

Die Algorithmen, die wir untersucht haben, sind ebenfalls in Abschnitt 7.9.2 beschrieben. Wir haben auf dem Suchraum S einen lokalen Mutationsoperator definiert, der genau eine uniform zufällig gewählte Stelle des aktuellen Suchpunktes x ändert. Der neue Wert wird dabei uniform zufällig aus den $n-2$ Möglichkeiten gewählt. Der globale Mutationsoperator würfelt zunächst eine Zahl gemäß der Poisson-Verteilung mit Parameter $\lambda = 1$ aus und führt entsprechend viele lokale Operationen durch. Wir haben den (1+1) EA mit diesem globalen Mutationsoperator angewandt. Dabei ist zu beachten, dass der (1+1) EA das aktuelle Individuum x durch das erzeugte Kind x' genau dann ersetzt, wenn die Fitness von x' größer

oder gleich der Fitness von x ist. Das heißt also insbesondere, dass bei der multikriteriellen Fitnessfunktion ein Kind, dessen Fitnessvektor nicht mit dem Fitnessvektor des aktuellen Individuums vergleichbar ist, verworfen wird.

9.4.2 Experimente

Scharnow et al. [39] haben zwar für die erwartete Laufzeit des (1+1) EA für die multikriterielle Fitnessfunktion eine obere Schranke zeigen können, die auf Worst-Case-Instanzen in der Größenordnung n^3 liegt und für typische realistische Instanzen in der Größenordnung $n^2 \log n$. Die Frage, ob die einkriterielle Fitnessfunktion zu einer kleineren oder größeren erwarteten Laufzeit führt als die multikriterielle, wurde allerdings bislang noch nicht theoretisch behandelt. Es wurde jedoch vermutet, dass der (1+1) EA mit der multikriteriellen Fitnessfunktion schneller das Optimum findet, da sie mehr Informationen zur Verfügung stellt als die einkriterielle Fitnessfunktion.

Wir haben die Fitnessfunktionen auf zwei verschiedenen Klassen von zufälligen Graphen verglichen. Die erste Klasse ist beschrieben durch eine Wahrscheinlichkeitsverteilung $U(n, m)$ auf der Menge aller $(n \times n)$-Distanzmatrizen D. Dabei ist $U(n, m)$ die Gleichverteilung auf der Menge aller symmetrischen $(n \times n)$-Matrizen mit Einträgen aus der Menge $\{1, \ldots, m\}$.

Auch die zweite Klasse ist beschrieben durch eine Wahrscheinlichkeitsverteilung $S(n, m)$, die ausschließlich symmetrischen Matrizen positive Wahrscheinlichkeiten zuordnet, weswegen wir bei der Beschreibung dieser Verteilung von ungerichteten Graphen ausgehen können. Das Zufallsexperiment, das uns zufällige Graphen gemäß der Verteilung $S(n, m)$ liefert, besteht darin, zunächst uniform zufällig eine Teilmenge der Kantenmenge des Graphen mit $n - 1$ Elementen auszuwählen, die einen Baum bildet und damit azyklisch ist. Den Kanten in dieser Menge wird jeweils das Gewicht 1 zugeordnet. Die Gewichte aller anderen Kanten werden danach unabhängig voneinander uniform zufällig aus der Menge $\{2, \ldots, m\}$ gewählt.

Wir haben für beide Klassen die Parameter n und m variiert und für jede Parameterkombination jeweils 1000 Läufe durchgeführt. Für die erste Klasse haben wir Graphen mit $n \in \{10{,}20{,}30, \ldots, 150\}$ Knoten und maximal erlaubten Gewichten $m \in \{2^1, 2^2, \ldots, 2^{\lfloor \log n^2 \rfloor + 1}\}$ getestet. Für die zweite Klasse haben wir Graphen mit $n \in \{10{,}20{,}30, \ldots, 130\}$ Knoten und ebenfalls mit den maximal erlaubten Gewichten $m \in \{2^1, 2^2, \ldots, 2^{\lfloor \log n^2 \rfloor + 1}\}$ getestet.

