Metaheuristiken für die mehrkriterielle Optimierung

Endbericht der Projektgruppe 447

Universität Dortmund

27. April 2005

Teilnehmer:
Nicola Beume, Matthias Brinkmann, Torsten Lickfeld, Michael Janas, Ralf Kosse, Jens Kossek, Hendrik Noot, Miriam Padberg, Daniel Saltmann, Benedikt Schultebraucks, Kay Thielmann, Igor Tipura

Betreuer:
Thomas Bartz-Beielstein, Jörn Mehnen, Tim Richard, Karlheinz Schmitt

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Vorwort

Die Teilnahme an einer Projektgruppe ist Pflichtbestandteil der Studiengänge Informatik und Angewandte Informatik der Universität Dortmund. Im Zeitraum von zwei Semestern wird in einem Team von bis zu zwölf Studierenden eine komplexe Aufgabe bearbeitet. Die Aufgabe der PG-447 war es, eine plattformunabhängige Arbeitsumgebung für die Mehrzieloptimierung mit heuristischen Verfahren zu erstellen. Diese wurde dann u. a. an zwei praktischen Problemen von uns getestet.

Wir wollen uns bei den Menschen bedanken, die uns bei der Erfüllung dieser Aufgabe geholfen haben.
An erster Stelle wollen wir uns bei unseren Betreuern Thomas Bartz-Beielstein, Karlheinz Schmitt, Jörn Mehnen und Tim Richard bedanken, die uns mit Rat, Tat und Kritik zur Seite standen. Wir bedanken uns auch bei Michael Emmerich für seinen Vortrag über die Anwenderumgebung TOP (Tool for Optimizing Parameters). Diese diente uns als Richtungsweiser. Die Temperierbohrer sagen ein „Danke“ an Herrn Thomas Michelitsch für die Bereitstellung des Simulators „Evolver“ sowie für die Unterstützung und die Beantwortung zahlreicher Fragen. Die Metriker bedanken sich bei den Autoren des SMS-EMOA: Michael Emmerich, Nicola Beume und Boris Naujoks für die Bereitstellung des C-Quellcodes. Vielen Dank an Boris Naujoks, Karlheinz Schmitt und Klaus Friedrichs, die uns bei der Fehlersuche und bei der Klärung des Phänomens auf ZDT-Funktionen beratend und unterstützend zur Seite standen. Desweiteren danken wir dem Administrator Ulli Hermes für seine Unterstützung bei technischen Problemen. Wir bedanken uns bei allen, die an den Programmen mitgearbeitet haben, welche uns kostenlos zur Verfügung standen.

2 Einleitung

Diese Einleitung soll einen kurzen Einblick in die Problemstellung und den Ablauf der Projektgruppe 447 „Metaheuristiken für die mehrkriterielle Optimierung“ (PG447) geben.
Aufgabe der PG447 war es, innerhalb des Sommersemesters 2004 und des Wintersemesters 2004/2005 ein Softwareprodukt zu erstellen, das als Experimentier-Umgebung für die Mehrzieloptimierung mit heuristischen Verfahren dienen sollte. Diese Umgebung sollte objektorientiert implementiert werden und eine offene Schnittstelle zur Erweiterung zur Verfügung stellen. Außerdem sollten verschiedene Heuristiken, Bewertungsfunktionen, Visualisierungs-Tools und Testprobleme bzw. Anwendungsprobleme in dieser Umgebung integriert werden. Die Experimentier-Umgebung sollte durch die PG447-Teilnehmer entwickelt und angewendet werden, um mit den heuristischen Verfahren empirisch zu forschen und praxis-bezogene Ergebnisse zu erhalten.
Diese Arbeit wurde erfolgreich durchgeführt und ein Tool mit dem Namen *NObELTJE – N-Objective Evolutionary Learning-Tool in a Java Environment* erstellt, für das bereits im Sommersemester alle Grundfunktionen fertig gestellt wurden.
Das zweite Projekt-Semester diente zur Weiterentwicklung dieser Software und zur Erforschung von Meta-Heuristiken. *NObELTJE* ist unter der GPL Lizenz erhältlich, Details dazu im Kapitel Lizenz (siehe 6.3).
Im Folgendem wird auf die Gliederung des Gesamtdokuments (siehe 2.1) und den zeitlichen Ablauf der Projektgruppe (siehe 2.2) eingegangen.

2.1 Aufbau des Endberichts

In diesem Dokument werden die Funktionalität von *NObELTJE* und die erzielten Forschungs-Ergebnisse vorgestellt, über dessen Struktur nun ein Überblick gegeben wird. Nach der kurzen Einleitung (Kapitel 2), wird auf die Arbeitsorganisation (Kapitel 3) eingegangen, die einen Einblick in die Struktur einer Teamsitzung und der internen Kommunikation gibt.

Das Kapitel 4 beschäftigt sich mit der Softwareentwicklung. Es werden die Entwicklungstools, das Konzept, das die PG447 bei der Entwicklung von *NObELTJE* befolgte, und die Verifikation desselben beschrieben.

Das Softwareprodukt wird im Kapitel 5 erläutert. Die PG447 entwickelte im ersten Semester den größten Teil der Klassen und Pakete. Die PG lernte in der Seminarphase zu Anfang des ersten Semesters eine Reihe von heuristischen Verfahren (siehe Kapitel 5.1) kennen, die implementiert werden sollten, um auf Optimierungsprobleme angewendet zu werden. Aus diesen Heuristiken wurden im ersten Semester zwei evolutionäre Strategien, zwei Particle Swarm Optimization Ansätze, ein genetischer Algorithmus, ein Räuber-Beute-Algorithmus und ein Tabu Search-Ansatz ausgewählt und implementiert. Für die zweite Phase der Projektgruppe sollte der SMS-EMOA-Algorithmus implementiert und erweitert werden.
Um diese Algorithmen zu beurteilen, wurden Metriken (siehe Kapitel 5.2) benutzt, und zwar die Euklid-Metrik, die C-Metrik und die S-Metrik. Die PG versuchte weitere Metriken zu entwickeln. Neben den Metriken wurden weitere Analyse-Tools (siehe

Kapitel 5.3) zur Beurteilung der Ergebnisse der einzelnen Algorithmen (Heuristiken) umgesetzt, dazu sollten Statistik- bzw. Visualisations-Tools, die die Ergebnisse und deren Metrik-Werte darstellen, implementiert werden. Die Algorithmen wurden bereits im ersten Semester auf Testfunktionen (siehe Kapitel 5.4) angewendet, speziell wurde eine Funktion von Fonseca, das Knapsackproblem und die Double Parabola implementiert. Das zweite Semester sollte weitere Testfunktionen (die ZDT-Funktionen, die H-IV von Deb, und eine Schaffer-Funktion), bzw. die Anbindung an das Fahrstuhl- und Temperierbohrungsproblem hervorbringen.
Danach werden Tools und andere Hilfsklassen erläutert, die zur Darstellung von Ergebnissen und der Möglichkeit der Erstellung von Designs für Versuchspläne entwickelt wurden (siehe Kapitel 5.5).

Im Anschluss an das Software-Kapitel werden im Kapitel 6 die Benutzung der einzelnen Klassen und die Möglichkeit beschrieben, das Projekt um eigene Komponenten zu erweitern.

Die PG447 erschuf *NObELTJE*, um Meta-Heuristiken zu erforschen, Aussagen über deren Verhaltensweisen im Bezug auf das Temperierbohrungs- bzw. Fahrstuhlproblem, deren Robustheit und Darstellungsmöglichkeiten zu erhalten. Das Kapitel 7 ist all den Ergebnissen der Forschungen der PG447 gewidmet, die innerhalb des zweiten Semesters erfolgten.

Das Kapitel 8 gibt dem Leser einen Überblick über erreichte Ziele und Ergebnisse.

2.2 Zeitplan

Die PG447 arbeitete im Verlauf der beiden Semester nicht immer gemeinsam an einer Aufgabe mit einer Gruppenstärke von zwölf Personen. In diesem Teil soll kurz erläutert werden, wie die Gesamtgruppe in Kleingruppen aufgeteilt wurde, um einzelne Aufgaben, Forschungen und Entwicklungen durchzuführen. Die PG 447 arbeitete zusammen an dem Zwischenbericht, dem Endbericht und an dem Endvortrag.

2.2.1 Erstes Semester

Bereits im ersten Semester, dem Sommersemester 2004, wurde abhängig von der modularen Struktur des Projekts *NObELTJE* in folgenden Kleingruppen gearbeitet: „Algorithmen“, „Testfunktionen“, „Metriken-Analyse-Visualisierung“ (MetAnaVisu) und „Design-Schnittstelle“.
Die Gruppen arbeiteten selbständig, dokumentierten ihre Arbeit im *Wiki* (beschrieben in Kapitel 4.1) und berichteten über Fortschritte in den Teamsitzungen, dabei änderten die Kleingruppen-Mitglieder ihre Gruppenzugehörigkeit mehrmals. Die Tabelle 1 gibt einen Einblick in die benötigte Zeit der jeweiligen Kleingruppen in Kalenderwochen für die einzelnen Phasen der Entwicklung von *NObELTJE*.
Im Verlauf des ersten Semesters gab es mehrere Phasen, die die Kleingruppen durchlaufen haben. Auffällig ist z. B. der Verlauf der Kleingruppe Testfunktionen, der ähnlich zu anderen Kleingruppen ist. In den Kalenderwochen 19, 21 und 25 (siehe Tabelle 1) plante und suchte die Gruppe nach Testfunktionen, in den jeweiligen Wochen nach

der Planung wurden diese implementiert und am Ende des Semester noch getestet. Die Projektgruppe wechselte immer wieder, je nach Ansprüchen, die aus der Gruppe selbst an die jeweilige Teilgruppe gestellt wurden, ihren Modus von Plan- (P) und Implementierungs-Phasen (I). Die Tabelle 1 zeigt auch, dass in den Kalenderwochen $24 - 25$ neue Teilgruppen entstanden, die eine Auswahl von verschiedenen heuristischen Verfahren planten und implementierten.

Tabelle 1: Arbeitsaufwand der Kleingruppen im Sommersemester 2004

Kalenderwoche	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
Seminar	●														
Anforderung		●	●		●	●									
Use-Cases					●	●									
TestAlgo			P	I									T		
OnePlusOnePG					P	I						T			
MueRhoLES								P	I	IT	T				
PredatorPrey									P	I	IT	IT			
MopsoOne									P	I	I	IT	IT		
BasisGAPG									PI	I	I	I	IT		
TabuSearch									P	I	I	I	T		
Testfunktionen			PI	I	P	I	I	I	PI	I	T	T			
MetAnaVisu			P	I	PI	I	I	I	PI	I			T		
Schnittstellen						P	P	PI	I	I					
Design							P					PI	I	T	
Klassen-diagramme									●	●					
Zwischenbericht									●	●	●	●	●	●	●

Legende:
P=Planung; I=Implementierung; T=Tests mit JUnit

2.2.2 Zweites Semester

Aufgrund der positiven Erfahrung mit den Kleingruppen und deren Effektivität sollte im zweiten Semester, dem Wintersemester $2004/2005$, eine solche Einteilung wiederholt werden.
Zu Beginn wurde wieder ein Zeitplan erstellt. Die Mitglieder erkannten das dieser Plan nicht eingehalten werden konnte und erarbeiteten sich in der Kalenderwoche 44 (siehe Tabelle 2) Fragestellungen (F) für ihre jeweilige Teilgruppe. Es wurden folgende Teilgruppen gebildet: „Visualisierung“, „Robustheit“, „Metriken-Selektion“ und die „Praxisanwendungen“, die in die die Teilgruppen „Fahrstuhlproblem“ und „Temperierbohrungsproblem“ weiter unterteilt wurden. Die unten stehende Tabelle 2 stellt den Weg, den die Gruppen zur Lösung ihrer Fragen gegangen sind, im zeitlichem Ablauf dar. Dabei ist es sehr deutlich zu erkennen, dass ein dynamischer Wechsel von Experimentier- (Ex), Implementations- (I) und Auswertungs-Phasen (A) über das gesamte Semester stattfand.
Während die Kleingruppen ihre Experimente durchführten, wurden weitere Tools implementiert, Anpassungen der Simulatoren und Algorithmen und die Auswertung be-

reits beendeter Experimente vorgenommen. Dies kann man in der Tabelle 2 am Beispiel der Gruppe „Robustheit“ gut erkennen. Diese startete bereits in der KW 45 ihre ersten Experimente, da sie bereits in *NObELTJE* implementierte Algorithmen untersuchen wollte. Aufgrund der Ergebnisse dieser Experimente stellte sich die Gruppe neue speziellere Fragestellungen und implementierte weitere Analyse-Tools. Dieser Prozess wiederholte sich bis in die Kalenderwoche 02 (siehe Tabelle 2).

Tabelle 2: Arbeitsaufwand der Kleingruppen im Wintersemester 2004/2005

Kalenderwoche	**44**	**45**	**46**	**47**	**48**	**49**	**50**	**51**	**52**	**53**	**01**	**02**	**03**	**04**	**05**
Visualisierung	F	Ei	Ei I	I Ex	I Ex	I	I Ex	I Ex	I Ex						
Metriken-Selektion	F	Ei	I	I Ex	I Ex A	I Ex A	I Ex	I Ex A	Ex	Ex	Ex A	Ex A			
Robustheit	F	Ex I Ei A	Ex I F	Ex I A	Ex I A	Ex I A	Ex I	Ex I	Ex I	Ex I A	Ex I A	Ex I A			
Temperierbohrung	F	Ei I	Ex I	Ex A	Ex I A	Ex A	Ex	Ex	Ex	Ex	Ex A	Ex A			
Fahrstuhl	F	Ei	I	I Ex	Ex	I Ex	I Ex	Ex	I Ex A	Ex	Ex A	Ex A			
Endbericht											●	●	●	●	●
Endvortrag														●	●

Legende:
A=Auswertung der Experimente; Ei=Einarbeitung in die Teilgebiete; Ex=Experimente; F=Fragestellung für die Kleingruppe festgelegt; I=Implementierungen

3 Arbeitsorganisation

Um unserer Arbeit eine Richtung zu geben und auftretende Probleme und Fragen diskutieren zu können, fanden regelmäßige Treffen am Lehrstuhl 11 statt. Für diese Sitzungen gab es immer einen Protokollanten und einen Sitzungsleiter. Diese Aufgaben musste jeder Teilnehmer übernehmen, wobei man bei der Auswahl zyklisch nach dem Alphabet vorging. Das Protokoll sollte spätestens am nächsten Tag für alle einsehbar sein, um schellstmöglich allen Teilnehmern eine Übersicht über die gefassten Beschlüsse zu ermöglichen.

Die Aufgaben des Sitzungsleiters beinhalteten neben der Vorbereitung der Sitzung die Moderation der Versammlungen, die Leitung der Diskussionen sowie das Vorantreiben und Lenken der Gruppe im Sinne der Themen, damit das Projekt als solches sich stets weiter entwickeln konnte. Um dieses zu erreichen, wurde vereinbart, am Vortag die Tagesordnung mit den einzelnen Punkten, die in der kommenden Sitzung besprochen werden sollten, ins *Wiki* (siehe Kapitel 4.1) zu stellen. Jeder Teilnehmer hatte so die Möglichkeit, weitere Punkte der Tagesordnung bei Bedarf hinzuzufügen.

Bezüglich der Aufgaben der Protokollführung und Sitzungsleitung sollte auch der erzielte Lerneffekt nicht verschwiegen werden. Anfangs liefen die Sitzungen weniger geordnet ab, die Protokolle wiesen noch etliche Mängel auf. Es zeigte sich bei allen Teilnehmern eine deutliche Verbesserung bei der Wahrnehmung dieser Aufgaben im Laufe der PG.

Um die Kommunikation innerhalb der Sitzungen effizienter zu gestalten, haben wir ein paar „Kommunikationsregeln“ vereinbart, die im Folgenden kurz vorgestellt werden. Diese Regeln haben im wesentlich dazu beigetragen, dass Diskussionen sachlich und geordnet abliefen.

Einführung eines Kommunikationsballs In den Sitzungen wurde die Nutzung eines Balles eingeführt um eine eindeutige Regelung der Wortfolge zu gewährleisten. Nur derjenige, der den Ball in der Hand hielt, durfte reden.

Kurze Zusammenfassung der erzielten Ergebnisse Nach Beschlüssen oder nach Beendigung eines Tagesordnungspunktes hat der Sitzungsleiter gefasste Beschlüsse oder Ergebnisse einer Diskussion kurz zusammengefasst. So konnte der Protokollant die Kernaussage mit seinen Aufzeichnungen abgleichen und die einzelnen Sitzungsteilnehmer bekamen das endgültige Resultat noch einmal vor Augen geführt.

Wörterbuch Die PG beschäftigte sich mit einem großen Themenkomplex, in dem viele Fachbegriffe verwendet wurden. Um alle Teilnehmer auf einen Wissensstand zu bringen, haben wir im *Wiki* eine Seite erstellt, auf der ein Lexikon sowie eine Vokabelliste gepflegt wurden.

Nach jeder Sitzung gab es eine Feedback-Runde. Jeder Sitzungsteilnehmer konnte dann dem Sitzungsleiter mitteilen, welche Aspekte der Sitzungsleitung ihm gut gefielen, was nicht gut war und ihm generelle Verbesserungsvorschläge geben. Im Anschluss äußerte der Sitzungsleiter seine Meinung über das Verhalten der Gruppe und

den Verlauf der Sitzung.

Diese Sitzungen fanden im Plenum und während des ersten PG-Semesters zweimal wöchentlich statt und dauerten in der Regel zwei Stunden. Natürlich wurden Teilarbeiten an Kleingruppen delegiert, die sich neben diesen Versammlungen auch innerhalb ihrer Gruppe getroffen und abgesprochen haben. Die Ergebnisse der so bearbeiteten Aufgaben wurden zwischendurch und nach Abschluss der Arbeiten im Plenum vorgestellt und ggf. diskutiert.

Da zu Beginn des zweiten PG-Semesters eine besonders intensive Arbeit in auf Teilgebiete spezialisierten Kleingruppen erforderlich war, wurde zeitweise nur noch eine wöchentliche Gesamtsitzung durchgeführt, in der fast ausschließlich die Ergebnisse der Kleingruppen vorgestellt und geprüft wurden. So sollte den Fachgruppen mehr Raum für ihre Entwicklungs- und Forschungsarbeit gewährt werden.

Die Folge davon war jedoch, dass in der wöchentlichen Sitzung doch rasch zu viele Inhalte bearbeitet werden mussten, sodass kaum noch ein Austausch der Gruppen untereinander stattfinden konnte und die Kommunikation untereinander erheblich litt. Das führte dazu, dass (z. B. aufgrund des Zeitdrucks oder mangelnder Kenntnisse der Vorgänge in den anderen Teilgruppen) bei den Sitzungen Aufgaben nicht klar definiert oder verteilt und somit Arbeiten zum Teil unkoordiniert oder doppelt erledigt wurden.

Als der Gesprächsbedarf zu groß wurde und durch die wöchentliche Sitzung nicht mehr gedeckt werden konnte, wurde wieder auf das Modell des ersten Semesters mit zwei wöchentlichen Sitzungen gewechselt und somit dafür Sorge getragen, dass die Einzelergebnisse der Untergruppen sich wieder in Richtung eines Gesamtergebnisses der gesamten Gruppe entwickelten.

4 Softwareentwicklung

Dieses Kapitel beschreibt den von der PG447 eingeschlagenen Entwicklungsprozess. Es werden zunächst die vielen Tools und Programme vorgestellt, die wir in den beiden Semestern verwendet haben.
Danach folgt eine Erläuterung unserer Aufgabenstellung und dem daraus resultierenden Entwicklungsprozess. Am Schluss wird noch einmal gesondert auf die Verifikation unserer Software eingegangen.