9.4.3 Ergebnisse

Die Ergebnisse unserer Experimente sind eindeutig ausgefallen. Wir haben für beide Verteilungen und für jede getestete Kombination von n und m auf die beiden zugehörigen Messreihen mit der einkriteriellen und der multikriteriellen Fitnessfunktion einen Mann-Whitney-Test angewandt. Jeder dieser Tests lieferte das Ergebnis, dass die Laufzeiten des (1+1) EA auf der multikriteriellen Fitnessfunktion signifikant größer sind als die Laufzeiten auf der einkriteriellen Fitnessfunktion. Die Signifikanz dieser Ergebnisse lag bei fast jedem Test auf drei Stellen gerundet bei 0.,000. Nur bei den Tests mit der Verteilung $S(n, m)$ gab es für wenige Kombinationen von n und m Messreihen, bei denen das Ergebnis weniger signifikant als 0,000 ist. Das am wenigsten signifikante Ergebnis hat aber immer noch eine Signifikanz von 0,02.

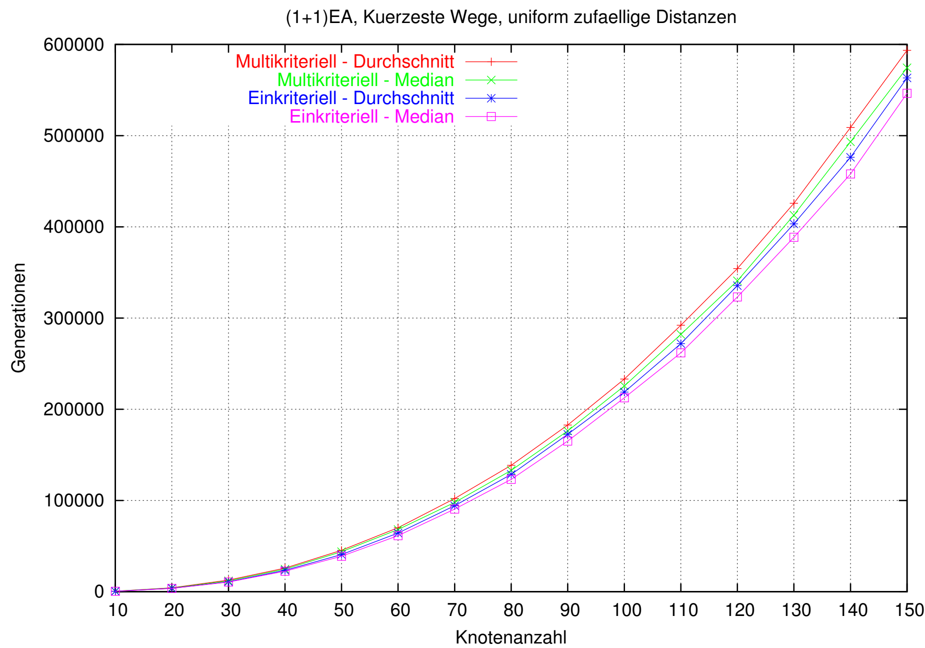


Abbildung 9.38: Laufzeiten des (1+1) EA auf kürzesten Wegen und uniform zufällige Distanzen (erste Klasse) mit den jeweils größten getesteten Kantengewichten.

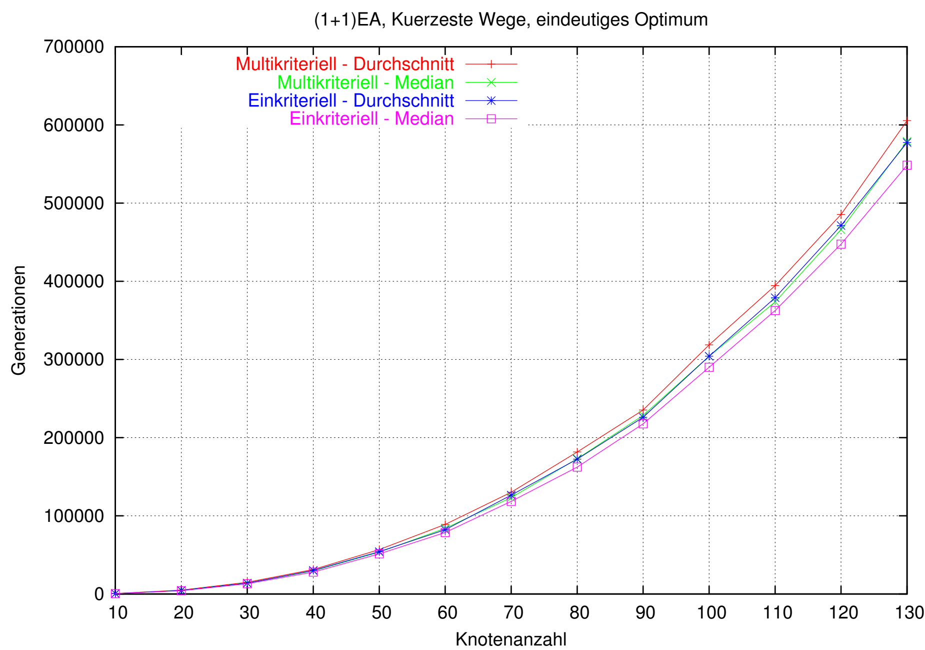


Abbildung 9.39: Laufzeiten des (1+1) EA auf kürzesten Wegen und eindeutigem Optimum (zweite Klasse) mit den jeweils größten getesteten Kantengewichten.

Das Ergebnis steht somit im Widerspruch zur anfänglichen Vermutung, dass der (1+1) EA auf der multikriteriellen Fitnessfunktion schneller ist, weil diese mehr Informationen liefert. Es lässt sich aber dadurch erklären, dass der (1+1) EA die ihm zur Verfügung gestellten Informationen im Grunde nicht ausnutzt. Werden in einem Schritt mehrere lokale Mutationen ausgeführt, so können sich in diesem Schritt einige Weglängen verkürzen und andere verlängern. Die Summe der Weglängen kann sich dabei aber trotzdem reduzieren, was dazu führt, dass dieser Schritt bei der einkriteriellen Fitnessfunktion akzeptiert wird. Sobald sich aber in einem Schritt ein Weg verlängert, wird das neue Individuum bei der multikriteriellen Fitnessfunktion auf jeden Fall verworfen, weil seine Fitness unvergleichbar oder schlechter als die des aktuellen Individuums ist. Umgekehrt wird aber jeder Schritt, der bei der multikriteriellen Fitnessfunktion akzeptiert wird, auch bei der einkriteriellen akzeptiert. Der Grund für die schnellere Laufzeit des (1+1) EA auf der einkriteriellen Fitnessfunktion muss also genau in den Schritten liegen, die einen Weg verlängern, aber insgesamt die Summe der Weglängen verkürzen.