4.1 Verwendete Tools

Um die Ziele der PG erreichen zu können, waren wir auf viele Tools angewiesen, die unsere Arbeit erst ermöglicht haben oder diese zumindest deutlich erleichterten. Hier werden nun die verschiedenen Werkzeuge, mit denen wir gearbeitet haben, kurz aufgeführt. In späteren Kapiteln wird auf einige dieser Tools noch näher eingegangen, wenn sie im unmittelbaren Zusammenhang zu den Ergebnissen der PG stehen.

Java Die gesamte Software, die im Rahmen dieser PG entstanden ist, wurde in der Sprache *Java* geschrieben. *Java* ist eine objektorientierte, plattformunabhängige Programmiersprache der Firma *Sun Microsystems*, die hauptsächlich unter der Führung von James Gosling entwickelt wurde.

CVS Um gemeinsam die Dateien zu verwalten und Änderungen für alle zugänglich zu machen, haben wir uns für das Tool *CVS* (*Concurrent Versions System*) entschieden. Dieses stellt eine zentrale Sammelstelle für die unterschiedlichen Dateien dar. Vorteilhaft an diesem Tool ist, dass eine Versionskontrolle integriert ist, so dass ältere Versionen einer Datei jederzeit wieder übernommen werden können. Zudem bietet das *CVS* die Möglichkeit, Dateien gleichzeitig bearbeiten zu können; falls Konflikte auftreten, können diese mittels *CVS* gelöst werden.
Dieses Tool hat sich bei unserer PG als sehr geeignet erwiesen. Im *CVS* haben wir nicht nur unseren Programmcode verwaltet, sondern auch die Seminarausarbeitungen, den Zwischenbericht sowie die Texte zu diesem Endbericht.
Die Nachteile von *CVS*, wie beispielsweise die ineffiziente Speicherung von Binärdateien und Probleme beim Verschieben und Umbenennen von Dateien, haben die PG nicht stark in ihrer Arbeit beeinflusst. Jedoch sollte dies bedacht werden, insbesondere bei Projekten, die ein häufiges Refactoring durchlaufen.

Wiki Auf der Suche nach einem geeigneten Kommunikationstool haben wir uns, analog zu der Vorgänger-PG 431 MooN, für das Tool *Wiki* entschieden. Wir haben dafür ein *UseMod-Wiki* auf einem externen Server aufgesetzt, welches sich insbesondere durch seine Zuverlässigkeit während der gesamt Zeit auszeichnete.
Durch Beamer und Laptop hatten wir in den meisten Teamsitzungen auch direkten Zugriff auf das *Wiki*. Dieses unterstützte uns in vielen Bereichen. Einige Beispiele:

- Die Tagesordnung zu jeder Sitzung wurde in das *Wiki* gestellt. Dadurch konnte jeder Teilnehmer sowohl problemlos Anmerkungen notieren, als auch die Tagesordnung direkt ergänzen oder umstellen. Nach den Sitzungen wurde das Protokoll im *Wiki* veröffentlicht.

- Termine für Treffen in den Kleingruppen wurden größtenteils über das *Wiki* abgesprochen.
- Das *Wiki* diente als Sammelort für alle relevanten Dinge wie aktuelle Links, Informationen zu verschiedenen Tools usw. Weiterhin wurden wichtige Daten, wie z. B. eine detaillierte Beschreibung zu den Fortschritten in den einzelnen Gruppen, ins *Wiki* gestellt, so dass alle Teilnehmer sich zu jeder Zeit über den Stand der Arbeit in den Kleingruppen informieren konnten.
- Die Sammlung von Fehlern und auch die Liste, wer gerade woran arbeitet, hat sich als sehr vorteilhaft erwiesen und half bei der Koordination, vor allem in der Implementierungs-Phase.
- Insbesondere im zweiten Semester wurden Ergebnisse der Kleingruppen mit Hilfe des *Wiki*s präsentiert und Zeitpläne und Aufgabenzuordnungen direkt im *Wiki* festgehalten.

Insgesamt stellte sich das *Wiki* als generelle Kommunikationsplattform als sehr vorteilhaft heraus.

Forum Zu Beginn der PG wurde ein Forum gewünscht, in dem man aktuelle Probleme diskutieren und besprechen könnte. Es stellte sich jedoch während des ersten Semesters heraus, dass dieses Forum nur anfangs häufig benutzt wurde. Dies änderte sich auch im zweiten Semester nicht.

Latex Für die Dokumentation und andere wichtige Schriftstücke, wie auch für diesen Endbericht, haben wir uns geeinigt, *pdflatex* zu verwenden. Dadurch wird ermöglicht, dass umfangreiche Dokumente in Modulen geschrieben und anschließend zu einem Dokument zusammengefügt werden können.
Anfangs gab es bei einigen Teilnehmern Schwierigkeiten, da LaTeX eine doch sehr eigene Syntax darstellt. Vorteilhaft ist, dass unabhängig vom Betriebssystem gearbeitet werden konnte. Zudem bietet LaTeX sehr viele Möglichkeiten, wie z. B. eine gute Darstellung von komplexen Formeln.

Eclipse *Eclipse* ist ein Framework für Programmierumgebungen mit einem Plugin für die *Java*-Entwicklung, welches die Implementierung vereinfacht. *Eclipse* zeigt Syntaxfehler an und hilft damit, Fehler in der Implementierung zu verhindern. Ein weiterer Vorteil von *Eclipse* ist die komplette *CVS*-Integration, mit der man ein *CVS*-Projekt verwalten kann. Da *Eclipse* kostenlos erhältlich ist und auf vielen Plattformen läuft, haben wir uns auf dieses Tool geeinigt. Weitere Informationen zu *Eclipse* kann man unter [2] erhalten.

JUDE Im Gegensatz zu dem Entwicklungstool *Together* bietet *Eclipse* leider keine Möglichkeit, ohne ein entsprechendes Plugin, UML-Diagramme zu erstellen. Von daher wurde nach einer betriebssystemunabhängigen, kostengünstigen Alternative zum Erstellen von UML-Diagrammen gesucht. Entschieden haben wir uns für das Tool *JUDE*, welches einfach zu bedienen ist und die wesentlichen Funktionalitäten unterstützt. Ein Nachteil ist jedoch die fehlende Exportfunktion für Vektorgrafiken. Mehr Informationen zu diesem Programm erhält man unter [3].

Gnuplot Um die Ergebnisse unserer Implementierung auch grafisch darzustellen, haben wir das Tool *Gnuplot* benutzt. Zunächst mit manuellen Aufrufen, die jedoch später weitgehend automatisiert wurden. Viele unserer Visualisierungstools, wie der `BohrVisualizer` 5.3.7, verwenden *Gnuplot* zur Onlinedarstellung oder Erzeugung von Bilddateien. Eine offizielle *Gnuplot*-Dokumentation ist unter [4] zu finden.

JUnit *JUnit* ist ein Framework für Klassentests. Mit diesem haben wir für alle wichtigen Klassen-Funktionstests erstellt.

GGobi Zur Visualisierung mehrdimensionaler Daten haben wir ab dem zweiten Semester auch *GGobi* eingesetzt. Dies ist eine Weiterentwicklung von *XGobi*, die es ermöglicht verschiedene Darstellungen der Daten gleichzeitig anzuzeigen.

R *R* ist eine Sprache und eine Statistikumgebung. Es kann als Open-Source-Implementierung der Statistiksoftware *S* gesehen werden, welche zur Standardsoftware der Statistik gehört. Mit *R* kann man statistische Auswertungen und Visualisierungen realisieren. Man spricht von einer Umgebung bei *R*, da sich diese aus vielen Programmen und Plugins zusammensetzt. Im Laufe der PG sind einige Tools entstanden, die *R* einsetzen, um die erzeugten Daten auszuwerten. Viele der Plots in diesem Endbericht, beispielsweise die Plots im Kapitel 7.3.2, entstanden mit *R*.

4.2 Implementierungskonzept

Die Entwicklung von *NObELTJE* stellte die PG vor einige Herausforderungen. Dies lag insbesondere an der relativ freien Aufgabenstellung. Unser Ziel war es, eine „Experimentier-Umgebung für die Mehrzieloptimierung mit heuristischen Verfahren“ zu erstellen. Da es kein Pflichtenheft oder eine vorgegebene Anforderungsdefinition gab, lag es an uns, diese Anforderungen genau zu definieren. Dies ist der Grund, weshalb wir nicht die klassischen Verfahren der objektorientierten Softwareentwicklung anwenden konnten. Denn diese Verfahren zielen darauf ab, ein Design für die Entwicklung zu erstellen. Dieses basiert auf den Erkenntnissen einer Analysephase. Eine klassische Analyse besteht jedoch insbesondere daraus, einen Ist- und einen Sollzustand zu ermitteln. Da die PG ein neues Projekt ohne bekannte Nutzergruppe erstellen sollte, fehlte der Istzustand. Für den Sollzustand wurden von den Betreuern keine strikten Vorgaben gemacht, wodurch man der PG, auch für ihre eigene Forschung, viel Freiraum ließ. Da die Teilnehmer zwar einiges an Fachwissen hatten, aber nur wenig Praxiserfahrung im Umgang mit Mehrzieloptimierung und Metaheuristiken, wurde zunächst ein Konzept erstellt.

Die erste Frage war, wer diese Software nutzen soll und welche Aktivitäten diese Anwender mit der Software durchführen können sollten. Im Wesentlichen sind dies für unser Projekt der normale Anwender, der das Tool zur Optimierung von Mehrzielproblemen verwendet, und der Forscher/Wissenschaftler, der insbesondere an den Algorithmen interessiert ist. Das Use-Case Diagramm für den Anwender (siehe Abbildung 1) soll exemplarisch die Möglichkeiten dieses Anderwendertyps darstellen.

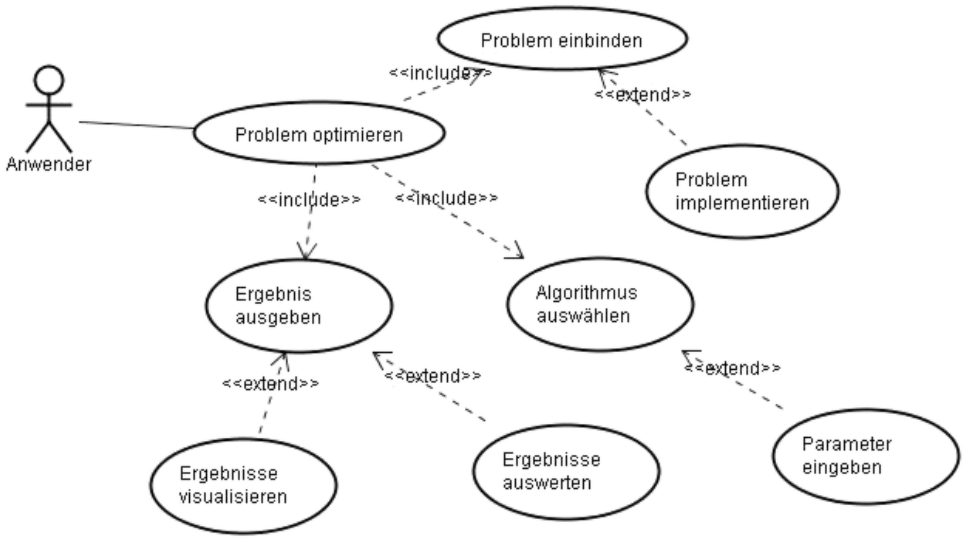

Abbildung 1: Anwendungsfälle des Akteurs „Anwender“

Durch die Identifizierung dieser Anwendertypen wurde sehr schnell klar, dass unser Programm in hohem Maße erweiterbar sein muss. Ein potentieller Anwender würde in erster Linie sein eigenes Optimierungsproblem einbinden wollen und Wissenschaftler ihre eigenen Algorithmen und Metriken.

Hier wurden dann auch die drei Hauptelemente der zu entwickelnden Software deutlich: Algorithmen, Optimierungsprobleme und Metriken. Die Betreuer begrüßten eine so aufgebaute Struktur und motivierten die Gruppe, dass Projekt so umzusetzten. An dieser Stelle beschloss die Gruppe, zunächst einen modularen Prototypen zu erstellen, um Erfahrungen zu sammeln, die beim Entwurf der Struktur helfen sollten. Es wurde zunächst ein Algorithmus mit einer Testfunktion und einer Metrik implementiert.

Der Aufbau dieses Prototyps war dabei schon so gut, dass die PG die Schnittstellen zwischen den Modulen nur noch minimal anpassen und in einer Spezifikation festhalten musste. Dies geschah unter anderem durch UML Klassen-, Sequenz- und Kommunikationsdiagramme, wodurch die Grundlage der weiteren Entwicklung gebildet wurde. Sie enthielten auch die noch fehlenden Module wie Design und Analyse. Allerdings wurden die Anforderungen im Verlaufe der PG noch häufig erweitert, wodurch die damalige Spezifikation nur einen Teil der finalen Spezifikation darstellte.

Nach dieser Planungs- und Prototypenphase wurden zunächst der Prototyp erweitert, um die Basis für die weitere Arbeit zu bilden. Diese bestand dann im ersten Semester hauptsächlich in der Erweiterung der Module. Es wurden in Kleingruppen weitere Algoritmen, Metriken und Testfunktionen implementiert, wie auch Tools in den neuen Modulen, wie beispielsweise den Analysetools. Die Schnittstellen sind dabei nicht auf reellwertige Probleme beschränkt worden, was durch die Einbindung eines kombinatorischen Problems demonstriert wurde. Der modulare Aufbau hat sich auch dadurch bewährt, dass die einzelnen Gruppen meist unabhängig an ihren Teilen arbeiten konn-

ten, ohne dadurch die weiteren Funktionalitäten des Gesamtprojekts zu beeinflussen.

4.3 Verifikation der Software

Der von uns gewählte Entwicklungsprozess erforderte eine kontinuierliche Qualitäts- und Funktionskontrolle. Anders als bei klassischen modellorientierten Entwicklungsmethoden gab es keine vorgegebene Schnittstellendefinition für jede Klasse. Im Laufe der Entwicklung wurden viele Klassen mehrfach modifiziert und erweitert. Es galt also sicherzustellen, dass jede Klasse ihre Anforderungen erfüllte, sowohl die bisherigen als auch die erweiterten. Dazu benötigten wir für alle Basisklassen entsprechende Klassentests.
Ein Framework zur Erstellung von Klassentests in *Java* ist *JUnit*, welches wir für unser Projekt verwendet haben. Es stellt Methoden zum Test von Klassen bereit und erleichtert die Entwicklung der Tests. Der Start aller dieser Tests erfolgt durch eine Methode von *JUnit*, welche die Ergebnisse dann ausgiebt. Die meisten unserer Tests wurden als „White-Box-Test“ geschrieben. Solche Tests werden erst nach Erstellung der zu testenden Klassen und von den selben Entwicklern geschrieben werden. Besser wäre es gewesen, die Tests gemäß dem Test-First Prinzip zuerst zu erstellen. Dies wären dann sogenannte „Grey-Box-Tests“, welche im Allgemeinen besser sind. Allerdings war dies bei vielen Klassen nicht möglich, da ein großer Teil der Basisklassen schon zu Beginn der PG entstanden ist, als das ganze noch als kurze Experimentierphase galt. Die Einschränkungen von Klassentests wurden den Teilnehmern allerdings schnell bei der Verifikation der Algorithmen klar. Insbesondere bei der Verwendung von Zufallszahlen wird es praktisch unmöglich, die korrekte Funktion, selbst bei speziellen Testfällen, zu überprüfen. Die Algorithmen mussten deshalb größtenteils manuell getestet werden. Unter anderem durch manuelle Überprüfung des Quellcodes von verschiedenen Teilnehmern, Ablaufkontrollen der Algorithmen im Debugger sowie Plausibilitätskontrollen der Ergebnisse.

5 Struktur des Software-Projekts

NObELTJE ist ein kommandozeilenbasiertes Framework, dass sich aufgrund des modularen Aufbaus leicht für eigene Experimente erweitern oder anpassen lässt. Die Hauptmodule sind die Algorithmen, die Testfunktionen bzw. reellen Anwendungsprobleme (Simulatoren), die Metriken und die Analyse/Visualisierung. Details zum Zusammenspiel und zur Anwendung werden im Kapitel 6 erläutert.

Wie bereits in Kapitel 3 erwähnt, wurde viel in Kleingruppen gearbeitet, die sich auf bestimmte Module spezialisiert haben. Natürlich wurde daher das Projekt entsprechend der unterschiedlichen Themen strukturiert. Wie aus Abbildung 2 ersichtlich, finden sich neben den zu *Java*-Paketen zusammengefassten Klassen eines Moduls noch weitere Pakete. Als Wichtigstes davon sei das Package `design` erwähnt: Mit der enthaltenen Klasse kann ein komplettes Versuchs-Design mit variierenden Parametern gestartet werden. Sie ist ein mächtiges Werkzeug und ein wichtiges Hilfsmittel, das im Zusammenspiel mit den unter `analysis` befindlichen Tools die Durchführung umfangreicher Experimente mit wenig „Handarbeit“ ermöglicht. Die restlichen Pakete enthalten nützliche Werkzeuge, die von den Klassen der Hauptmodule verwendet werden oder aber die Arbeit mit *NObELTJE* weiter vereinfachen.

5.1 Algorithmen

Zur Analyse und metaheuristischen Optimierung haben wir verschiedene Algorithmen betrachtet und diese in *NObELTJE* integriert.
Dieses Kapitel soll einen Überblick geben, welche Algorithmen zur Verfügung stehen und in welchen Varianten diese implementiert wurden. Zunächst werden das generelle Vorgehen sowie die allgemeinen Einstellungen erläutert. Anschließend werden bei jedem Algorithmus auf die Schwachstellen und die Verbesserungsmöglichkeiten eingegangen.

Gemäß dem in Kapitel 4.2 beschriebenen Implementierungskonzept wurde mit dem `TestAlgo` zunächst ein rudimentärer Algorithmus implementiert, der einen variablen Aufruf einer beliebigen Testfunktion unterstützte. Um eine Auswahl an Algorithmen zur Verfügung zu haben, aus der für Vergleiche gewählt werden kann, wurden in Kleingruppen insgesamt sieben weitere Algorithmen implementiert. Dabei sollten diverse Klassen von Heuristiken abgedeckt werden: Die Evolutiosstrategien (`MueRhoLES`), die Genetischen Algorithmen (`BasisGAPG`), die Gruppe der Particle Swarm Optimization-Algorithmen (`BasisPSO` und `MopsoOne`), die Räuber-Beute-Algorithmen sowie Tabu Search mit einem entsprechenden kombinatorischen Algorithmus für das Rucksack-Problem. Im zweiten PG-Semester wurde der `SMS_EMOA_pg` entwickelt, der eine Metrik als Selektionskriterium verwendet.

Durch die redundanten Operationen, die jeder Algorithmus benötigt und eine gewollte Standardisierung der Bedienung wurde zu den Algorithmen ein übersichtliches Interface und eine umfangreichere abstrakte Klasse `AbstractAlgo` geschrieben. Diese übernimmt das Einlesen aller benötigten Kommandozeilenparameter und einer optionalen Parameterdatei, sowie die Anbindung der Testfunktion und das Bereitstellen ei-

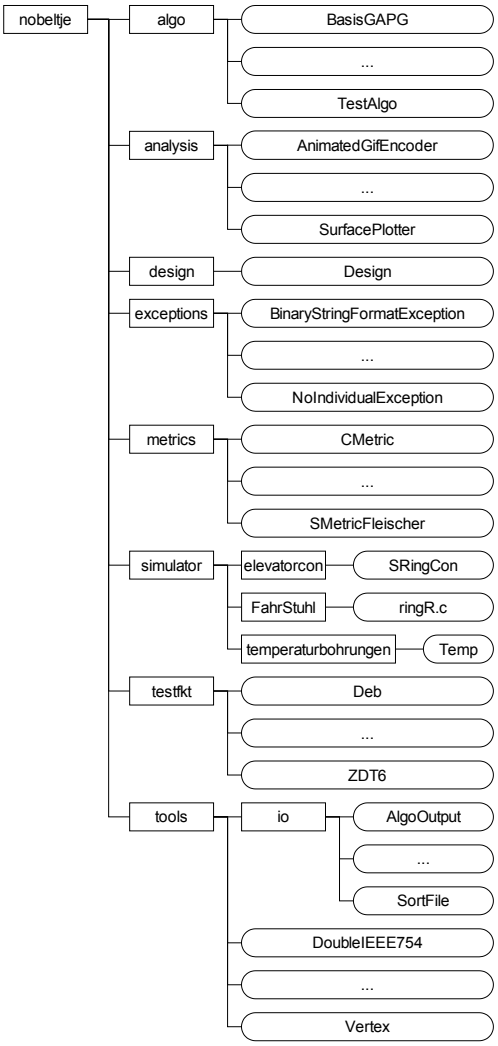

Abbildung 2: Übersicht über die *NObELTJE*-Paketstruktur. Hauptmodule sind `algo`, `analysis`, `metrics` sowie `simulator` und `testfkt`.

nes Objekts zur Generierung von Zufallszahlen.