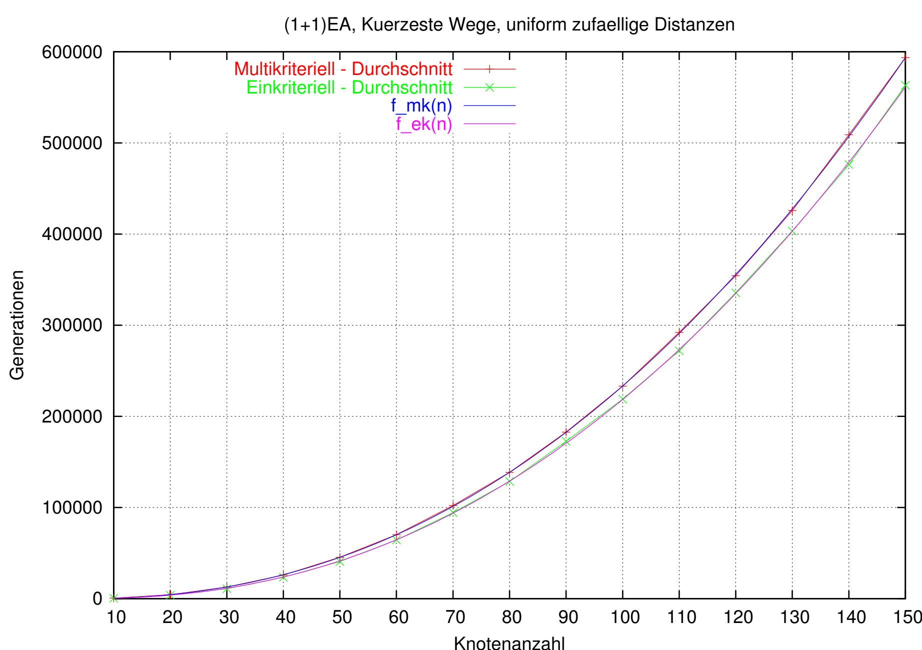


Abbildung 9.40: Durchschnittliche Laufzeiten des (1+1) EA auf kürzesten Wegen (uniform zufällige Distanzen) mit den jeweils größten getesteten Kantengewichten und den approximierten Funktionen.

Eine Regressionanalyse mit Gnuplot für die Laufzeiten auf uniform zufällige Distanzen (erste Klasse) mit den jeweils größten getesteten Kantengewichten der einkriteriellen und multikriteriellen Fitnessfunktion $f_{ek}(n)$ und $f_{mk}(n)$ ergab mit $n =$ Knotenzahl folgendes:

$f_{ek}(n) = 7{,}6446\ n^2 \log n - 13{,}3456\ n^2$

$f_{mk}(n) = 7{,}53736\ n^2 \log n - 11{,}3783\ n^2$

Die Summe der quadratischen Abweichungen beträgt für $f_{ek}(n)$ $1{,}75668 \cdot 10^7$ bzw. für $f_{mk}(n)$

$1{,}22978 \cdot 10^7$. Somit lassen sich die multikriteriellen Laufzeiten besser approximieren als die der einkriteriellen. Ein zusätzlicher Term von n^3 bekam durch die Regressionsanalyse nur einen Faktor von $-0{,}000396798$ bzw. $0{,}00291828$. Die Größenordnung von $n^2 \log n$ wird also in diesem Fall untermauert.

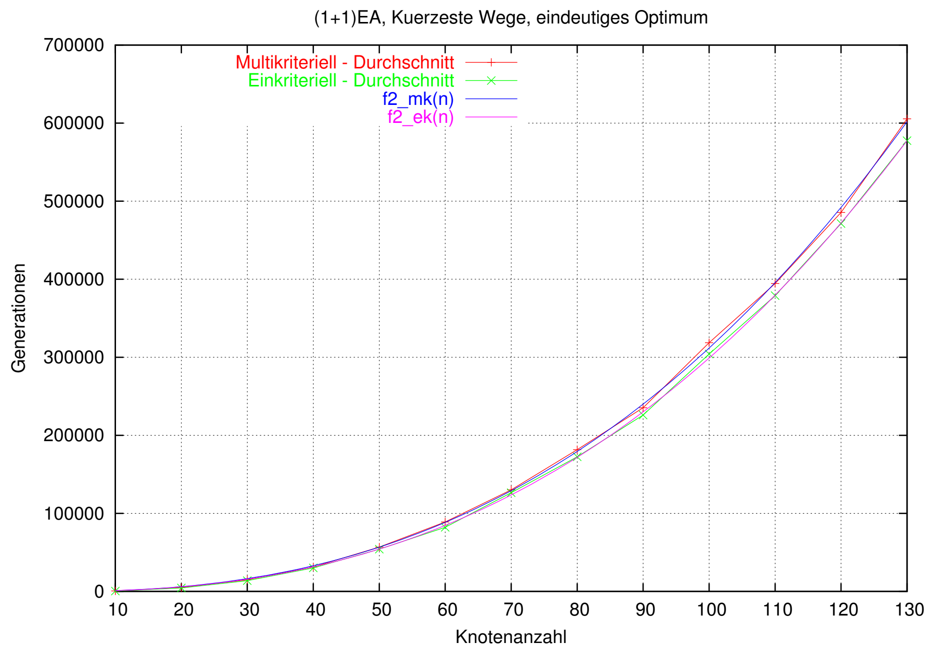


Abbildung 9.41: Durchschnittliche Laufzeiten des (1+1) EA auf kürzesten Wegen (eindeutiges Optimum) mit den jeweils größten getesteten Kantengewichten und den approximierten Funktionen.