Durch die zentrale Bereitstellung dieser Funktionen benötigen alle implementierten Algorithmen folgende Parameter:

Testfunktion: Dem Algorithmus muss beim Aufruf mittels Parameter `-t` die Klasse der zu optimierenden Funktion angegeben werden. Diese wird mit vollem Paket- und Klassenname angegeben.
Beispiel: `-t testfkt.FonsecaF1`

Anzahl der Generationen / Iterationen: Jeder Algorithmus hat zumindest die maximale Anzahl an Generationen oder Iterationen als Abbruchkriterium, welche durch den Parameter `-g` angegeben wird.
Beispiel: `-g 1000`

Zusätzlich verfügen alle Algorithmen über weitere **optionale** Parameter:

Parameterdatei: Neben der Eingabe der Parameter über die Kommandozeile ist es auch möglich, diese aus einer Datei einlesen zu lassen. Diese wird mit der Option `-c` angegeben und muss die Dateiendung „`.param`“ haben (welche bei der Angabe der Datei nicht zu berücksichtigen ist).
Wird ein Parameter sowohl in der Datei als auch über die Kommandozeile gesetzt, so wird der Wert, der in der Kommandozeile angegeben wurde, verwendet. Somit ist es möglich, Standardwerte in einer Datei festzuhalten und beim Aufruf von einzelnen abzuweichen.
Beispiel: `-c algo/OnePlusOnePG`

Random-Seed: Um reproduzierbare Läufe der teils randomisierten Algorithmen zu erhalten kann eine Initialisierung der Zufallszahlen mittels Parameter `-r` erfolgen. Bei gleicher Initialisierung werden die gleichen „Zufallszahlen“ erzeugt und die Algorithmen laufen praktisch deterministisch ab.
Beispiel: `-r 1001`

Präfix für die Dateinamen der Ausgabe: Die Algorithmen produzieren diverse Ausgabedateien. Um die Ausgabearten (Objektvariablen, Funktionswerte der Individuen, ...) eines Programmlaufs trennen zu können, erhalten die Dateien für die verschiedenen Ausgabetypen jeweils entsprechende Dateiendungen (Suffixe). Durch unterschiedliche Präfixe der Dateinamen kann auch zwischen verschiedenen Läufen der Algorithmen unterschieden werden. Diese können mit dem optionalen Parameter `-o` manuell gesetzt werden. Erfolgt dies nicht, verwenden die Algorithmen automatisch eine Kombination aus akutuellem Datum und Uhrzeit als Präfix.
Beispiel: `-o ErsterLauf`

Ausführlichere Ausgaben: Die Ausgaben der Algorithmen beschränken sich meist auf das Erzeugen von Ausgabedateien und umfassen wenig oder gar keine Angaben über den internen Ablauf. Mit dem Parameter `-d` können auch solche Informationen aktiviert werden, die meist auf der Konsole erscheinen. Dieser Parameter benötigt keine weiteren Angaben.
Beispiel: `-d`

Im Folgenden werden nun alle implementierten Algorithmen vorgestellt.

5.1.1 Test-Algorithmus

Dieser Algorithmus ist von der Implementierung her sehr einfach. Er generiert zufällig ein Individuum. Aufgrund der Tatsache, dass ein mehrkriterieller Algorithmus für Minimierungsprobleme implementiert werden sollte, haben wir für jedes Individuum nicht nur einen Fitnesswert, sondern einen Vektor von Funktionswerten. Um diesen Vektor jedoch als Fitnesswert zu beschreiben, summiert der Algorithmus die einzelnen Funktionswerte einfach auf und die Gesamtsumme wird als Fitness betrachtet.
Des Weiteren arbeitet der Algorithmus mit einem Archiv, in dem die Elemente eingefügt werden, die den kleinsten Fitnesswert erreicht haben. Da die Fitnesswerte durch die Transformation eindeutig geordnet sind, beinhaltet das Archiv nur Elemente mit dem gleichen Fitnesswert.

5.1.1.1 Der Algorithmus

Der Algorithmus benötigt beim Aufruf einige Angaben: Welche Testfunktion soll optimiert werden, wie viele Iterationen sollen durchgeführt werden und wie groß soll das Archiv maximal werden. Die Dimension des Entscheidungsraums wird bei der Testfunktion erfragt. In jeder Iteration wird ein Individuum mit genau dieser Anzahl von Objektvariablen randomisiert erzeugt. Nach diesem Schritt werden die Funktionswerte bei der Testfunktion erfragt und der Fitnesswert als Summe über die Funktionswerte ermittelt. Anschließend wird überprüft, ob das erzeugte Individuum einen besseren Fitnesswert hat als die Individuen, die bereits im Archiv vorgehalten werden. Falls dieses der Fall ist, wird das Archiv gelöscht und das Individuum in das Archiv eingefügt, bei gleichen Fitnesswerten wird das Archiv um das erzeugte Individuum erweitert.

Das Ablaufdiagramm (siehe Abbildung 3) soll den Algorithmus verdeutlichen.

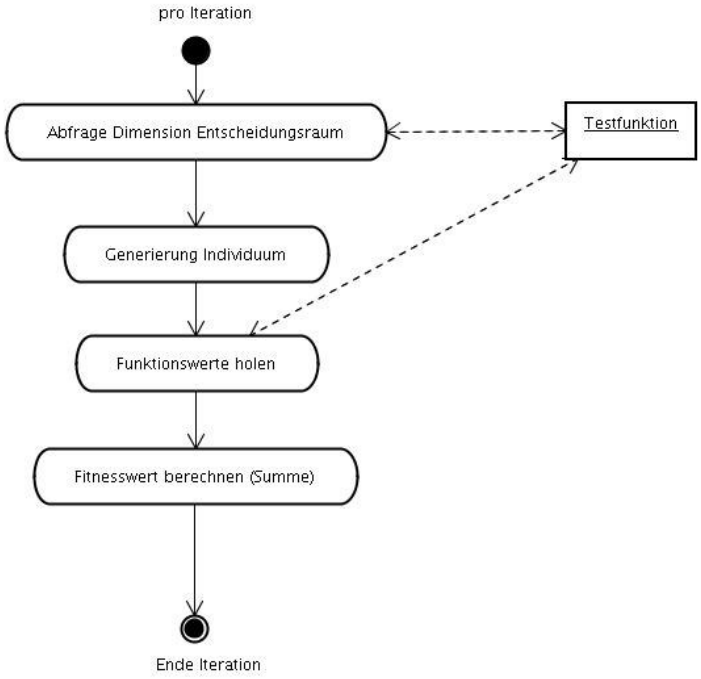

Abbildung 3: Ablaufdiagramm einer Iteration des Test-Algorithmus.

5.1.1.2 Einstellmöglichkeiten

Es gibt als einzige Einstellmöglichkeit die Größe des Archivs. Einige Testdurchläufe mit Testfunktionen, die konvexe Pareto-Fronten haben, zeigten, dass am Ende nur ein Element im Archiv enthalten ist. Aufgrund der Transformation ist die Wahrscheinlichkeit, dass am Ende viele Elemente im Archiv sind, sehr gering. Von daher ist die Archivgröße nicht so relevant.

5.1.1.3 Schwachstellen

Wie eingangs erwähnt, diente der `TestAlgo` nur zur Gewinnung eines groben Überblicks über die einzelnen Abläufe zwischen den Modulen. Das Prinzip der Aufsummierung der Funktionswerte, um einen Fitnesswert zu erhalten, ist ein Verfahren, um ein mehrkriterielles Problem in ein einkriterielles Problem zu transformieren. Dadurch gibt es die bekannten Probleme, wie sie auch in der Seminarausarbeitung zum Thema „Konventionelle Verfahren“ beschrieben werden. Diese Seminarausarbeiten sind am Lehrstuhl 11 bei den PG-Betreuern einsehbar.
Weitere Probleme ergeben sich dadurch, dass die Individuen randomisiert erzeugt werden, und somit kein Einfluss darauf ausgeübt werden kann, wie die Individuen sich entwickeln. Falls ein recht gutes Individuum erzeugt wurde, wird z. B. nicht in der Umgebung dieses Individuums weiter gesucht, sondern in der nächsten Iteration vollkommen unabhängig davon einfach ein neues Individuum generiert.
Eine ganz große Schwachstelle ist die Berechnung der Fitness. Falls die Funktionswerte o. B. d. A. des ersten Kriteriums nur kleine Werte liefern, die des zweiten Kriteriums jedoch recht große Funktionswerte, so hat das erste Kriterium nur recht wenig Einfluss auf die Fitness. Dadurch, dass die Summe über die Funktionswerte als Fitnesswert genommen wird und folglich nur die Elemente in das Archiv kommen, die den kleinsten gefundenen Fitnesswert haben, wird bei solchen Funktionen nach einem Element gesucht, das in dem zweiten Kriterium möglichst kleine Funktionswerte erreicht.

5.1.1.4 Verbesserungsmöglichkeiten

Eine einfache Verbesserungsmöglichkeit liegt in der Bewertung des Algorithmus. Statt die Funktionswerte aufzusummieren, könnte man die Archivbewertung auch nach Pareto-Dominanz vornehmen. Dadurch garantiert man zumindest, dass die Elemente nach Kriterien beurteilt werden und es werden Elemente gesucht, die minimale Funktionswerte in allen Kriterien erreichen.
Da dieser Algorithmus jedoch nur zur Veranschaulichung diente, haben wir ihn im zweiten PG-Semester nicht weiter verändert.

5.1.2 One-Plus-One Algorithmus

Nach dem ersten einfachen `TestAlgo` haben wir einen einfachen evolutionären Algorithmus implementiert, der insbesondere die Pareto-Optimalität verwendet.
Daher wurde ein simpler (1+1)-EA mit einem Archiv pareto-optimaler Elemente ausgewählt. In einem solchen Archiv sind nur Individuen, die keinem anderen in allen Funktionswerten unterlegen sind. Die Größe des Archivs ist per Parameter wählbar und bestimmt durch die rechenaufwändige Berechnung der nicht dominierten Individuen die Laufzeit des Algorithmus entscheidend mit.

5.1.2.1 Der Algorithmus

Bei einem (1+1)-EA besteht eine Generation nur aus einem Individuum, aus dem für die Generation der Nachkommen auch nur ein neues durch Reproduktion und Mutation generiert wird. Somit muss nur zwischen diesen beiden selektiert werden.

Das erste Individuum wird mit zufällig generierten Doublewerten initialisiert, die innerhalb der Grenzen liegen, die die Testfunktion liefert.

Reproduktion: Um die Nachkommen-Generation von evolutionären Strategien zu bilden, müssen aus Individuen der Elter-Generation Nachkommen rekombiniert werden. Da bei diesem Algorithmus die Generationen aber nur aus einem Individuum bestehen, genügt zur Rekombination eine einfache Reproduktion, d.h. das Elter-Individuum wird einfach kopiert.

Mutation: Zur Mutation wird auf jede Objektvariable des Individuums ein gaußverteilter Wert addiert, der vorher mit der Mutationsschrittweite multipliziert wurde.
Die Stärke der Mutation hängt somit von der exogen bestimmten statischen Varianz, der Mutationsschrittweite, ab. Bei einer zu kleinen Schrittweite verbessern oder verschlechtern sich die Individuen nur gering und eine erste Annäherung an die Pareto-Front dauert entsprechend lange. Ist der Wert zu groß gewählt, wird die feine Bestimmung, also die präzise Annäherung an die Pareto-Front, wenn diese bereits sehr nah ist, schwierig, weil sehr gute Werte in der Nähe der Individuen übersprungen werden können. Andere Algorithmen, wie z. B. der `MueRhoLES` (5.1.4), verwenden daher noch zusätzliche Variabeln, welche diese Varianz dynamisch halten und ebenfalls zu optimieren versuchen.

Selektion: Bei der Selektion wird aus der Nachkommen-Population eine neue Eltern-Generation ausgewählt. Beim `OnePlusOnePG` wird das Elter-Individuum dem Nachkommen nur dann vorgezogen, wenn es den Nachkommen dominiert, also in allen Funktionswerten übertrifft und die Mutation somit ein Schritt in die falsche Richtung war. Wird das Nachkommen-Individuum nicht vom Elter dominiert, so bildet dieses die neue Eltern-Generation.

5.1.2.2 Einstellmöglichkeiten

Neben den beiden notwendigen Parametern (Testfunktion und Anzahl der Generationen) kann beim `OnePlusOnePG`-Algorithmus die maximale Größe des Pareto-Archivs, sowie die Varianz der Mutation exogen bestimmt werden (siehe Abschnitt 5.1.2.1). Für die Bestimmung der Varianz ist der optionale Parameter `-standardDeviation` notwendig. Wird dieser nicht angegeben, so wird der Standardwert 1 verwendet.
Die Beschränkung der Archivgröße mit dem optionalen Parameter `-archive` (Standard ist unbeschränkt) reduziert die notwendigen Berechnungsschritte beim Einfügen eines neuen Elementes in ein volles Archiv. Dies geht jedoch zu Lasten der Fülle oder Diversität der gefundenen Pareto-Front. Eine unbeschränkte Größe des Archivs führt bei vielen Generationen meist zu einer stetig wachsenden Laufzeit pro Generation.

5.1.2.3 Schwachstellen

Durch die Einführung des Pareto-Archivs sind die Ergebnisse des `OnePlusOnePG` natürlich deutlich umfangreicher als die des `TestAlgo`s.
Die Beschränkung auf ein Individuum pro Generation stellt insbesondere bei größeren

Suchräumen eine Einschränkung dar. Somit ist die Front der Funktionswerte oft nicht geschlossen, aber trotz dieser Schwäche ist sie unerwartet dicht. Da es hier jedoch gerade um die Implementierung eines (1+1)-EA ging, ist dieser Punkt nur durch neue Algorithmen zu verbessern.
Die Beschränkung auf ein Individuum pro Generation macht diesen Algorithmus im Gegenzug entsprechend schnell.

5.1.2.4 Verbesserungsmöglichkeiten

Aufgrund der rechenaufwändigen Bestimmung der nicht-dominierten Individuen ist eine Beschleunigung des Algorithmus durch eine effizientere Implementierung des Pareto-Archivs erreichbar.
Dies könnte eventuell über mehrstufige Archive funktionieren, da beim Einfügen eines nicht dominierten Elements dieses mit allen Individuen verglichen werden muss, die sich im Archiv befinden. Durch mehrstufige Archive müssten bei der ersten Einfüge-Stufe nur eine begrenzte Anzahl Individuen verglichen werden. Wenn davon ausgegangen wird, dass dort aufgenommene Individuen später wiederum dominiert werden, können Vergleiche mit allen Elementen des „großen“ Archivs eingespart werden. Dies gilt natürlich auch für alle anderen Algorithmen, die ein Pareto-Archiv verwenden. Ein anderer Ansatz wären möglicherweise probabilistische Algorithmen zur Bestimmung des Archivs. Eine gründlichere Recherche deckt hier vermutlich auch bessere bereits umgesetzte Lösungen auf.
Im zweiten PG-Semester haben wir uns nicht näher mit einer anderen Implementierung des Archivs beschäftigt, da unser Schwerpunkt in der Forschung und nicht in der Softwareimplementation lag.

5.1.3 Genetischer Algorithmus

Aus Sicht des Informatikers ist die biologische Evolution eine besonders geschickte, wenn auch langwierige Strategie zur Lösung von Optimierungsproblemen. Mit Genetischen Algorithmen versucht man, diese Strategie auf dem Computer zu simulieren. Genetische Algorithmen wurden in den USA seit 1962, also fast zeitgleich mit Rechenbergs und Schwefels Evolutionsstrategien, durch Holland [5] entwickelt. Unser Algorithmus lehnt sich an dessen Vorgaben an. Die wesentlichen Eigenschaften wie Selektion und Mutation wurden übernommen, während die Fitnessbewertung zur Lösung mehrkriterieller Optimierungsprobleme angepasst wurde.

5.1.3.1 Der Algorithmus

Bezug zur Biologie: Um die ursprüngliche Strategie aus der Natur zur Lösung von Optimierungsproblemen geeignet zu simulieren, hat der vorliegende GA folgende Eigenschaften:

- Ein Individuum q ist eine Folge von Einsen und Nullen, welche eine mögliche Lösung darstellt. Dieser binäre String ist in der Biologie vergleichbar mit einem Chromosom. Bei den genetischen Algorithmen entspricht jedes Bit einem Gen. Der `BasisGAPG` dient zur Lösung reellwertiger Probleme mit doppelter Genauigkeit. Um dem ursprünglichen Gedanken eines GA nahe zu kommen

werden die reellwertigen Objektvariablen tatsächlich in jedem Individuum binär kodiert.
Der Algorithmus verwendet dazu die 64 Bit-Darstellung gemäß Norm IEEE 754; zur Abbildung der boolschen Werte wird eine Zeichenkette herangezogen, deren Länge folglich ($Dimension \cdot 64$) ist.

- Jede Generation von Individuen hängt von den Bewertungen seiner Elternpopulation ab; aus Individuen mit guten Lösungswerten werden bevorzugt Nachkommen erzeugt, da sie bei der Selektion für das „Mating-Pool“ durch gute Fitnesswerte forciert werden. Bei diesem Algorithmus wird eine nicht-diskriminierende fitness-proportionale Selektion angewendet. Die μ Individuen einer Population erzeugen μ Nachkommen. Wie bei einer (μ,μ)-ES fließen die Eltern nicht mehr in die neue Population mit ein. Unsere voreingestellte Wahl für den Umfang der Population ist $\mu = 200$.

- Kreuzung: Mit der Wahrscheinlichkeit p_k werden die Individuen q_1, q_2 gekreuzt; dabei tauschen die Individuen mittels „Crossover“ ihre Gene. Beim hier in Anlehnung an Holland [5] realisierten „1-Punkt Crossover“ wählt man zufällig einen Punkt auf dem Bitstring aus und tauscht ab dort die nachfolgende Gensequenz mit dem Kreuzungspartner.

- Die Mutation invertiert anschließend mit der Mutationswahrscheinlichkeit p_m einzelne Bits der μ Individuen.

- Um zu vermeiden, dass durch die Rekombination und Mutation gute Lösungen in der Folgepopulation verloren gehen, arbeitet der `BasisGAPG` entgegen dem biologischen Vorbild mit einem zusätzlichen Archiv unbeschränkter Größe, in dem nicht-pareto-dominierte Individuen gespeichert werden.

Fitnesswerte: Zur Bestimmung der Güte eines Individuums wird der Fitnesswert benutzt, den das Individuum durch die zu optimierende Zielfunktion erhält. Bei unserer Implementierung liegt der optimale Fitnesswert bei 0.
Bei der mehrkriteriellen Optimierung erhält ein Individuum jedoch mehr als nur einen Fitnesswert, nämlich genau einen je Kriterium. Um die Güte eines Individuums trotzdem mit der Güte anderer Individuen vergleichen zu können, zieht der `BasisGAPG` alle Fitnesswerte zu einer einzigen „Gesamtfitness“ zusammen. Diese entspricht dem arithmetischen Mittel aller Fitnesswerte.

Selektion: Um zu bewirken, dass gute Individuen sich bei der Rekombination durchsetzen können und die Selektion dennoch nicht-diskriminierend vorgeht, muss sichergestellt werden, dass jedes Individuum abhängig von seiner Fitness mehrfach selektiert werden kann. Zu diesem Zweck wird in jeder Generation ein „Glücksrad“ G generiert. Bevor G gefüllt werden kann, werden die Fitnesswerte der Individuen normiert. Dies geschieht durch die Formel:

$$abs\left(\frac{Gesamtfitness\ des\ Individuums - \sum aller\ Gesamtfitnesswerte}{\sum aller\ Gesamtfitnesswerte}\right).$$

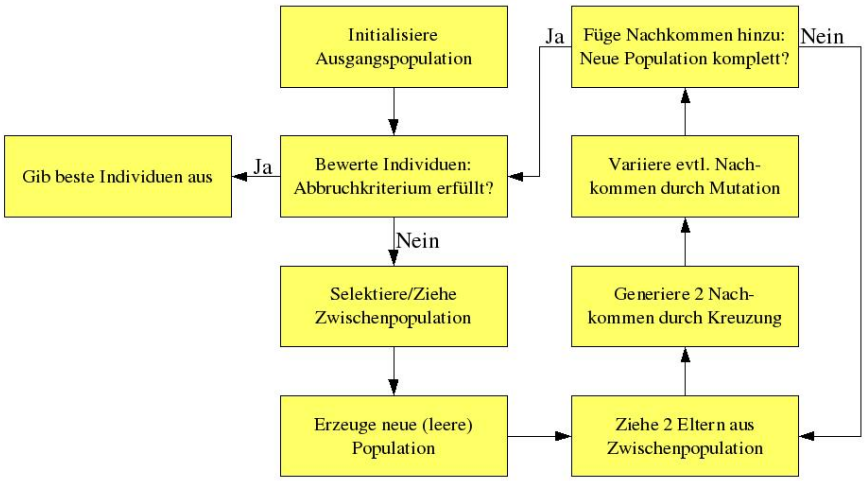

Abbildung 4: Einfache schematische Darstellung des Basis-GA. Die verwendete Abbruchbedingung ist die Anzahl der Generationen.

So erhalten wir eine neue Fitness, die abhängig von den Fitnesswerten der anderen Individuen ist und bei der große Werte gewünscht sind. Diese Umkehrung der Minimierung macht der Algorithmus sich zu Nutze, um für jedes Individuum die Anzahl der Segmente auf dem „Glücksrad“ zu bestimmen: Hat ein Individuum beispielsweise eine normierte Fitness von 0,94..., wird es in G 94 Mal berücksichtigt, während ein Individuum mit einer normierten Fitness von 0,30... nur 30 Anteile am „Glücksrad“ erhält.
Nachdem G mit den Anteilen aller Individuen gefüllt worden ist, entscheidet der Java-Zufallsgenerator μ mal, welches Individuum für die Rekombination herangezogen wird.

Rekombination und Mutation: Bei der Rekombination verwendet der `BasisGAPG` das herkömmliche 1-Punkt-Crossover. Anhand der Wahrscheinlichkeit p_k wird bei jeder anstehenden Rekombination entschieden, ob diese durchgeführt werden soll oder nicht.
Bei der anschließenden Mutation entscheidet der Algorithmus für jedes Gen eines Individuums neu, ob das Bit invertiert werden soll. Die Mutation ist abhängig von der Wahrscheinlichkeit p_m.

Ablauf: Der genaue Ablauf des `BasisGA` wird in Abbildung 4 bildlich dargestellt: Im ersten Schritt werden die Individuen der aktuellen Population (die Ausgangspo-

pulation wird stochastisch generiert) durch die Zielfunktion mit Fitnesswerten bewertet. Anschließend werden in Abhängigkeit des Fitnesswertes die Individuen für das „Mating-Pool“ selektiert. Jeweils zwei Individuen aus diesem Pool werden zur Erzeugung von zwei Nachkommen stochastisch gewählt; die so gewonnenen Nachkommen werden in die neue Population eingefügt. Wenn alle Eltern aus dem Pool gezogen wurden und somit die Nachkommenpopulation vollständig gefüllt ist, wird mit dieser die nächste Iteration durchgeführt. Erst wenn eine vorgegebene Anzahl von Generationen erzeugt wurde, wird der Algorithmus beendet und die besten gefundenen, nicht-paretodominierten Individuen aller Generationen werden ausgegeben.

5.1.3.2 Einstellmöglichkeiten

Der Algorithmus kann durch optionale oder notwendige Parameter in seiner Leistungsfähigkeit beeinflusst werden:

Generationen: Das Abbruchkriterium des `BasisGAPG` muss beim Start angegeben werden.

Populationsgröße μ: Die Anzahl der Individuen in einer Population ist ein optionaler Parameter. Wird dieser nicht angegeben, wird standardmäßig $\mu = 200$ gesetzt.

Rekombinations-Wahrscheinlichkeit p_c: Die Wahrscheinlichkeit, mit der zwei Individuen rekombiniert werden, kann als Fließkommazahl angegeben werden. Wird die Wahrscheinlichkeit vom Benutzer nicht gesetzt, verwendet der Algorithmus die Standardgröße 0,6.

Mutations-Wahrscheinlichkeit p_m: Wie wahrscheinlich das Invertieren eines Bits bei der Mutation ist, kann mit diesem Parameter bestimmt werden. Der Standardwert ist 0,01.

5.1.3.3 Schwachstellen

Laufzeit: Aufgrund der großen Anzahl an Umrechnungen von der Binärdarstellung in Fließkommazahlen (und ggf. umgekehrt), die je Individuum in jeder Generation für jede Objektvariable durchgeführt werden müssen, benötigt der `BasisGAPG` im Vergleich zu anderen Algorithmen eine merklich längere Rechenzeit. Performance-Gewinne durch weitere Codeoptimierungen sind jedoch anzunehmen.

Gesamtfitness: Die Bewertung eines Individuums durch das arithmetische Mittel aller Fitnesswerte ist nicht sehr genau; eine evtl. Gewichtung oder die besondere Güte einzelner Objektvariablen werden nicht berücksichtigt.

Selektion: Die verwendete fitnessproportionale Selektion sorgt zwar in den ersten Generationen für einen schnellen Anstieg guter Individuen in den Folgepopulationen; wenn jedoch $\frac{\mu}{2}$ Individuen eine gute Fitness vorweisen, benötigen weitere Verbesserungen zunehmend mehr Generationen [6].

Rekombination: Das 1-Punkt-Crossover hat einen hohen „positional bias“ [5]; das heißt, dass die Wahrscheinlichkeit eines Bits mit dem anderen Elter getauscht zu werden von der Postition auf dem binären String abhängt:
Während das erste Bit auf einem String nie getauscht wird (denn ein 1-Punkt-Crossover an der ersten Stelle entspricht einem Tausch des kompletten binären Strings und stellt keine „Kreuzung“ im herkömmlichen Sinne dar), wird das letzte Element auf dem String mit Wahrscheinlichkeit 1 getauscht. Durch dieses Verhalten werden bestimmte Bereiche im Suchraum anderen gegenüber forciert, und es ist unmöglich, nicht auf dem String zusammenhängende Bits eines Individuums gemeinsam zu tauschen.

IEEE 754: Die Zahlendarstellung im IEEE 754-Standard ermöglicht (zum Beispiel bei der Mutation durch das Invertieren eines einzelnen Bits) einen großen Sprung des Individuums in einen neuen Bereich des Suchraums. Dies führt häufig zur Über- oder Unterschreitung der durch die Zielfunktion vorgegebenen Grenzen und damit zur „Zerstörung“ des evolutionär gewachsenen Individuums, da es in diesem Falle vom `BasisGAPG` auf den von der Zielfunktion gelieferten Grenzwert zurückgesetzt wird. Aus diesem Grunde ist bei dem `BasisGAPG` bei der Verwendung kleiner Mutationswahrscheinlichkeiten mit besseren Ergebnissen zu rechnen.

Archivgröße: Durch die nicht festgelegte maximale Archivgröße kann es bei großen Archiven zu zusätzlicher Belastung der Rechenzeit kommen, da im Worst Case jedes Individuum einer Population gegen jedes Individuum im Archiv auf Pareto-Dominanz geprüft wird.

5.1.3.4 Verbesserungsmöglichkeiten

Gesamtfitness: Als Alternative zur Gesamtfitness ist die Verwendung der Pareto-Dominanz zur Beurteilung der Güte eines Individuums im Verhältnis zu anderen Individuen zu bevorzugen. Auch eine gewichtete Fitness wäre bereits ein Fortschritt gegenüber der zur Zeit verwendeten Methode.

Selektion: Aus der Fülle der bekannten Selektionsformen sei hier als Überlegung eine Wettkampfselektion angeführt; bei dieser werden zum Beispiel zwei Individuen mit gleicher Wahrscheinlichkeit selektiert, und nach deren Vergleich nur das Beste in den Mating-Pool übernommen. Da der `BasisGAPG` die Testfunktion minimiert, kann ohne Umformung der Fitness die Wettkampfselektion angewandt werden. Außerdem bietet sie vor dem Hintergrund der sowieso hohen Laufzeit des Algorithmus Vorteile hinsichtlich des Rechenaufwandes gegenüber den Selektionsformen die auf „Ranking“ basieren.

Rekombination: Um den schlechten Eigenschaften des verwendeten 1-Punkt-Crossovers entgegen zu wirken, könnte an seine Stelle ein „Shuffle-N-Point-Crossover“ mit Selbstanpassung treten, dessen Anzahl von Crossover-Punkten von der Dimension der Testfunktion (also der Länge des binären Strings) abhängt.

Kombinatorische Probleme: Als zukünftige Erweiterung wurde an die Bereitstellung des `BasisGAPG` für kombinatorische Probleme gedacht, um die Effizienz des Algorithmus unabhängig von den kostspieligen Transformationen zwischen Binär- und Real-Darstellung testen zu können. Da im Verlauf der PG jedoch die Untersuchung kombinatorischer Probleme nicht im Mittelpunkt stand und in *NObELTJE* mehrheitlich reellwertige Testfunktionen und Anwendungsprobleme implementiert sind, wurde von diesem Vorhaben abgelassen. Somit bleibt dieser Hinweis nur als Anregung für künftige Erweiterungen und Experimente bestehen.

5.1.4 Evolutionäre Strategie MueRhoLES

Die Evolutionsstrategien (ES) lehnen sich an den biologischen Prozess der Evolution an. Dabei geht es um zwei Prozesse der biologischen Evolution: Mutation und Selektion. Diese Vorgänge werden für die ES stark vereinfacht. Evolutionsstrategien wurden von Schwefel [7] und Rechenberg [8] in den 60er Jahren entwickelt und brachten sehr gute Ergebnisse bei Problemen, die auf konventionelle Art nur sehr schwer zu lösen waren. Die ursprüngliche einfache Strategie mit einem Individuum wurde ständig erweitert. Insofern ist der `MueRhoLES` eine Erweiterung des `OnePlusOnePG`-Algorithmus.

5.1.4.1 Der Algorithmus

Beschreibung: Bei dem implementierten Algorithmus handelt es sich um eine (μ/ρ, κ, λ)-Evolutionsstrategie mit σ-Selbstadaption und *non-dominated-sorting*. Dabei steht:

- μ für die Anzahl der Individuen in der Eltern-Generation,
- ρ für die Anzahl der Eltern, aus deren Variablen ein Individuum der Nachkommen-Generation geschaffen wird,
- κ für die Lebensdauer des Individuums,
- λ für die Anzahl der Individuen in der Nachkommen-Generation.

Der Begriff der σ-Selbstadaption beschreibt eine fortgehende Anpassung der Strategievariablen. Jede Strategievariable fungiert bei der Mutation für genau eine Objektvariable als Mutationsschrittweite und bestimmt somit die Stärke der Mutation der Objektvariablen, also wie sehr diese verändert werden. Jede Objektvariable hat zur individuellen Mutation eine eigene Strategievariable. Jedes Individuum hat also gleich viele Objekt- und Strategievariablen.

Allgemeiner Pseudo-Code:

```
BEGIN
    Generation := 0;
    initialisiere die Population (μ Individuen mit Objekt- und Strategie-
    variablen);
    wiederhole
        For i:=1 To λ Do
```

```
                wähle zufällig ρ Eltern aus;
                s_i := rekombiniere die Strategievariablen der ρ Eltern;
                y_i := rekombiniere die Objektvariablen der ρ Eltern;
                mutiere die Strategievariablen s_i;
                mutiere die Objektvariablen y_i;
                erzeuge ein neues Individuum aus s_i und y_i;
            End;
            erzeuge aus den λ Individuen die Nachkommen-Generation;
            lösche zu alte Individuen und wähle μ Individuen für die nächste
            Eltern-Generation;
            End;
            Generation++;
        wiederhole bis zum Abbruchkriterium
    End
```

Implementierung: Zuerst wird die Start-Population bzw. die erste Eltern-Population erzeugt, d. h. es wird eine Liste der μ Individuen mit dazugehörigen Objekt- und Strategievariablen erzeugt. Die Objektvariablen werden zufällig innerhalb der von der Testfunktion gegebenen Grenzen initialisiert.
Die Double-Werte der Strategievariablen werden zufällig aus dem Intervall $]0; 5]$ gewählt. Wichtig ist, dass die Strategievariablen nicht 0 werden, weil dann keine Mutation stattfinden würde. Da die Mutation bei zu großen Strategievariablen eine zu große Veränderung der Objektvariablen bewirken würde und sich das Individuum womöglich ungewünscht groß verändere, wurde die obere Grenze für die Initialisierung auf fünf festgelegt, was ein guter Wert für die anfängliche „grobe“ Suche nach der Pareto-Front ist.

Rekombination und Mutation: Aus der Eltern-Population werden ρ Individuen zufällig ausgewählt. Aus diesen Individuen wird ein neues Nachkommen-Individuum (Kind) erzeugt. Jedes Kind wird also aus ρ Individuen rekombiniert. Die Rückgabe der rekombinierten Individuen erfolgt ohne Fitnesswerte, da diese für das neue Individuum natürlich noch berechnet werden müssen.
Jedes Individuum hat für jede Objektvariable eine Strategievariable, die die Mutation der Objektvariable als Standardabweichung beeinflusst. Nach der Rekombination werden die Strategie- und Objektvariablen mutiert. Die Mutation der Strategievariablen erfolgt logarithmisch, kombiniert aus einem Wert, der bei einem Mutationsschritt für alle Strategievariablen eines Individuums gleich ist und einem anderen, der für jede Strategievariable unabhängig gewählt wird. Die Objektvariablen werden durch Addition eines mit der entsprechenden Strategievariablen gewichteten Gauß-verteilten Zufallswertes mutiert. Nach der Mutation wird der Fitnesswert berechnet und dem Individuum übergeben. Durch λ Wiederholungen wird eine Nachkommen-Generation mit λ Individuen generiert.

Selektion: Die Lebenszeit eines Individuums ist durch κ begrenzt. Ist diese überschritten, wird das Individuum nicht mehr in die neue Eltern-Population übernommen. Somit ergibt sich für $\kappa = 1$ eine Komma-Selektion, bei der die Selektion nur aus

der Nachkommen-Generation auswählt. Der Algorithmus ist so implementiert, dass bei einem κ-Wert unter 1 automatisch eine Plus-Selektion angewendet wird, d. h. die Lebenszeit der Individuen ist unbegrenzt, wodurch immer aus der Vereinigung der kompletten Eltern- und der Nachkommen-Generation selektiert wird.
Die noch verbleibenden Individuen der Eltern-Generation und der Nachkommen-Generation werden gemischt und aus der Menge werden μ Individuen für die nächste Eltern-Generation ausgewählt.

Die weitere Selektion erfolgt mittels Einteilung in Fronten nicht dominierter Individuen (*non-dominated-sorting*) und läuft sehr ähnlich der vom *NSGA-II* ab [9]. Zuerst werden die nicht-dominierten Individuen ausgewählt, in die neue Eltern-Generation hinzugefügt und aus der alten Menge gelöscht. Dadurch entsteht eine neue Pareto-Front in der alten Menge. Diese neue Pareto-Front wird auch in die neue Generation hinzugefügt und aus der alten gelöscht. Das wird solange wiederholt, bis die neue Generation vollständig ist. Falls es in der Front mehr nicht-dominierte Individuen gibt als benötigt, wird aus diesen zufällig ausgewählt.

5.1.4.2 Einstellmöglichkeiten

Der Algorithmus braucht neben den oben genannten die folgenden (exogenen) Startparameter:

- die Anzahl der Individuen in der Eltern-Generation (Parameter `-mue`),
- die Anzahl der Eltern eines Individuums (`-rho`),
- die Anzahl der Individuen in der Nachkommen-Generation (`-lambda`),
- die Lebensdauer eines Individuums in Generationen angegeben (`-kappa`),
- die Standardabweichung für den festen Teil der Mutation der Strategievariablen (`-tau1`) und
- die Standardabweichung für den variablen Teil der Mutation der Strategievariablen (`-tau2`).

5.1.4.3 Verbesserungsmöglichkeiten

Durch die große Anzahl der Einstellmöglichkeiten bei der Evolutionsstrategie können Ergebnisse stark variieren. Allgemein ist es nicht immer einfach, die richtige Schrittweite zu bestimmen. Wenn die Schrittweite gegen 0 läuft, besteht die Gefahr dass sich der Algorithmus zu sehr verlangsamt. Wenn die Schrittweite zu groß wird, besteht die Gefahr, gute Lösungen zu verfehlen. Durch die verwendete σ-Selbstanpassung schränkt man diese Gefahren ein.
Im zweiten Semester hat sich die Projektgruppe schließlich kaum um die weitere Erforschung dieser Strategie gesorgt. Sie wurde jedoch von der Robustsheitsgruppe in Kapitel 7.3.2 näher untersucht. Weiterhin wurde der `MueRhoLES` auf neue Testfunktionen und Praxisprobleme angewandt und es wurde dabei notwendigerweise auch problemabhängig nach guten Einstellungen gesucht.

5.1.5 Basis-PSO

Beim `BasisPSO` handelt es sich um die einfache Implementierung eines Standard Particle Swarm Optimization Algorithmus, der so angepasst wurde, dass er auf mehrkriterielle Probleme anwendbar ist.
Erste Ansätze hierzu hierzu lieferte ein Artikel von Parsopoulus und Vrahatis [10].

5.1.5.1 Der Algorithmus

Beschreibung: Der PSO Algorithmus simuliert die Bewegung eines Schwarms. Anders als bei anderen evolutionären Algorithmen gibt es hier keine Mutation, Rekombination etc. Stattdessen verändert jedes einzelne Partikel pro Iteration seine aktuelle Position, so dass es quasi über den Entscheidungsraum „fliegt“. In jedem Durchgang wird für alle Partikel erst die aktuelle Position bewertet, daraufhin wird ein neuer Geschwindigkeitsvektor berechnet und schließlich wird die Position aktualisiert.
Der klassische PSO-Algorithmus ist für einkriterielle Probleme ausgelegt. Um sie auf multikriterielle anzuwenden, wurden verschiedene Methoden getestet, die einzelnen Kriterien mit jeweils anderen Gewichtungen aufzusummieren, wie in [10] vorgeschlagen. Schließlich gingen wir aber dazu über, die Positionen auf Pareto-Dominanz hin zu vergleichen und zu bewerten.
Bei unserer Implementierung wird pro Lauf immer nur ein Ergebnis erzeugt, so dass für eine brauchbare Lösung mehrere Läufe benötigt werden.