Ganz anders sieht es mit der zweiten untersuchten Klasse mit eingebautem eindeutigen Optimum aus. Diese sind deutlich schlechter zu approximieren und der Term n^3 wird benötigt. Es ergeben sich für $f_{ek}^{(2)}(n)$ und $f_{mk}^{(2)}(n)$ ganz andere Funktionen:

$f_{ek}^{(2)}(n) = 0{,}105889\ n^3 + 4{,}19498\ n^2 \log n$ (Summe quadratischer Abweichungen: $6{,}59897 \cdot 10^7$)

$f_{mk}^{(2)}(n) = 0{,}108843\ n^3 + 4{,}40966\ n^2 \log n$ (Summe quadratischer Abweichungen: $1{,}37533 \cdot 10^8$)

Mit dem Term $n^2 \log n + n^2$ ergeben sich quadratische Abweichungen von $1{,}09413 \cdot 10^8$ bzw. $2{,}25735 \cdot 10^8$. Das Ergebnis ist also nicht ganz eindeutig, da sich in diesem Fall die Laufzeiten deutlich schlechter approximieren lassen.

Literaturverzeichnis

[1] G. Alder. *The home page of JGraph*. WWW page. `http://jgraph.sf.net/`.

[2] T. Bäck, D. Fogel und Z. Michalewicz (Hg.) (1997). *Handbook of Evolutionary Computation*. Institute of Physics Publishing and Oxford University Press.

[3] F. Barth und R. Haller (1996). *Stochastik Leistungskurs*. Ehrenwirth, 5. Auflage.

[4] C. Igel und M. Toussaint (2003). *On classes of functions for which no free lunch results hold*. In *Information Processing Letters 86*, S. 317–321.

[5] E. Cantú-Paz (2002). *On random numbers and the performance of genetic algorithms*. In W. B. Langdon, E. Cantú-Paz, K. Mathias et al. (Hg.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, S. 311–318. Morgan Kaufmann.

[6] E. Cantú-Paz und D. E. Goldberg (2003). *Are multiple runs of genetic algorithms better than one?*. In E. Cantú-Paz, J. A. Foster, K. Deb et al. (Hg.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003)*, Band 2723 von *Lecture Notes in Computer Science*, S. 801–812. Springer.

[7] K. de Jong und J. Sarma (1995). *On decentralizing selection algorithms*. In L. Eshelman (Hg.), *Proceedings of the Sixth International Conference on Genetic Algorithms*, S. 17–23. Morgan Kaufmann.

[8] M. Dietzfelbinger, B. Naudts, C. van Hoyweghen et al. (2003). *The analysis of a recombinative hill-climber on H-IFF*. In *IEEE Transactions on Evolutionary Computation*, 7: 417–423.

[9] S. Droste, T. Jansen, K. Tinnefeld et al. (2003). *A new framework for the valuation of algorithms for black-box optimization*. In *Foundations of Genetic Algorithms 7*, S. 253–270. Morgan Kaufmann.

[10] S. Fischer (2003). *Evolutionäre Algorithmen für verallgemeinerte Ising-Modelle*. Diplomarbeit, Universität Dortmund.

[11] S. Fischer und I. Wegener (2004). *The Ising model on the ring: Mutation versus recombination*. Eingereicht für GECCO 2004.

[12] A. S. Fukunaga (1998). *Restart scheduling for genetic algorithms*. In A. E. Eiben, T. Bäck, M. Schoenauer et al. (Hg.), *Parallel Problem Solving from Nature – PPSN V*,

Proceedings, Band 1498 von *Lecture Notes in Computer Science*, S. 357–366. Springer, Berlin.