Bewertung der Position: Jedes Partikel merkt sich neben seiner aktuellen Position zusätzlich seine beste bisher erreichte. Um zu bestimmen, wie gut eine Lösung ist, kann man die Fitnesswerte der Objektvariablen gewichtet aufsummieren und erhält einen Fitnesswert F mit $F = \Sigma_{i=1}^{k} w_i f_i(x)$, wobei $w_i, i = 1, ..., k$, die nicht-negativen Gewichtungen sind. Normalerweise wird angenommen, dass $\Sigma_{i=1}^{k} w_i = 1$. Um dies zu erreichen, gibt es mehrere Möglichkeiten:

- Conventional Weighted Aggregation (CWA): Bei dieser Vorgehensweise werden alle Fitnesswerte gleichgewichtet addiert, also $w_1 = w_2 = ... = w_k$. Allerdings bereitet es CWA Probleme, konkave Regionen der Pareto-Front zu finden.

- Bang-Bang Weighted Aggregation (BWA): Natürlich müssen die Gewichtungen nicht gleich sein und können sogar im Verlauf des Algorithmus noch geändert werden. Für ein zweikriterielles Problem sieht die Gleichung wie folgt aus:

  $$w_1(t) = sign(sin(2\pi t/F)), w_2(t) = 1 - w_1(t)$$

  wobei t der Iterations-Index ist. Die $sign()$ Funktion wechselt beim Aufruf das Vorzeichen des Ausdrucks. Die Konstante F bei BWA und DWA wird als change frequency bezeichnet.

- Dynamic Weighted Aggregation (DWA): Alternativ können die Gewichtungen auch allmählich fortschreitend geändert werden und nicht so abrupt wie bei BWA:

  $$w_1(t) = |sign(2\pi t/F)|, \qquad w_2(t) = 1 - w_1(t)$$

In dieser Implementierung kommt allerdings mittlerweile eine andere Idee zum Einsatz. Die aktuelle Position $P = \{x_1, x_2, ..., x_k\}$ eines Partikels ist genau dann besser als die bisher beste, wenn der Zielvektor $F = \{f(x_1), f(x_2), ..., f(x_k)\}$ nicht dominiert wird.

Berechnung des Geschwindigkeitsvektors: Der neue Geschwindigkeitsvektor wird nach folgender Gleichung bestimmt:

$$v_i(t+1) = w * v_i(t) + c_1 * rand() * (pBest_i - position_i)$$
$$+c_2 * rand() * (gBest_i - position_i)$$

wobei w der Trägheitswert ist. Mit seiner Hilfe lässt sich steuern, wie stark der alte Geschwindigkeitsvektor in die Gleichung mit eingeht, und somit, ob eher eine lokale oder globale Suche stattfindet.
Die Lernfaktoren c_1 und c_2 sind Konstanten, die üblicherweise aus dem Intervall $[0..4]$ gewählt werden.
Bei $pBest$ und $gBest$ handelt es sich um die bisher beste Position des aktuellen Partikels bzw. des Schwarmbesten.
Die $rand()$ Funktion liefert einen Zufallswert aus dem Bereich $[0..1]$.

Aktualisieren der Position: Wenn jedes Partikel bewertet und ein neuer Geschwindigkeitsvektor bestimmt wurde, muss schließlich die Position noch aktualisiert werden. Dies geschieht einfach, indem der Geschwindigkeitsvektor zur Position hinzu addiert wird:

$$x_i(t+1) = x_i(t) + v_i$$

5.1.5.2 Beobachtungen

Bei PSO nähern sich die Individuen sehr schnell der Pareto-Front, so dass man bereits mit wenigen Iterationen gute Ergebnisse erhält.
Möglicherweise liegt es daran, wie die einzelnen Partikel bewertet werden (Vergleich über Pareto-Dominanz). Auf jeden Fall weichen die „guten“ Einstellungen von den Standardeinstellungen aus der Literatur leicht ab. So erzielt der Algorithmus gute Ergebnisse für die Testfunktion `FonsecaF1`, wenn der Trägheitswert eher niedriger gewählt wird (ca. 0,2), dafür aber mit einer größeren Population von etwa 100 gearbeitet wird.
Ein Nachteil dieser Variante von PSO für mehrkriterielle Probleme ist, dass pro Durchgang immer nur eine Lösung erzeugt wird, nämlich die Position des besten Partikels im gesamten Schwarm. Deswegen muss der Algorithmus für ein brauchbares Ergebnis mehrmals laufen. Eine Alternative hierzu ist zum Beispiel mit dem `MopsoOne` implementiert worden (nachfolgend in Kapitel 5.1.6 beschrieben).

Weiterhin ist im Verlauf der Experimente aufgefallen, dass der Algorithmus zwar gute, brauchbare Lösungen liefert, allerdings kommt es durchaus vor, dass sich die einzelnen Partikel der Lösungsmenge zum Teil dominieren. Dies kommt daher, dass pro Lauf jeweils nur das beste Partikel in die Lösungsmenge aufgenommen wird, und die einzelnen Läufe unabhängig voneinander sind. Der Algorithmus wurde dahingehend

modifiziert, dass die Partikel in ein pareto-optimales Archiv einsortiert werden. Jedesmal, wenn ein neues Partikel eingefügt wird, werden bereits bestehende Lösungen, die dominiert werden, verworfen.

5.1.6 Multikriterieller PSO

Der von uns implementierte Algorithmus ist eine mehrkriterielle Version eines Partikel Swarm Algorithmus für reelwertige Minimierungsprobleme. Dabei haben wir uns an dem Artikel von Coello Coello und Salazar Lechuga [11] orientiert.

5.1.6.1 Der Algorithmus

Beschreibung: Der Algorithmus basiert auf einer Population von Partikel, die sich wie Vögel in einem Schwarm gegenseitig bei der Bewegung im Suchraum beeinflussen.
Die Bewegung eines einzelnen Partikels wird durch seine letzte, seine aktuelle, seine beste bisher erreichte Position und die beste von allen Partikel erreichte Position bestimmt. Dazu wird für jeden Partikel seine aktuelle Position `pos`, seine bisher beste erreichte Position `pBest` ein Geschwindigkeitsvektor `vel` und die Fitnesswerte `posFit` und `pBestFit` (`posFit` ist natürlich der Wert der Testfunktion für `pos`) als Vektoren im Suchraum bzw. Zielraum gespeichert. Die beste von allen Partikel erreichte Position wird bei jeder Iteration aus dem Pareto-Archiv ermittelt. In jeder Iteration wird für jeden Partikel aus diesen Informationen und Zufallsvariablen mittels einfacher Vektorrechnung ein neuer Geschwindigkeitsvektor berechnet und zur aktuellen Position addiert. Nachdem die Funktionswerte für die neue Position berechnet sind, wird überprüft, ob die neue Position besser als die bisher beste erreichte Position ist und im positiven Fall wird `pBest` aktualisiert. Anschließend werden die nicht dominierten Partikel ins Pareto-Archiv eingefügt und die nun dominierten Partikel aus dem Archiv entfernt.
Die Entscheidung, ob eine Position besser als eine andere ist, wird anhand der Pareto-Dominanz gefällt. Es wird keine Fitnessfunktion benutzt, die aus dem Vektor der Fitnesswerte eine reelle Zahl berechnet, um so eine totale Ordnung zu definieren. Sollte die Pareto-Dominanz für zwei oder mehrere Partikel kein eindeutiges Ergebnis liefern, werden andere Methoden benutzt, um den besten Partikel zu bestimmen.
Der Pseudo-Code sieht folgendermaßen aus:

1. Initialisiere Partikel
2. Aktualisiere das Pareto-Archiv
3. FOR i = 1,.., Anzahl Generationen
    - Berechne für jeden Partikel neuen Geschwindigkeitsvektor
    - Berechne neue Positionen durch Addition der Geschwindigkeitsvektoren zu den aktuellen Positionen
    - Aktualisiere das Pareto-Archiv

Einstellungsmöglichkeiten: Neben den Parametern, die im Kapitel 5.1 beschrieben wurden, können dem `MopsoOne`-Algorithmus folgende Parameter angegeben werden:

- `quantityParticle`: Anzahl der Partikel
- `fevals`: maximale Anzahl der Funktionsauswertungen. Wird dieser Parameter per Kommandozeile oder Datei übergeben, so wird die Anzahl der Generationen in Abhängigkeit von der Anzahl der Partikel bestimmt und im Algorithmus überschrieben. Die Anzahl der Generationen wird dann durch die Formel `g` = `fevals` / `quantityParticle` bestimmt, wobei `g` für die Anzahl der Generationen steht, und für jeden Algorithmus angegeben werden muss. Wird die Anzahl der Partikel nicht angegeben, so wird diese durch die Formel `quantityParticle` = `fevals` / `g` bestimmt. Wird der Algorithmus mittels der Designklasse gestartet, so werden die Werte für die Parameter `fevals`, `g` und `quantityParticle` auch im Dateinamen korrigiert.
- `maxVelocity`: Maximale Schrittweite der Partikel.
- `maxRepository`: Größe des Pareto-Archivs.
- `quantityHypercube`: Ungefähre Anzahl der Hypercuben im Zielraum.
- `inertia`: Trägheitswert (Einfluss des letzten Geschwindigkeitsvektors für die Berechnung der neuen Position).
- `scalingInertia`: Gibt an, ob der Parameter `inertia` während des Laufs verändert werden soll. Wird `scalingInertia` per Kommandozeile auf 1 gesetzt, so wird der Wert für `inertia` am Anfang des Laufs auf den Wert des Parameters `maxInertia` gesetzt. Mit dem Parameter `dueGeneration` kann dann bestimmt werden, wie lange `inertia` verändert werden soll. Bei einem Wert von 0,7 für `dueGeneration` und 100 Generationen wird `inertia` bis zur 70. Generation linear verändert. Der Endwert für `inertia` wird durch den Parameter `minInertia` bestimmt. Dieses Vorgehen hat sich im Einkriteriellen bewährt und soll die Schrittweite des Algorithmus zum Ende des Laufs verkleinern.
- `maxInertia`: Startwert ders Parameters `inertia` (nur von Bedeutung, wenn `scalingInertia` den Wert 1 hat).
- `minInerta`: Wert des Parameters `inertia` am Ende des Laufs (nur von Bedeutung, wenn `scalingInertia` den Wert 1 hat).
- `dueGeneration`: Gibt den Anteil an Generationen an, in denen `inertia` verändert wird. Der Wert sollte zwischen 0 und 1 liegen (nur von Bedeutung, wenn `scalingInertia` den Wert 1 hat).
- `c1`: Einfluss der besten bisher erreichten Position des Partikels.
- `c2`: Einfluss der global besten Position.

- `onlineViz`: Wird dieser Parameter auf 1 gesetzt, so wird in jeder Generation ein png-Bild der Partikel im Suchraum erstellt und in einem separaten Verzeichnis gespeichert. Damit kann mittels der Klasse `AnimatedGifEncoder` (siehe Kapitel 5.3.8) ein animiertes Bild erstellt werden, dass die Bewegung der Partikel im Suchraum währen des Laufs zeigt.

Initialisierung der Partikel: Für jeden Partikel wird die aktuelle Position zufällig gemäß Gleichverteilung innerhalb des von der Testfunktion gegebenen Startbereichs gewählt.
Die beste bisher erreichte Position erhält den Wert der aktuellen Position und die Geschwindigkeitsvektoren werden mit 0 initialisiert. Anschließend wird die Testfunktion ausgewertet und `posFit` und `pbestFit` mit diesen Werten initialisiert:
`pbest` = `pos`,
`vel` = `0`.

Berechnung der Geschwindigkeitsvektoren: Fur jeden Partikel werden die neuen Geschwindigkeitsvektoren `vel` nach dieser Formel berechnet :

$$\overrightarrow{vel} = inertia * R_1 \times \overrightarrow{vel} + c_1 * R_2 \times (\overrightarrow{pbest - pos}) + c2 * R_3 \times (\overrightarrow{repBest - pos})$$

wobei R_1, R_2 und R_3 Zufallszahlen zwischen 0 und 1 sind. $*$ steht für Multiplikation $\times$ für skalare Multiplikation und $+$ für die Vektoraddition. Der Vektor $\overrightarrow{repBest}$ soll die Position eines sehr guten Partikels aus dem Pareto-Archiv sein, an dem sich alle Partikel orientieren und der in jeder Iteration neu bestimmt wird. Dies geschieht, indem man den Zielbereich in Hypercubes unterteilt und einen Hypercube findet, der am wenigsten bewohnt ist. Daraus wird ein Partikel zufällig gewählt.

Berechnung der neuen Positionen: Die neue Position eines Partikel ergibt sich durch Addition des Geschwindigkeitsvektors `vel` zur aktuellen Position. Wird die bisher beste Position von der neuen Position dominiert, so wird diese zur neuen besten Position. Sind beide Positionen bezüglich der Pareto - Optimalität nicht vergleichbar, so wird zufällig mit Wahrscheinlichkeit 0,5 die beste Position beibehalten.

Paretoarchiv: Nach jeder Iteration wird das Pareto-Archiv aktualisiert. Sollte die Kapazität des Archivs überschritten werden, so wird mit Hilfe der Hypercubes ein Partikel entfernt, der sich in einem stark bewohnten Teil des Zielbereichs befindet, was eine gute Streuung der Pareto-Front garantieren soll.

5.1.6.2 Einstellungen und Beobachtungen

Anzahl der Iterationen und Partikel Bei geringer Anzahl der Partikel kann es vorkommen, dass die Partikel in einem schlechten Teil des Suchbereichs initialisiert werden und sich sehr langsam der Pareto-Front nähern. Bei einer Partikelanzahl von 30 und einer Iterationsanzahl von 50 lieferte der Algorithmus schon sehr gute Ergebnisse, die sich bei Erhöhung der Werte auch nicht verschlechtert haben.

Trägheitswerte Die Konstanten `inertia`, c_1 und c_2 haben einen entscheiden Einfluss auf die Schrittweite und somit die Konvergenzgeschwindigkeit des Algorithmus. Ein hoher Trägheitswert im Bereich von 1 ermöglicht eine schnelle Annäherung an die Pareto-Front, was aber bei der lokaler Suche den Nachteil hat, dass die Partikel über die optimalen Werte „hinwegfliegen“. Die bisher besten Ergebnisse lieferte der Algorithmus mit der Belegung `inertia` = 0,5, c_1 = 0,3 und c_2 = 0,3, was aber auch an der Einfachheit der benutzten Testfunktionen liegt, bei der die Gefahr im lokalen Minima stecken zu bleiben nicht besteht.

Anzahl der Hypercubes Mit dem Parameter `quantityHypercube` kann nur eine ungefähre Anzahl der Hypercubes im Zielbereich angegeben werden. Das liegt daran, dass der Zielraum bezüglich jeder Dimension in gleich viele Teilintervalle geteilt wird. Die eigentliche Anzahl der Hypercubes im Zielbereich ergibt sich durch diese Formel:

$$Anzahl = (round(\sqrt[Dim]{quantityHypercube}))^{Dim}$$

Ein großer Wert für die Anzahl der Hypercubes garantiert eine optimale Streuung der Pareto-Front. Ein Nachteil ist jedoch, dass die Randwerte der Pareto-Front, die für die Optimierung eher uninteressant sind, großen Einfluss auf die Bewegung der Partikel haben.

5.1.6.3 Verbesserungsmöglichkeiten

Die naheliegendste Verbesserungsmöglichkeit ist die Bestimmung der optimalen Werte für den Trägheitswert `inertia` und die Konstanten c_1 und c_2.
Ein weiterer Aspekt wäre, das optimale Verhältnis zwischen der Anzahl der Partikel und Anzahl der Iterationen bei einer vorgegebenen Anzahl der Funktionsauswertungen zu finden, die bei zeitaufwendigen Simulatoren beschränkt sind.

Im zweiten Semester wurden diese Themen von der Gruppe „Robustheit “ ausführlich im Kapitel 7.3.2 untersucht.

5.1.7 Räuber-Beute

Der Räuber-Beute-Algorithmus ist ein competitiver coevolutionärer Algorithmus (*competitive CEA*). Ein competitiver CEA besteht aus konkurrierenden Populationen, die in Interaktion miteinander gleichzeitig optimiert werden. Der von uns implementierte Räuber-Beute-Algorithmus lehnt sich stark an das von Laumanns [12] beschriebene Modell an. Er entspricht dem Konzept der Coevolution insofern, als dass er aus einer Beute- und einer gegnerischen Räuber-Population besteht, welche die Beute-Individuen „frisst“. Allerdings durchläuft nur die Beute-Population eine Evolution, wohingegen die Räuber nur symbolisch für das Auslösen von Selektions-Ereignissen (das „Fressen“ der Beute) stehen.
Im Vergleich zu Laumanns Modell wurden einige Vereinfachungen vorgenommen, um den Algorithmus möglichst überschaubar und gut nachvollziehbar zu halten, es wurden aber auch zusätzliche Optionen realisiert.

5.1.7.1 Der Algorithmus

Beschreibung Die Populationen des Algorithmus leben auf einem Graphen, dessen räumliche Struktur einen Torus bildet. Jedes Beute-Individuum hat darin einen Knoten als feste Position. Die Räuber bewegen sich über die Graph-Knoten in Form eines Irrlaufs (*random walk*). Dabei verursachen sie an ihrem Aufenthaltsort eine lokale Selektion der Beute-Individuen, das heißt, das schwächste Beute-Individuum innerhalb ihrer Nachbarschaft wird aussortiert, also ersetzt. Das neue Beute-Individuum entsteht an der entsprechenden Position durch lokale Rekombinations-Operatoren aus den Individuen der Nachbarschaft und wird anschließend mutiert.
Jedem Räuber ist ein Kriterium des mehrkriteriellen Optimierungsproblems zugeordnet. Der Selektions-Operator, den Laumanns beschreibt, bezieht sich jeweils nur auf dieses eine Kriterium. Es wird also das Beute-Individuum aussortiert, das bezüglich des Räuber-Kriteriums den schlechtesten Wert hat. Zudem wurde von der Projektgruppe noch ein weiterer Selektions-Operator implementiert, welcher nicht ausschließlich an ein Kriterium gekoppelt ist. Der Ablauf des Algorithmus ist schematisch durch den Pseudo-Code dargestellt.

Pseudo-Code des Räuber-Beute-Algorithmus

1 Initialisiere Populationen:
- 1.1 Für alle Graph-Knoten:
  - 1.1.1 erzeuge Beute-Individuum mit zufälligen Objektvariablen
  - 1.1.2 ordne Beute-Individuum den Knoten als Position zu
- 1.2 Für alle Kriterien:
  - 1.2.1 erzeuge Räuber, ordne ihm Kriterium zu, wähle zufällige Position

2 Für <Anzahl Generationen>:
- 2.1 Für alle Räuber:
  - 2.1.1 bewege Räuber gemäß random walk
  - 2.1.2 Selektion:
    - 2.1.2.1 bestimme Nachbarschaft des Räubers
    - 2.1.2.2 bestimme schwächste Beute-Individuen
    - 2.1.2.3 wähle gleichverteilt aus 2.1.2.2 ein Individuum „Beute“
  - 2.1.3 Rekombination (diskret):
    - 2.1.3.1 bestimme Nachbarschaft von Beute
    - 2.1.3.2 rekombiniere Nachbarschaft diskret zu neuem Individuum
  - 2.1.4 Mutiere neues Individuum
  - 2.1.5 weise neuem Individuum die Position von Beute zu

Operatoren Hier werden nun die einzelnen Operatoren des Algorithmus detaillierter beschrieben.

Räuber-Bewegung (random walk): Die Räuber bewegen sich zufällig vom jeweils aktuellen Knoten zu einem der Nachbarknoten (*random walk*). Die sogenannte Nachbarschaft bilden hier die direkten Nachbarn, also beim Torus genau die vier Knoten, die mit dem aktuellen adjazent sind. Die Bewegung in jede

Richtung ist dabei gleichwahrscheinlich. Man könnte andere Wahrscheinlichkeitsverteilungen wählen und damit einem Räuber gewisse Bewegungstendenzen zuordnen.

Selektion: Der Räuber selektiert aus seiner unmittelbaren Nachbarschaft (dem aktuellen Knoten, sowie den vier direkten Nachbarn) das „schwächste“ Individuum, das heißt, das Individuum, welches in Bezug auf das Kriterium des Räubers den größten Fitness-Wert aufweist (falls es sich um ein Minimierungsproblem handelt). Das multi-kriterielle Problem wird also bei dieser Methode in mono-kriterielle Funktionen zerlegt, die sequentiell optimiert werden.
Neben diesem Selektionsoperator nach Laumanns [12] haben wir noch einen weiteren implementiert, der alternativ gewählt werden kann. Es wird die gleiche Nachbarschaft verwendet, aber daraus wird die Menge der innerhalb der Nachbarschaft dominierten Individuen bestimmt. Von diesen wird zufällig gleichverteilt eines zur Ersetzung ausgewählt.

Rekombination: Ein neues Beute-Individuum wird aus seinen vier nächsten Nachbarn rekombiniert. Wir haben in unserem Räuber-Beute-Algorithmus die „diskrete“ Rekombination gewählt, da sie auch von Laumanns als vorteilhaft klassifiziert wurde und zudem einfach zu implementieren ist.
Bei der diskreten Rekombination wird für jede Objekt-Variable des neuen Individuums der entsprechende Wert eines zufällig ausgewählten Individuums seiner Nachbarschaft übernommen.

Mutation: Nach der Rekombination werden die Objekt-Variablen des neuen Beute-Individuums mutiert. Dabei wird zur jeweiligen Objekt-Variable eine normalverteilte Zufallsvariable mit per Parameter einstellbarer Varianz addiert.
Die Varianz wird zunächst beim Algorithmus-Aufruf als fester Wert übergeben. Da die Varianz (entspricht der Mutationsschrittweite) die Ergebnisse des Algorithmus stark beeinflusst und stark vom gestellten Optimierungsproblem abhängt, wäre eine dynamische Mutationsschrittweiten-Steuerung eine erstrebenswerte Erweiterung des Operators.

5.1.7.2 Einstellmöglichkeiten

Der Algorithmus bietet eine Reihe von externen Parametern, mit denen sein Verhalten beeinflusst werden kann. Wir listen diese auf und geben Werte an, die per Voreinstellung (*default*) angenommen werden.

Graphengröße: Als Voreinstellung wird ein Graph der Größe (32×32), also ein Torus mit 1024 Knoten, verwendet. Einen Torus kann man sich als ein an den Enden verbundenes (sowohl in horizontaler, als auch in vertikaler Ausrichtung) Gitter vorstellen. Die Größe des Gitters in den zwei Dimensionen x und y kann über Eingabeparameter eingestellt werden. Es ergibt sich ein Torus der Größe ($\texttt{sizeX} \times \texttt{sizeY}$).

Anzahl der Generationen: Die Anzahl der Generationen (`g`) stellt derzeit das einzige Abbruchkriterium des Algorithmus dar und hat daher großen Einfluss auf

die Qualität der Optimierung. Als Default-Wert werden zunächst 1000 Generationen vorgeschlagen, die innerhalb weniger Sekunden zu berechnen sind. Da der Graph bei 32×32 Knoten 1024 Individuen enthält, ist ein Wert größer 1000 zu empfehlen.

Anzahl der auszugebenden Individuen: Die Anzahl der auszugebenden Individuen wird mit dem Aufrufparameter `archive` eingestellt. Es werden dann aus der letzten Generation `archive`-viele nicht-dominierte Individuen ausgegeben (sofern so viele enthalten sind).

Mutationsschrittweite: Wie in 5.1.7.1 beschrieben, ist die Mutationsschrittweite statisch und wird über den externen Parameter `variance` gesetzt. Sie sollte im zweiten Semester der Projektgruppe dynamisch angepasst werden, zu Beginn des Algorithmus-Laufes groß sein und mit zunehmender Generationen-Anzahl kleiner werden, um „gute“ Individuen später nicht mehr zu stark zu mutieren. Dies ist allerdings nicht erfolgt, da sich die Gruppe auf andere Aufgaben konzentriert hat.

Selektion: Neben der Selektion nach Laumanns, die in 5.1.7.1 beschrieben wurde, wurde noch ein weiterer Selektionsoperator implementiert. Der gewünschte Operator kann über den Parameter `selection` gewählt werden. Den Operatoren sind Nummern zugeordnet. 0 entspricht der Selektion nach Laumanns (default) und 1 einer Selektion, die auf der Eigenschaft der Pareto-Dominanz basiert.

5.1.7.3 Beobachtungen

Trotz einfachster Einstellungen des Algorithmus und zudem fester Mutationsschrittweite liefert der Algorithmus sehr gute Ergebnisse. Das Testproblem `DoubleParabola` wurde erstaunlich gut angenähert, sowohl bei Mutationsschrittweiten von 5, 1 oder 0,5.
Das Testproblem `FonsecaF1` zeigte sich schon sensibler. Selbst bei der bisher besten gefunden Mutationsschrittweite von 0,5 wird die Pareto-Front nur grob angenähert.
Als sehr gute Kombination erwies sich, wie auch schon von Laumanns benutzt, eine große Anzahl von Generationen, verbunden mit sehr kleinen Schrittweiten. So ergab die Kombination von $1.000.000$ Generationen und eine Mutationsschrittweite von lediglich 0,02 eine sehr gute Annäherung an die Pareto-Front.

Selektion Durch anfängliche Fehler in der Implementierung der Selektion haben wir einige interessante Feststellungen entwickelt und wurden zu neuen Selektions-Operatoren inspiriert.

Selektion nach Laumanns Bei der korrekten Implementierung von Laumanns Selektion weisen die Individuen entlang der Pareto-Front eine gute Diversität auf. Nach unseren ersten Überlegungen hatten wir erwartet, dass Individuen mit extremen, also sehr großen Werten einer Zielfunktion, vorrangig selektiert werden. Dadurch hätte die Abdeckung der Pareto-Front vor allem an den Rändern Lücken aufweisen müssen. Da ein Räuber immer das Individuum selektiert, das für sein Kriterium den größten

Wert hat, scheinen Individuen, die eher durchschnittliche Werte in allen Kriterien aufweisen, vor der Selektion geschützt.
Wegen der Eigenschaft der Pareto-Optimalität haben die Individuen, die bzgl. eines Kriteriums besonders schlecht sind, bei einem anderen besonders gute Werte. Sind die Randbereiche der Pareto-Front nicht abgedeckt, dann fehlen somit auch die extrem guten Lösungen der jeweiligen Kriterien.
Bei unseren Überlegungen haben wir die Lokalität der Selektion unterschätzt. Dadurch, dass die Selektionsmenge lokal auf die Nachbarschaft beschränkt ist, wird nur das schlechteste einer kleinen Teilmenge der Population selektiert. Außerdem wird ein neues Individuum wiederum aus der Nachbarschaft erzeugt, wodurch lokale Merkmale stabil bleiben. Laumanns beschreibt, dass der von uns erwartete Effekt, dass die Randbereiche der Pareto-Front nicht abgedeckt werden, tatsächlich auftritt, wenn man den Selektionsradius (die Größe der Nachbarschaft) zu weit ausdehnt.
Außerdem ist zu beobachten, dass die Abdeckung der Pareto-Front Lücken aufweist. Diese können ebenfalls durch die Lokalität der Operatoren erklärt werden. Individuen, die sich stark von ihrer Nachbarschaft unterscheiden, können anfangs nur schwer überleben und später gar nicht mehr entstehen. Eine Nachbarschaft kann demnach nur einen kleinen Bereich der Pareto-Front abdecken. Um eine lückenlose Abdeckung zu erreichen, darf die Varianz (Mutationsschrittweite) nicht zu klein eingestellt sein und die Evolution muss außerdem zu einem günstigen Zeitpunkt abgebrochen werden.

Selektion nach Pareto-Dominanz Unser zweiter Selektionsoperator arbeitet, wie in 5.1.7.1 beschrieben, basierend auf der Pareto-Dominanz. Hierbei zeigt sich, dass die Abdeckung der Pareto-Front zwar keine Lücken aufweist, aber die Ecken der Pareto-Front überhaupt nicht abgedeckt werden. Dies ist genau das Verhalten, das nicht unserer Erwartung entsprach; wir hatten keine intuitive Erklärung für diesen Effekt gefunden. Für das zweite Projektgruppen-Semester war daher eine gründlichere Erforschung dieser Phänomene geplant, auch mit Hilfe weiterer Selektions-Operatoren sowie modifizierter Räuber-Bewegung. Zu diesem Vorhaben ist es jedoch nicht gekommen, da sich die Projektgruppe auf andere Probleme konzentrierte.

5.1.7.4 Verbesserungsmöglichkeiten

Mutationsschrittweite Es wäre sinnvoll, eine dynamische Anpassung der Mutationsschrittweite zu implementieren, damit der Algorithmus zu Beginn mit großer Varianz schnelle Fortschritte erzielen und lokale Optima überwinden kann. Nach etlichen Generationen und damit verbundener Annäherung an die Pareto-Front sollten nur noch minimale Mutationen stattfinden, damit gefundene gute Lösungen nicht mehr zu stark zerstört werden.
Eine einfache Realisierung wäre z. B. eine lineare Verringerung der Varianz im Verlauf der Generationen.

Selektion Wie oben beschrieben, ist die Auswahl des Selektions-Operators entscheidend für die Annäherung an die wahre Pareto-Front und die Diversität der gefundenen Lösungen. Da unsere beiden bisher untersuchten Selektions-Operatoren (jeder Räuber selektiert nach einem Kriterium vs. Selektion aus den dominierten Individuen)

keine befriedigenden Ergebnisse brachten (Lücken in der Pareto-Front vs. geringe Diversität), wäre es interessant, bei eventuellen weitergehenden Forschungen auf diesem Gebiet die folgenden Operatoren zu implementieren und zu untersuchen.

- **Koordinaten-Maximum-Selektion**
  Es wird das Individuum der Nachbarschaft selektiert, dessen Koordinatensumme (die Summe der einzelnen Fitness-Werte) am größten ist.

- **Fronten-Dominanz-Selektion**
  Aus der Nachbarschaft wird das Individuum gewählt, das dominiert wird, aber kein anderes der Nachbarschaft dominiert, also das der hintersten nicht-dominierten Front. Bei mehreren kann aus diesen nach dem Räuber-Kriterium oder zufällig gewählt werden.

- **Dominanz-Laumanns-Selektion**
  Aus der Nachbarschaft werden die innerhalb der Nachbarschaft dominierten Individuen ausgewählt. Von dieser Menge wird das Individuum selektiert, welches den schlechtesten Wert bezüglich des dem Räuber zugeordneten Kriteriums hat.

- **Laumanns-Invers-Selektion**
  Man bestimmt die dominierten Individuen der Nachbarschaft und ignoriert dabei das Räuber-Kriterium. Bei der Überprüfung der Pareto-Dominanz der Individuen wird also jeweils der Wert des aktuellen Räuber-Kriteriums nicht betrachtet. Aus dieser Menge wird das schlechteste Individuum bezüglich des Räuber-Kriteriums gewählt, oder alternativ ein zufälliges.

Nachbarschaft Die Nachbarschaft sollte mit einem Radius versehen werden, so dass alle Individuen mit einem bestimmten Abstand zum aktuellen Knoten zur Nachbarschaft gehören.

Räuber-Anzahl Die Anzahl der Räuber ist im Moment auf jeweils einen pro Optimierungs-Kriterium beschränkt. Für weitere Forschungen wäre es interessant, wenn die Räuber-Anzahl pro Kriterium per Aufrufparameter eingestellt werden könnte.

Räuber-Bewegung Die Räuber bewegen sich zur Zeit mit einer Schrittweite von einem Knoten. Die Schrittweite könnte vergrößert werden, damit ein Individuum nicht vom selben Räuber mehrmals hintereinander angegriffen werden kann. Der Räuber müsste dazu so große Sprünge machen können, dass sich seine jeweiligen Nachbarschaften nicht überschneiden. Dies könnte zu einer erhöhten Diversität führen, da das Selektionskriterium bezogen auf ein bestimmtes Individuum öfter wechselt.

Archiv Die besten Individuen, die ausgegeben werden, werden zur Zeit nur aus der Population der letzten Generation ausgewählt. Es sollte als Verbesserung ein Archiv verwaltet werden, welches immer die bisher besten gefundenen Individuen enthält.

5.1.8 Tabu-Search

Tabu-Search ist ein deterministischer Algorithmus für kombinatorische Probleme, welcher sich in erster Linie für lokale Suchen eignet. Ausführlich wurde dieses Verfahren von Glover und Laguna [13] beschrieben. Es wird hierbei eine klar definierte Nachbarschaft verwendet. Zwei Individuen sind benachbart, wenn sie sich nur durch einen „Schritt“ unterscheiden. Die Nachbarschaft ist im Allgemeinen relativ klein. Tabu-Search wählt dann aus der Nachbarschaft immer den besten Nachbar aus, zu dem der „Schritt“ nicht verboten wurde. Das Verbieten nennt man Tabuisieren. Ein Schritt, der tabu ist, kann dennoch durchgeführt werden, wenn er zu einem Nachbarn führt, dessen Fitness die bisher Beste in dieser Suche ist, welches man Aspiration bzw. Aspirationskriterium nennt. Der genaue Ablauf wird im Aktivitätendiagramm (Abbildung 5) dargestellt.

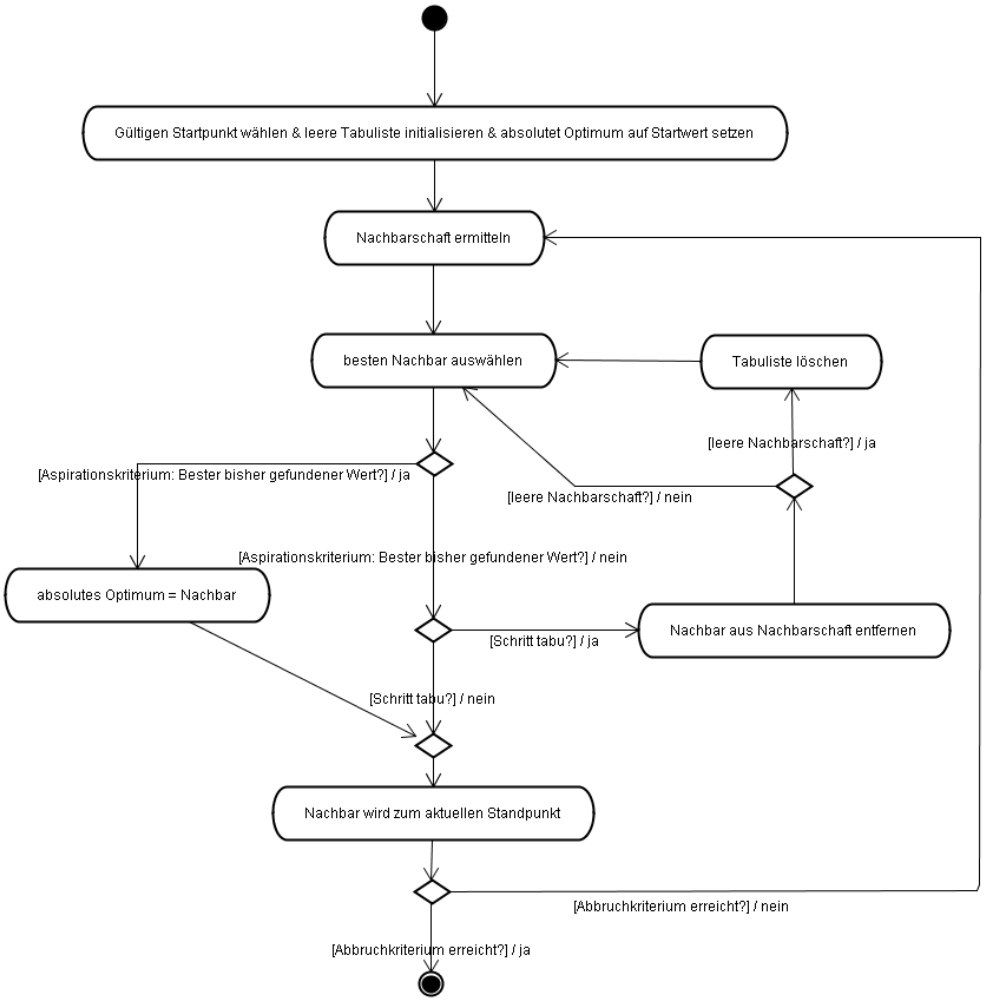

Abbildung 5: Aktivitätendiagramm Tabu-Search

5.1.8.1 Der Algorithmus

Um diese Nachbarschaft zu bekommen, nimmt man meistens das Problem und stellt den Lösungsraum des kombinatorischen Problems durch einen Graphen dar. Ein Knoten in dem Graphen stellt dabei eine Kombination von Lösungsmöglichkeiten dar. Ein Nachbar zu einer Kombination ist durch eine Kante in dem Graphen gekennzeichnet. In dem Fall des Rucksackproblems ist ein Nachbar einer bestimmten Kombination dadurch gekennzeichnet, dass man in einem Schritt zu dieser neuen Kombination kommen kann, d. h. entweder wird eines der n Gegenstände aus der Eingabe in den Rucksack gepackt oder ein Gegenstand, der in dem Rucksack ist, aus diesem wieder hinaus genommen. Grundsätzliches Problem ist bei Tabu-Search einen Konfigurationsgraphen zu finden, der möglichst viele kombinatorische Probleme abdeckt, d. h. einen Konfigurationsgraphen zu finden, der für das Rucksackproblem, Traveling-Salesperson-Problem, Bin Packing und weitere kombinatorische Probleme geeignet ist.
Da dies nicht so einfach zu realisieren ist, haben wir uns auf das mehrkriterielle $0/1$ Rucksackproblem und äquivalente Probleme beschränkt.
Die Größe des Konfigurationsgraphen hängt stark von der Eingabegröße ab. Tabu-Search versucht nun in diesem Konfigurationsgraphen einen guten Weg zu finden, so dass die Konfiguration zulässig ist, d. h. die Summe der Gewichte der eingepackten Gegenstände darf die zulässige Rucksackbepackungsgrenze nicht überschreiten, und die bestmöglichen Nutzenwerte erreicht werden. Ziel dabei ist es, eine gute Bepackung zu finden, ohne den gesamten Graphen zu durchlaufen. Tabu-Search geht dabei deterministisch vor, indem er immer das Objekt in den Rucksack packt, das zu bestmöglicher Nutzenvergrößerung führt bzw. das zu einer geringsten Verschlechterung der Nutzenwerte führt. Dabei kann es schnell zu so genannten Zyklen kommen, so dass ein Gegenstand in den Rucksack gepackt wird, im nächsten Zug wieder herausgenommen wird, in dem darauf folgenden Zug wieder hineingepackt wird, usw. Um dieses zu vermeiden, wird eine so genannte Tabuliste initialisiert, die gewisse Züge (z. B. die gerade gemachten) für eine fest definierte Dauer verbietet. Fraglich ist dabei, wie lang diese Tabuliste maximal sein soll und wie lange ein Zug verboten wird, d. h. in unserem Fall für wie viele Züge ein bestimmter Gegenstand nicht bewegt werden darf.

5.1.8.2 Einstellmöglichkeiten

Der Algorithmus bietet neben den bekannten Parametern noch folgende optionale Startparameter:

- `-maxtabulistlength`: Die maximale Anzahl Züge in der Tabuliste (default=15).
- `-tabuiterations`: Die maximale Verweildauer eines Zuges in der Tabuliste (default=3).
- `-archive`: Dieser optionale Parameter bestimmt die Maximalgröße des globalen Archivs, welches am Ende ausgegeben wird. Der Standardwert ist 0, was einem unbegrenzt grossen Archiv entspricht.