[13] O. Giel (2003). *Expected runtimes of a simple multi-objective evolutionary algorithm.* In *Proceedings of the 2003 Congress on Evolutionary Computation (CEC 2003)*, S. 1918–1925.

[14] O. Giel und I. Wegener (2003). *Evolutionary algorithms and the maximum matching problem.* In *Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2003)*, Band 2607 von *Lecture Notes in Computer Science*, S. 415–426. Springer.

[15] D. E. Goldberg (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison–Wesley Longman, 1. Auflage.

[16] J. Grefenstette (1986). *Optimization of control parameters for genetic algorithms.* In *IEEE Transactions on Systems, Man and Cybernetics*, 16 (1): 122–128.

[17] S. G. Hohmann, J. Schemmel, F. Schürmann et al. (2002). *Exploring the parameter space of a genetic algorithm for training an analog neural network.* In W. B. Langdon, E. Cantú-Paz, K. Mathias et al. (Hg.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, S. 375–382. Morgan Kaufmann.

[18] W. Hoschek. *The colt distribution.* WWW page. `http://hoschek.home.cern.ch/hoschek/colt/`.

[19] T. Jansen (2002). *On the analysis of dynamic restart strategies for evolutionary algorithms.* In J. J. M. Guervós, P. Adamidis, H.-G. Beyer et al. (Hg.), *Parallel Problem Solving from Nature – PPSN VII, Proceedings*, Band 2439 von *Lecture Notes in Computer Science*, S. 33–43. Springer.

[20] T. Jansen und K. de Jong (2002). *An analysis of the role of offspring population size in EAs.* In W. B. Langdon, E. Cantú-Paz, K. E. Mathias et al. (Hg.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, S. 238–246. Morgan Kaufmann.

[21] T. Jansen und I. Wegener (2000). *On the choice of the mutation probability for the (1+1) EA.* In M. Schoenauer, K. Deb, G. Rudolph et al. (Hg.), *Parallel Problem Solving from Nature – PPSN VI, Proceedings*, Band 1917 von *Lecture Notes in Computer Science*, S. 89–98. Springer.

[22] T. Jansen und I. Wegener (2001). *Real royal road functions – functions where crossover provably is essential.* In L. Spector, E. D. Goodman, A. Wu et al. (Hg.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001)*, S. 375–382. Morgan Kaufmann.

[23] T. Jansen und I. Wegener (2002). *On the analysis of evolutionary algorithms – a proof that crossover really can help.* In *Algorithmica*, 34 (1): 47–66.

[24] J. D. Knowles und D. W. Corne (2000). *Approximating the nondominated front using the pareto archived evolution strategy.* In *Evolutionary Computation*, 8 (2): 149–172.

[25] M. Laumanns, L. Thiele, E. Zitzler et al. (2002). *Running time analysis of a multi-objective evolutionary algorithm on a simple discrete optimization problem*. In J. J. M. Guervós, P. Adamidis, H.-G. Beyer et al. (Hg.), *Parallel Problem Solving from Nature – PPSN VII, Proceedings*, Band 2439 von *Lecture Notes in Computer Science*, S. 44–53. Springer.

[26] S. Luke (2001). *When short runs beat long runs*. In L. Spector, E. D. Goodman, A. Wu et al. (Hg.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001)*, S. 74–80. Morgan Kaufmann.

[27] S. Luke und L. Panait (2002). *Is the perfect the enemy of the good?*. In W. B. Langdon, E. Cantú-Paz, K. Mathias et al. (Hg.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, S. 820–828. Morgan Kaufmann.

[28] Mark M. Meysenburg, Dan Hoelting, Duane McElvain et al. (2002). *How random generator quality impacts genetic algorithm performance*. In W. B. Langdon, E. Cantú-Paz, K. Mathias et al. (Hg.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, S. 480–483. Morgan Kaufmann.

[29] M. Matsumoto. *Mersenne twister home page*. WWW page. `http://www.math.keio.ac.jp/~matumoto/emt.html`.

[30] M. Matsumoto und T. Nishimura (1998). *Mersenne twister: A 623–dimensionally equidistributed uniform pseudorandom number generator*. In *ACM Transactions on Modeling and Computer Simulation*, Band 8, S. 3–30.

[31] M. M. Meysenburg und J. A. Foster (1999). *Randomness and GA performance, revisited*. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999)*, S. 425–432. Morgan Kaufmann.

[32] A. Mood, F. Graybill und D. Boes (1974). *Introduction to the Theory of Statistics*. Statistics Series. McGraw-Hill, 3. Auflage.

[33] F. Neumann und I. Wegener (2004). *Randomized local search, evolutionary algorithms, and the minimum spanning tree problem*. Eingereicht für GECCO 2004.

[34] G. Ochoa (2002). *Setting the mutation rate: Scope and limitations of the* $1/l$ *heuristic*. In W. B. Langdon, E. Cantú-Paz, K. Mathias et al. (Hg.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, S. 495–502. Morgan Kaufmann.

[35] R. Motwani und P. Raghavan (1995). *Randomized Algorithms*. Cambridge University Press, 1. Auflage.

[36] F. Rothlauf (2002). *Representations for Genetic and Evolutionary Algorithms*. Physica-Verlag, Heidelberg.

[37] A. Rummler. *eaLib – a Java evolutionary computation toolkit*. WWW page. `http://www.evolvica.org/ealib/index.html`.

[38] A. Rummler und G. Scarbata (2001). *Structuring evolutionary algorithms into components using the ealib framework*. In *5th IEEE International Conference on Intelligent Engineering Systems*, Band 5. IEEE.

[39] J. Scharnow, K. Tinnefeld und I. Wegener (2002). *Fitness landscapes based on sorting and shortest paths problems*. In J. J. M. Guervós, P. Adamidis, H.-G. Beyer et al. (Hg.), *Parallel Problem Solving from Nature – PPSN VII, Proceedings*, Band 2439 von *Lecture Notes in Computer Science*, S. 54–63. Springer.

[40] C. Schumacher, M. Vose und D. Whitley (2001). *The no free lunch and problem description length*. In L. Spector, E. D. Goodman, A. Wu et al. (Hg.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001)*, S. 565–570. Morgan Kaufmann.

[41] Sun Microsystems Inc. (2001). *Java look and feel design guidelines*. WWW page. `http://java.sun.com/products/jlf/`.

[42] The Java Software Human Interface Group. *Java look and feel graphics repository*. WWW page. `http://developer.java.sun.com/developer/techDocs/hi/repository/`.

[43] D. Thierens (2003). *Convergence time analysis for the multi-objective counting ones problem*. In C. M. Fonseca, P. J. Fleming, E. Zitzler et al. (Hg.), *Second International Conference on Evolutionary Multi–Criterion Optimization, Second International– EMO 2003, Proceedings*, Band 2632 von *Lecture Notes in Computer Science*. Springer.

[44] D. A. V. Veldhuizen und G. B. Lamont (2000). *Multiobjective evolutionary algorithms: Analyzing the state–of–the–art*. In *Evolutionary Computation*, 8 (2): 125–147.

[45] M. D. Vose (1999). *The Simple Genetic Algorithm: Foundations and Theory*. MIT Press.

[46] I. Wegener (2002). *Methods for the analysis of evolutionary algorithms on pseudo–boolean functions*. In R. Sarker, M. Mohammadian und X. Yao (Hg.), *Evolutionary Optimization*, Kapitel 14, S. 349–369. Kluwer Academic Publishers.

[47] I. Wegener und C. Witt (2004). *On the analysis of a simple evolutionary algorithm on quadratic pseudo-boolean functions*. Erscheint in Journal of Discrete Algorithms.

[48] I. Wegener und C. Witt (2004). *On the optimization of monotone polynomials by simple randomized search heuristics*. Erscheint in Combinatorics, Probability and Computing.

[49] D. Wolpert und W. Macready (1997). *No free lunch theorems for optimization*. In *IEEE Transactions on Evolutionary Computation*, 1: 67–82.