5.1.8.3 Schwachstellen

Tabu-Search führt meist eine relativ lokale Suche durch. Deshalb werden in der Praxis meist viele Aufrufe mit unterschiedlichen Startwerten erzeugt und die Ergebnisse zusammengefasst. Oft verwendet man auch erst sehr lange Tabulisten, um eine hohe Streuung zu erreichen und verwendet die Ergebnisse dann als Startwerte für weitere Suchen mit kürzeren Tabulisten, um die lokalen Optima zu ermitteln. Dies ist derzeit nicht automatisiert und sollte auch nicht direkter Bestandteil des Algorithmus werden. Weiterhin ist dieses bei unserer Implementierung auch manuell nicht möglich, da man den Startwert der Suche nicht selber setzen kann, sondern Grenzen für die Initialisierung von der Testfunktion vorgegeben bekommt.

5.1.8.4 Verbesserungsmöglichkeiten

Zunächst sollte der Algorithmus weiter mit größeren Problemen getestet werden. Danach sollte dann das Setzen eines Startwertes implementiert werden, da dies für den Einsatz von Tabu-Search sehr wichtig ist. Dies ermöglicht dann wiederum die Entwicklung spezieller Strategien zur Anwendung des Algorithmus.
Darauf folgen Optimierungen an der Tabu-Liste. Es kann eine variable Tabu-Dauer implementiert werden. Sinnvoll könnte es auch sein, mehrere Tabulisten zur besseren Vermeidung von Zyklen zu verwenden.
Denkbar wäre auch eine andere Ermittlung der Fitness der Individuen. Hansen [14] und Hertz [15] haben hier beispielsweise Methoden für Tabu-Search vorgestellt, die meist bessere Ergebnisse erzeugen, als bei Verwendung einer (gewichteten) Aufsummierung der Kriterien.
Als größtes Ziel sehen wir die Umsetzung anderer kombinatorischer Probleme, wie beispielsweise dem TSP, so dass auch diese mittels Tabu-Search gelöst werden können.
Im Verlauf der PG hat sich unser Fokus verändert. Während das erste PG-Semester primär Implementierungen beinhaltet hat, haben wir im zweiten PG-Semester die Praxisprobleme in Form der Simulatoren eingebunden und generell nach neuen Ideen und guten Parametern gesucht. Die Ideen zur Weiterentwicklung dieses Algorithmus wurden nicht umgesetzt, da wir auch keine weitere Testfunktion für kombinatorische Probleme implementieren wollten. Andere Klassen von Heuristiken als die kombinatorischen Probleme sahen wir – gerade im Hinblick auf die Simulatoren – als relevanter an.

5.1.9 Modifizierter SMS-EMOA

5.1.9.1 Der Algorithmus

Der ursprüngliche SMS-EMOA wurde am Ls11 von Emmerich, Beume, Naujoks [16] entwickelt und ist mit dem NSGA-II verwandt. Wir haben den Algorithmus um eine zusätzliche Selektionsmethode erweitert.

Beschreibung Bei dem SMS-EMOA handelt es sich um eine (μ+1)-ES. Das heißt, in jeder Generation wird ein Nachkomme (mit Hilfe von Rekombination und Mutation) erzeugt.

Zuerst wird die erste Eltern-Population (Start-Population) erzeugt, die der Implementierung der des `MueRhoLES` (siehe 5.1.4) entspricht.
Aus zwei zufällig ausgewählten Individuen wird durch Rekombination und Mutation ein neues Individuum erzeugt und zur Population hinzugefügt. Die μ+1 Individuen werden nach Fronten sortiert. Es werden zuerst alle Individuen herausgesucht, die nicht dominiert werden. Diese bilden die erste Front. Dann werden die Individuen der ersten Front gedanklich gestrichen und man sucht aus der reduzierten Menge wieder die Individuen, die jetzt nicht-dominiert werden (zweite Front) und so weiter bis alle Individuen einer Front zugeordnet sind. Anschließend soll noch ein Individuum aus der schlechtesten Front entfernt werden. Falls nur eine nicht-dominierte Front vorhanden ist, wird das Individuum mit dem kleinsten Beitrag zum S-Metrik-Wert der Front entfernt (sogenannte S-Metrik-Selektion). Bei mehreren Fronten wird für jedes Individuum der letzten Front überprüft, von wie vielen Individuen es dominiert wird. Das Individuum mit der höchsten Anzahl wird aus der Population entfernt.

Pseudo-Code (μ + 1) ES auf einer Funktion mit zwei Zielen f_1 und f_2

Initiale Population *P*
wiederhole
  erzeuge 1 Nachkommen *ind* durch Rekombination und Mutation
  sortiere *P* und *ind* nach Fronten
  Falls (Anzahl Fronten > 1)
    sortiere (nach Fitnesswerten) k Punkte der letzten Front $q_1, ..., q_k$
    behalte Randpunkte (ersten und letzten der sortierten Sequenz)
    Für alle inneren Punkte i der letzten Front (q_2 bis q_{k-1})
      Für alle Punkte j der vorderen Fronten
        Falls $f_1(q_i) \geq f_1(j)$ & $f_2(q_i) \geq f_2(j)$
          dominiertePunkte[q_i] += 1
        Ende (Falls)
      Ende (Für alle j)
    Ende (Für alle i)
    entferne Punkt mit größter Anzahl dominierender Punkte aus der Population
  sonst
    entferne Individuum mit dem kleinsten Beitrag zur S-Metrik der letzten Front aus der Population
... bis Stopp-Kriterium erreicht

Operatoren Hier werden nun die einzelnen Operatoren des Algorithmus detaillierter beschrieben.

Rekombination und Mutation Die Rekombination und Mutation wurden aus dem C-Code des am LS11 entwickelten SMS-EMOA übernommen. Bei der Rekombination handelt es sich um *Simulated Binary Crossover (SBX)* [18]. Vereinfacht, funktioniert *SBX* so:

- Schritt 1: wähle Zufallszahl u_i aus [0,1)

- Schritt 2: berechne β_{qi}
  $\beta_{qi} = (2u_i)^{\frac{1}{\eta_c+1}}$, wenn $u_i \leq 0.5$
  $\beta_{qi} = (\frac{1}{2(1-u_i)})^{\frac{1}{\eta_c+1}}$ sonst
- Schritt 3: berechne zwei Nachkommen x_i
  $x_i^{(1,t+1)} = 0.5[(1+\beta_{qi})x_i^{(1,t)} + (1-\beta_{qi})x_i^{(2,t)}]$,
  $x_i^{(2,t+1)} = 0.5[(1-\beta_{qi})x_i^{(1,t)} + (1+\beta_{qi})x_i^{(2,t)}]$

Bei $x_i^{(1,t)}$ bzw. $x_i^{(2,t)}$ handelt es sich um die i-te Variable des ersten bzw. zweiten Individuums der Generation t (hier Eltern-Generation). Bei $x_i^{(1,t+1)}$ handelt es sich also um die i-te Variable des ersten Individuums der Generation $t+1$ (bzw. Nachkommen-Generation). β_{qi} stellt den Verteilungsfaktor für die Variablen dar und wird durch die Zufallszahl u_i und den Parameter η_c berechnet. Der Wert von η_c (eine nicht-negative reelwertige Zahl) beeinflusst die Ähnlichkeit der Nachkommen zu den Eltern. Bei großen η_c-Werten ist die Wahrscheinlichkeit, dass die Variablen der Nachkommen zu den Eltern ähnlich sind, groß. D. h.: Individuen, die im Vergleich zu einem Elter leicht oder gar nicht verändert sind, sind wahrscheinlich. Nachkommen, die mittig zwischen den Eltern oder die weit von Eltern weg liegen, sind unwahrscheinlich. Dieser Effekt ist bei großen η_c-Werten deutlicher als bei kleineren. Aus zwei zufällig ausgewählten Individuen werden zwei Nachkommen erzeugt. Da wir nur einen benötigen, wird ohne Beschränkung der Allgemeinheit das Individuum $x_i^{(1,t+1)}$ ausgewählt.
Für jede Variable der Nachkommen wird zufällig entschieden, ob sie mutiert wird. Die Wahrscheinlichkeit ist der Kehrwert der Dimension, wobei die Dimension die Anzahl der Objektvariablen ist. Diese wird bei der Testfunktion abgefragt. Noch vor der Mutation werden für jede Variable x_i die Ober- und Untergrenze bei der Testfunktion abgefragt (x_u und x_l). Die Variablen bleiben nach der Mutation innerhalb dieser Grenzen. So können keine ungültige Individuen erzeugt werden.
Mutation der Variablen falls $x_i > x_l$:

- Schritt 1: wähle Zufallszahl u_i aus [0,1)
- Schritt 2: berechne δ
  $\delta = \frac{x_i - x_l}{x_u - x_l}$ falls $(x_i - x_l) < (x_u - x_i)$
  $\delta = \frac{x_u - x_i}{x_u - x_l}$ sonst
- Schritt 3: berechne δ_q
  $\delta_q = (2u_i + (1-2u_i)(1-\delta)^{21} - 1)^{\frac{1}{21}} - 1$ falls $u_i \leq 0.5$
  $\delta_q = 1 - (2u_i + (1-2u_i)(1-\delta)^{21} - 1)^{\frac{1}{21}}$ sonst
- Schritt 4: berechne neues x_i
  $x_i = x_i + \delta_q(x_u - x_l)$
- Schritt 5: x_i mit der Ober- und Untergrenze vergleichen und ggf. anpassen:
  $x_i = x_l$ falls $x_i < x_l$
  $x_i = x_u$ falls $x_i > x_l$

Mutation der Variablen falls $x_i = x_l$:

- Schritt 1: wähle Zufallszahl u_i aus [0,1)

- Schritt 2: berechne neues x_i:
  $x_i = u_i(x_u - x_l) + x_l$

Selektion Nach der Rekombination und Mutation wächst unsere Population auf $\mu + 1$ Individuen. Von diesen Individuen muss jetzt eins selektiert (gelöscht) werden, damit die Population wieder auf die Größe μ schrumpft. Nach der Sortierung der Individuen nach Fronten wird die Anzahl der Fronten überprüft. Falls nur eine Front vorhanden ist, wird die S-Metrik als Selektionskriterium verwendet (S-Metrik-Selektion). Die S-Metrik misst die Größe des Raums, der von den Individuen dominiert wird. Im zwei-dimensionalen Fall ist das also einfach eine Fläche, allgemein ein Hypervolumen. Wir löschen das Individuum, welches den kleinsten Beitrag zum Hypervolumen leistet. Das heißt anders herum, dass wir die Individuen behalten, die den Wert des Gesamt-Volumen der letzten Front maximieren.
Sonst betrachtet man die letzte (schlechteste) Front. Bei dem ursprünglichen SMS-EMOA würde auch jetzt die S-Metrik-Selektion verwendet werden. In der neuen Selektionsmethode wird für jedes Individuum der letzten Front überprüft, von wie vielen Individuen es dominiert wird. Das Individuum mit der höchsten Anzahl wird selektiert und entfernt.
Der modifizierte SMS-EMOA (`SMS-EMOA_pg`) bietet folgende Möglichkeiten zur Selektion, wobei der Parameter `-sm` für die Selektionsmethode steht:

- sm = 0 (die von uns entwickelte Selektionsmethode)
  Falls es zwei oder mehr Fronten gibt, wird für jedes Individuum in der schlechtesten Front die Anzahl der Punkte gezählt, die dieses dominieren. Das Individuum mit der größten Anzahl wird entfernt. In diesem Fall beträgt die Laufzeit $O(\mu^2 k)$ pro Generation, wobei μ für die Populationsgröße und k für Anzahl der Zielfunktionen steht.
  Falls nur eine Front vorhanden ist, wird die S-Metrik-Selektion eingesetzt und sie hat eine Laufzeit von $O(\mu^3 k^2)$.

- sm = 1 (zufällige Wahl)
  Falls es zwei oder mehr Fronten gibt, wird aus der schlechtesten Front ein Individuum zufällig entfernt.
  Falls nur eine Front vorhanden ist, wird die S-Metrik-Selektion eingesetzt.

- sm = 2 (original SMS-EMOA mit Randomisierung-Verfahren).
  Nur die S-Metrik-Selektion wird eingesetzt.

Zur besseren Robustheit haben wir die Selektionsmethoden sm = 0, sm = 1 und sm = 2 um zwei gängige Verfahren erweitert:

- **Extrempunkte / Randpunkte behalten**
  Individuen, die in einer Zielfunktion den besten (kleinsten) Wert haben, werden als Extrempunkte bezeichnet. Sie liegen sozusagen am Rand der Population, wenn man diese im Raum der Fitnesswerte darstellt. Diese Individuen werden vor der Selektion geschützt, um eine möglichst große Diversität zu erhalten. Eine Ausnahme ist sm = 1, der Fall der zufälligen Wahl. Hier kann jedes Individuum selektiert werden.

- **Randomisierung bei Unentscheidbarkeit**
  Falls mehrere Individuen bezüglich unserer Kriterien den gleichen Wert haben (also gleiche Anzahl dominierender Punkte bzw. gleichen Beitrag zum S-Metrik-Wert), dann wird aus diesen eines zufällig gewählt. Zuvor war es so, dass das letzte Individuum der schlechtesten ausgewählt wurde.

Die folgenden Varianten wurden nur zu Forschungs-Zwecken implementiert:

- sm = 3 (wie sm = 0 aber ohne Randomisierung und ohne Randpunkte behalten)
- sm = 4 (wie sm = 0 aber ohne Randomisierung bei Unentscheidbarkeit)
- sm = 5 (wie sm = 0 aber ohne Extrempunkte / Randpunkte behalten)

5.1.9.2 Einstellmöglichkeiten

Der Algorithmus braucht folgende exogene Startparameter:

- die Anzahl der Individuen in der Eltern-Generation μ (Parameter `-mu`),
- die Selektionsmethode (Parameter `-sm`),
- die Anzahl Genrationen (Parameter `-g`),
- die Testfunktion (Parameter `-t`),
- evtl. Randomseed (Parameter `-r`).

5.1.9.3 Beobachtungen

Der Algorithmus wurde im zweiten Semester weiter entwickelt und anschließend implementiert. Der SMS-EMOA war den etablierten Algorithmen, wie NSGA-II oder SPEA2, auf der Familie der ZDT-Funktionen schon in der Originalfassung überlegen. Dies wurde durch die Experimente auf den ZDT-Funktionen (siehe 7.2) bekräftigt. Die modifizierte Version brachte zusätzlich Vorteil im Rechenaufwand. Die Qualität der modifizierten Version des Algorithmus wurde auch durch die Experimente mit dem Fahrstuhlsimulator (siehe 7.4.2) bestätigt.

5.2 Metriken

Bewertungsfunktionen, die u.a. auf Ergebnisse von evolutionären Algorithmen angewendet werden, werden *Metriken* genannt. Für die Bewertung von mehrkriteriellen Lösungsmengen wurden erst in den letzten Jahren Metriken entwickelt. Qualitätsmaße für mehrkriterielle Lösungsmengen sind hilfreich, da zwischen einzelnen Kriterien typischerweise Trade-off-Effekte bestehen und daher der Vergleich von Lösungsmengen nicht trivial ist.

Die Menge von Metriken wird in drei Untermengen geteilt (siehe [22]), nämlich Metriken für den Vergleich:

- der Diversität;

- der Annäherung an die Pareto-Front;
- und der Maximierung der Anzahl nicht-dominierter Lösungen.

Es existiert keine Metrik, die eindeutig alle Vergleichsmöglichkeiten erfüllt. Die PG447 hat sich für das Projekt *NObELTJE* für drei Metriken entschieden: Die Euklid-Metrik (siehe 5.2.1), die S-Metrik (beschrieben in Kapitel 5.2.2) und die C-Metrik (siehe Kapitel 5.2.3).

In einer Arbeit von Knowles [23] werden die Metriken von Zitzler und anderen Entwicklern beschrieben und verglichen. Außerdem werden die benutzten Metriken in einer Seminar-Ausarbeitung teilweise beschrieben, die im Anhang des Zwischenberichts zu finden ist. In den erwähnten Dokumenten werden auch die zur Beschreibung der Metriken benutzten Begriffe wie Dominanz, Outperformance-Relation und Kompatibilität erklärt. Der Begriff „übertrifft“ bedeutet, dass eine Menge A besser als eine Menge B ist (bezüglich eines Minimierungsproblems), d. h. $A \prec B$.

Die drei Outperformance-Relationen beschreiben differenziert das Verhältnis zweier Mengen. Die Kompatibilität einer Metrik mit einer Outperformance-Relation bedeutet, dass die Metrik das entsprechende Verhältnis der Mengen korrekt bewertet.

schwache Outperformance-Relation O_w: Die Elemente der Mengen sind unvergleichbar, aber die bessere Menge hat mindestens ein Element mehr als die andere.

starke Outperformance-Relation O_s: Die bessere Menge dominiert mindestens ein Element der anderen Menge.

komplette Outperformance-Relation O_c: Die bessere Menge dominiert alle Elemente der schlechteren.

Knowles beschreibt weitere Eigenschaften der Metriken, die in folgenden Metrikbeschreibungen benutzt werden:

Ringschlussbeweis: Wenn eine Metrik eine Lösungsmenge „A besser als B“ bewertet und „B besser als C“, dann kann „A besser als C“ geschlossen werden.

Skalierung: Hier wird beschrieben, ob eine Metrik ab- oder unabhängig von der Skalierung des Raumes der Lösungen ist.

kardinale Messung: Eine kardinale Messung zählt z. B. die nicht-dominierten Elemente.

Referenz-Menge/-Punkt: Wenn die Metrik zur Bewertung einen zusätzlichen Punkt benötigt, oder eine weitere Menge z. B. die wahre Pareto-Front des Problems.

5.2.1 Euklid-Metrik

Die Euklid-Metrik berechnet die durchschnittliche Entfernung der berechneten Lösungsmenge von der „wahren“ Pareto-Front, also den gemittelten euklidischen Abstand jedes Punktes zu einem Pareto-optimalen Punkt. Die Euklid-Metrik benutzt also

Elemente der „wahren“ Pareto-Front als Referenz. Bei Problemen mit unbekannter Pareto-Front kann diese Metrik demnach nicht angewendet werden. Abbildung 6 zeigt ein illustratives Beipiel der „wahren“ Pareto-Front und der Lösungsmenge (in der Abbildung 6 als „berechnete Lösungen“ bezeichnet) im Zielraum.

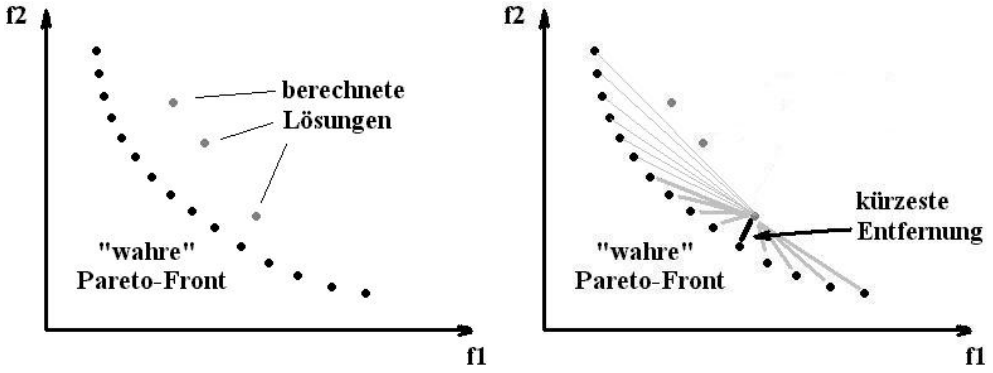

Abbildung 6: Berechnung des kleinsten Abstandes einer Lösung zu den Punkten der Referenzmenge („wahre“ Pareto-Front).

Die Testfunktion liefert die wahre Pareto-Front (sofern diese bekannt ist) über die Methode `writeParetoPoints` in Form einer Textdatei. In einer zweiten Textdatei befinden sich vom Algorithmus berechnete, nicht-dominierte Lösungen. Die beiden Dateien werden ausgelesen und der Abstand der Fronten wird ermittelt.

Die Entfernungen zwischen einem berechneten Vektor und allen Vektoren aus der „wahren“ Pareto-Front werden berechnet. Die kleinste Entfernung wird gespeichert und die anderen werden verworfen. Mit den restlichen Elementen der Lösungsmenge wird analog verfahren. Aus der Summe der minimalen Entfernungen wird das arithmetische Mittel berechnet, indem durch die Anzahl der Elemente der zur Bewertung betrachteten Lösungsmenge geteilt wird.

Bei n Vektoren der „wahren“ Pareto-Front müssen für jeden Lösungsvektor n Abstände berechnet werden. Bei m Lösungsvektoren müssen also für die Ermittlung der kürzesten Abstände insgesamt n mal m Berechnungen des Euklid-Abstands zweier Vektoren des R^k durchgeführt werden. Die Laufzeit beträgt also insgesamt $O(nmk)$.

Die Metrik ist mit der Outperformance-Relation O_w nicht kompatibel, jedoch mit O_s und O_c. Ein direkter Ringschluss ist mit dieser Metrik nicht möglich, da sie nicht zwei Lösungsmengen vergleicht, sondern nur eine Lösungsmenge bewertet. Durch drei Metriken-Aufrufe können allerdings die Bewertungen der Lösungsmengen miteinander in Beziehung gesetzt werden, vorausgesetzt, dass bei den Berechnungen die selbe Referenzmenge benutzt wurde. Je niedriger der Metrikenwert ist, desto besser ist die Lösungsmenge. Die Metrik ist unabhängig von der Skalierung des Lösungsraums. Die Euklid-Metrik benutzt keine „Zählmessung“, da die Metrik einen durchschnittlichen Abstand berechnet und keine Vektoren zählt.

5.2.2 S-Metrik

Die S-Metrik berechnet den Raum, der durch die nicht-dominierten Elemente einer Lösungsmenge dominiert und durch einen Referenzpunkte abgegrenzt wird. Die Größe dieses Raumes ist der Wert der Metrik. Zitzler entwickelte diese Metrik 1999 [24].

Die S-Metrik benötigt für die Wahl des Referenzpunkts die obere Grenzen der Funktionswerte der benutzten Testfunktion. Der Referenzpunkt soll außerhalb des Wertebereiches der Lösungen liegen, damit er von allen Elementen der Lösungsmenge dominiert wird. Bei praktischen Problemen kann der Referenzpunkt schwer zu bestimmten sein, bei Testfunktionen stellt die Wahl dagegen eher kein Problem dar.

Die Metrik ist mit allen drei Outperformance-Relationen kompatibel. Ein Ringschluss ist mit dieser Metrik nicht möglich, da die Größe des dominierten Raumes auch von einem Referenzpunkt abhängig ist. Wenn man also die Größe verschiedener Algorithmen mit verschiedenen Referenzpunkten vergleicht, kann dies zu falschen Ergebnissen führen. Die Metrik ist unabhängig von der Skalierung der Elemente, da durch Anpassung nur die Fläche verkleinert / vergrößert wird. Dies ist aber zum Vergleich mit einer anderen Metrik nicht entscheidend, solange der Referenzpunkt gleich bleibt. Die S-Metrik benutzt keine „Zählmessung“ zur Bestimmung der Größe des dominierten Raumes und berechnet eine Fläche.

Für den Fall von zwei Zielfunktionen wurde eine effiziente Implementierung mit der Klasse `NewSMetric` realisiert. Bei mehr als zwei Zielfunktionen wird die komplexere Berechnung mit Hilfe des Algorithmus von Fleischer [17] durchgeführt (`SMetric-Fleischer`). Der Beitrag eines bestimmten Punktes zum Gesamtwert der S-Metrik kann mit der Klasse `DeltaS` berechnet werden. Die verschiedenen Varianten werden im Folgenden beschrieben.

5.2.2.1 S-Metrik für zwei Zielfunktionen

Zur Berechnung der S-Metrik für den zwei-dimensionalen Fall wird der spezielle effiziente Algorithmus `NewSMetric` benutzt. Argumente der Berechnungsmethode sind die Elemente der Lösungsmenge und ein Referenzpunkt (*RefPoint* in Abbildung 7). Der Referenzpunkt ist standardmäßig auf die Koordinaten (10,0; 10,0) eingestellt, kann aber mit der Methode `setRefPoint` geändert werden. Diese Koordinaten liegen für die benutzten Testfunktionen außerhalb des Wertebereichs.

Die Elemente werden vor der ersten Berechnung bezüglich der Zielfunktion f_1 absteigend sortiert. Die Fläche ergibt sich durch die Berechnung der Summe aus den Flächen jedes einzelnen Elementes der Lösungsmenge und des Referenzpunktes (siehe Abbildung 7). Nach der Berechnung der Fläche zwischen dem ersten Element aus der Lösungsmenge und dem Referenzpunkt, wird die y-Koordinate des Referenzpunktes für die Berechnung der nächsten Teilfläche auf die y-Koordinate des ersten Punktes gesetzt (siehe Abbildung 7). Diese Anpassung erfolgt vor jeder weiteren Berechnung einer Teilfläche. Die Teilflächen werden addiert und ausgegeben. Die Berechnung der

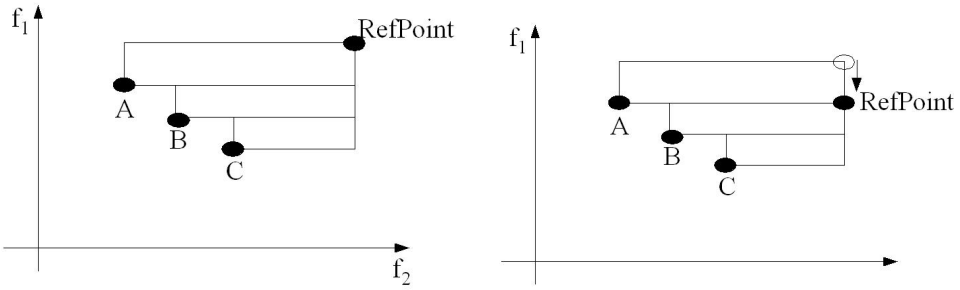

Abbildung 7: Die linke Abbildung zeigt, wie die S-Metrik an Hand des Referenzpunktes (*RefPoint*) und der Elemente aus der Lösungsmenge (*A*, *B*, *C*) die dominierte Fläche bildet. Die rechte Abbildung zeigt die Verschiebung des Referenzpunktes nach der Flächen-Berechnung des ersten Punktes.

Fläche im zweidimensionalem Raum erfolgt durch die Formel:

$$\sum_{i \in 1...|A|} \mid z_1^{ref} - z_1^i \mid * \mid z_2^{ref} - z_2^i \mid$$

$|A|$ entspricht der Anzahl der Elemente in der Lösungsmenge, z_1^{ref} *und* z_2^{ref} sind die Koordinaten des Referenzpunktes und z^i ist ein Element aus der Lösungsmenge.
Die Berechnungskomplexität des Algorithmus beträgt nur $O(m * logm)$. Sie wird bestimmt durch die Sortierung der Lösungen, die $O(m*logm)$ beansprucht. Die Berechnung der Teilflächen, sowie die Verschiebung des Referenzpunkts sind Operationen von geringerer Größenordnung.

5.2.2.2 S-Metrik für multi-dimensionale Zielfunktionen

Die Klasse `SMetricFleischer` enthält die Berechnung der S-Metrik nach dem Algorithmus von M. Fleischer [17]. Diese Implementierung ist im Gegensatz zu der Klasse `NewSMetric` für eine beliebige Anzahl von Zielfunktionen einsetzbar. Die Laufzeit des Algorithmus beträgt O(m^3n^2), wobei m die Anzahl der nicht-dominierten Punkte und n die Anzahl der Zielfunktionen beschreibt.
Die Klasse `SMetricFleischer` benötigt einen Referenzpunkt, der entweder über eine Methode zugewiesen werden kann, oder zur Berechnung des S-Metrikwertes automatisch auf einen Punkt mit möglichst kleinen Koordinaten gesetzt wird. Lösungen, die größere Funktionswerte als der Referenzpunkt haben, werden bei der Berechnung nicht betrachtet, genau wie dominierte Lösungen.

Der Betrag des S-Metrikwertes jedes Punktes (Hypervolumen) wird dann wie folgt berechnet. Zu einem Punkt x werden seine benachbarten Punkten bestimmt, das heißt, für jede Zielfunktion ein Punkt, der den nächst schlechteren Wert hat. Die Abstände zu den Nachbarn werden multipliziert und bilden damit einen Hyperkubus, der von dem Punkt x dominiert wird. Anschließend werden an den Ecken des Hyperkubus neue Punkte generiert (*spawn points*) und der abgearbeitete Punkt x wird aus der Datenstruktur entfernt. Die neu erzeugten Punkte werden der Datenstruktur hinzugefügt, falls sie

nicht dominiert werden. Man fährt nun die Berechnung analog mit einem neu erzeugten Punkt fort. Falls keine neuen nicht-dominierten Punkte generiert wurden, wird mit den Original-Punkten der Datenstruktur weitergerechnet, bis diese vollständig geleert ist. Anschaulich beschrieben, wird das dominierte Hypervolumen nach den durch die nicht-dominierten Punkte definierten Kuben iterativ abgetragen.

5.2.2.3 Beitrag eines Punktes zum S-Metrik-Wert

Die Klasse `DeltaSMetric` ist ein Werkzeug des Algorithmus `SMS_EMOA_pg`. Sie liefert für eine Punktmenge und einen darin enthaltenen bestimmten Punkt den Beitrag des Punktes zum S-Metrik-Wert der Gesamtmenge. Das ist die Größe des Hypervolumens, das ausschließlich von diesem Punkt dominiert wird. Die Berechnung des Hypervolumenbeitrags erfolgt mit Hilfe der Klasse `SMetricFleischer`, wobei die Bestimmung des Hypervolumens mit dem gewählten Punkt beginnt und abbricht, sobald die Größe des Kubus errechnet wurde.

5.2.3 C-Metrik

Eine weitere Metrik, die wir realisiert haben, ist die C-Metrik. Die C-Metrik benötigt zwei Lösungsmengen A und B. Es wird bewertet, wie stark die eine Approximation die andere dominiert. Sie bildet das geordnete Paar (A,B) auf das Intervall [0,1] ab.
Zur Berechnung werden alle Elemente der Lösungsmengen betrachtet und miteinander verglichen. Dazu überprüft man für jedes Element der einen, ob es durch ein Element der anderen Menge schwach dominiert wird. Es wird die Anzahl der dominierten Punkte der Menge gezählt und danach durch die Gesamtanzahl der Punkte der Menge geteilt. Als Ausgabe erhält man eine reelle Zahl. Dabei muss man beide Anordnungen C(A, B), C(B, A)) betrachten, da C(A, B) nicht gleich 1-C(B, A) ist. C ist eine kardinale Messung und durch eine direkte, vergleichende Bewertung bekommt man einen einzigen Gütefaktor, der nicht symmetrisch ist.

Es ist schwierig zu begründen, ob die Metrik eine komplette Ordnung hervorruft, weil nicht klar ist, wie das Paar von C-Werten zusammen interpretiert werden soll. Die nicht-symmetrische Natur der C-Metrik erschwert ihre Analyse zur Kompatibilität zu den Outperformance-Relationen. Es hängt davon ab, wie die beiden Ausgaben interpretiert oder kombiniert werden. Es ist allgemein möglich, zwischen C(A, B) und C(B, A) zu unterscheiden. Gilt C(A, B)=1 und C(B, A) < 1 wird die Menge A besser als B bewertet. Dann ist C mit der schwachen Outperformance-Relation kompatibel. Es gibt aber auch Fälle wo dies nicht der Fall ist (siehe Abbildung 8). Meistens gilt, dass die C-Metrik uns Ausgaben liefert, die nicht mit unserer Intuition zur Bewertung über die Qualität zweier Mengen übereinstimmt, ausgenommen die zwei Mengen enthalten sehr gleichmäßig verteilte Punkte und sind von gleicher Kardinalität. Die Berechnung lautet folgendermaßen:

$$C(A,B) = \frac{|b \in B | \exists a \in A : a \preceq b|}{|B|}$$

Der Wert C(A,B)=1 bedeutet, dass alle Entscheidungs-Vektoren in B durch die aus A schwach dominiert werden. Das Komplement, nämlich der Wert C(A,B)=0, repräsen-

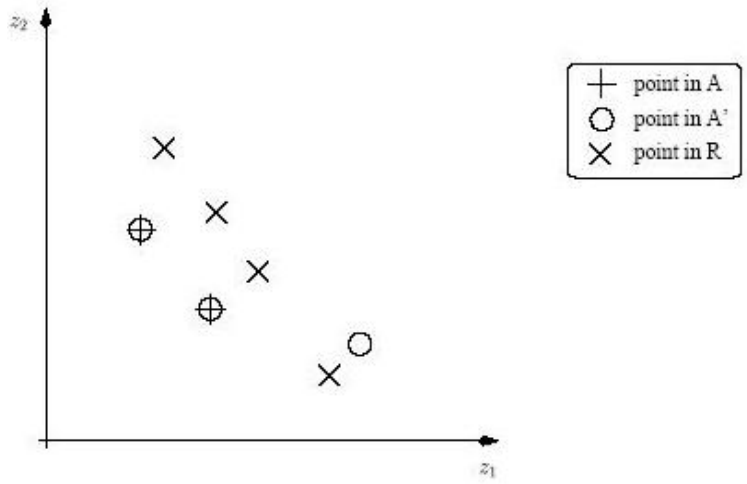

Abbildung 8: Folgende Faktoren werden berechnet: C(R,A)=0/2 = 0, C(R,A')=1/3, C(A,R)=3/4, C(A',R)=3/4. Gegenüber der Referenzmenge R wird A' schlechter als A bewertet. Jedoch *übertrifft* (siehe 5.2) A' A schwach, so dass man daraus schließen kann, dass die C-Metrik nicht schwach kompatibel mit der schwachen Outperfomance-Relation ist, wenn man eine Referenzmenge R benutzt.

tiert die Situation, wo keine Punkte in B durch Punkte aus A schwach dominiert werden.

Die Metrik ist im allgemeinen nicht mit der schwachen Outperformance-Relation kompatibel. Wenn die Verteilung sehr ungleichmäßig ist, gibt es unzuverlässige Ergebnisse. Außerdem kann der Grad der Outperformance-Relation nicht festgelegt werden, wenn eine Menge die andere komplett *übertrifft* (siehe Abschnitt 5.2). Die Metrik ist aber kompatibel zur starken und kompletten Outperformance-Relation.

Ringschlüsse sind im Allgemeinen nicht gültig. Werden drei Mengen betrachtet, kann es vorkommen, dass B besser als A, C besser als B, aber A besser als C bewertet wird.

Der Gütefaktor der Metrik ist abhängig von den gezählten nichtdominierten Punkten. Sind die zu vergleichenden Mengen von verschiedener Kardinalität, werden falsche Schlüsse aus den Ergebnissen gezogen.

Die C-Metrik benötigt keinen Referenzpunkt. Außerdem ist sie von der Pareto-Front unabhängig. Die Laufzeit ist im Vergleich zur S-Metrik gering. Die Unabhängigkeit von der Skalierung ist gegeben.

In unserer Implementation berechnen wir C(A,B), wie auch C(B,A). Sind beide Werte gleich, wird ein Fehler ausgegeben, da zwar beide Mengen als gleich angesehen werden, es aber doch sein kann, dass eine doch näher an der Pareto-Front liegt. Weitere Fälle, wo unseriöse Ergebnisse entstehen, wenn z.B. die Punkte sehr weit verteilt sind, könnten noch abgefangen werden, um den Anwender zu warnen.

5.3 Analyse: Visualisierung und Statistik

Es stehen diverse Klassen bereit, um die Ergebnisse der Design-Läufe grafisch darzustellen und auszuwerten. Zum einen ist dies eine Gruppe von Plottern, welche *Gnuplot* benutzt, um Metrik-Werte zu visualisieren. Die Darstellungen reichen hier von einfachen Balkendiagrammen bis hin zu Flächen im dreidimensionalen Raum.

Neben den Plottern sind weitere Klassen zur Visualisierung und Auswertung entwickelt worden, auf die im Anschluss noch näher eingegangen wird. Zum einen der `AnimatedGifEncoder`, mit dessen Hilfe sich aus mehreren Einzelbildern kleine Animationen in Form von animierten GIFs generieren lassen. Außerdem existiert ein Tool, welches Ergebnisse für das Temperierbohrungs-Problem (siehe 5.4.3) darstellt, der `BohrVisualizer`. Schließlich wurden mehrere Klassen implementiert, die, ähnlich wie die Plotter für *Gnuplot*, Skripte generieren und in *R* ausführen, um Scatterplots, Histogramme oder Boxplots für die Daten zu erstellen.

5.3.1 Übersicht über die Plotter

Mit den Plotter-Klassen stehen vier Tools zur Verfügung, mit denen sich die Ergebnisse eines Designaufrufs grafisch darstellen lassen, nachdem sie durch eine Metrik bewertet wurden.

Das Vorgehen ist dabei bei allen Plottern gleich. Als Parameter gibt man ein Verzeichnis für die Ausgangsdaten an. Der jeweilige Plotter liest alle `*.res` Dateien aus dem spezifizierten Verzeichnis ein und bewertet sie durch eine Metrik, die ebenfalls frei über Parameter wählbar ist. Schließlich wird Gnuplot automatisch gestartet und stellt die Metrikergebnisse einerseits grafisch auf dem Bildschirm dar, andererseits werden sie in einer Grafikdatei gespeichert, die ebenfalls angegeben werden kann.

5.3.2 Allgemeine Einstellmöglichkeiten der Plotter

Neben einigen speziellen Parametern der einzelnen Plotter gibt es einige allgemeine Parameter, die für jeden Plotter gültig sind:

- `-d` <`Verzeichnis`>: Dem Plotter muss beim Aufruf notwendigerweise das Verzeichnis übergeben werden, in dem sich die zu verarbeitenden Daten befinden. Dabei werden alle Dateien mit der Endung „`.res`“ einem Algorithmen-Lauf zugeordnet (z. B.: `-d MopsoOne-Ergebnisse/`).
- `-m` <`Metrikname`>: Alle Algorithmen werden mit der hier bestimmten Metrik bewertet. Daher ist auch dieser Parameter notwendig, bei dem der Metrik-Paketname nicht angegeben werden muss (z. B.: `-d SMetricFleischer`).
- `-o` <`Präfix`>: Um die Ausgaben mehrerer Aufrufe eines Plotters unterscheiden zu können, kann mit diesem Parameter der erste Namensbestandteil der Ausgabedateien gewählt werden. Da es abhängig vom verwendeten Plotter einen Standardwert für diesen Parameter gibt, muss er nicht angegeben werden (z. B.: `-o Mopso-Test-1`).

- `-r1` x_1`, -r2` x_2`, ..., -rn` x_n: Mit diesen Parametern ist es möglich, den für die *S-Metriken* benötigten Referenzpunkt manuell zu wählen. Sie sind nur möglich und dann optional, wenn eine *S-Metrik* verwendet wird. Die Koordinate jeder der n Dimensionen des Zielraums muss einzeln gesetzt werden und folgende Syntax einhalten: „`-r1 X -r2 Y -r3 Z`“. Mit diesem Aufruf wird der Referenzpunkt (X, Y, Z) für einen drei-dimensionalen Zielraum gesetzt. Die Anzahl der Parameter muss natürlich mit der Dimension der zu verarbeitenden Ergebnis-Daten übereinstimmen. Diese ist jedoch beliebig (z. B.: `-r1 7 -r2 10`).
  Werden diese Parameter nicht angegeben, so wird der Referenzpunkt automatisch berechnet, indem aus allen zu betrachtenden Punkten für jede Dimension automatisch die höchste Koordinate gesucht und um eins erhöht wird. Auf diese Weise entstehen nur Metrikwerte größer gleich eins.

5.3.3 MetricsPlotter

5.3.3.1 Beschreibung

Der `MetricsPlotter` dient zur Erstellung einfacher 2-D Balkendiagramme, die dazu dienen, eine Übersicht über die einzelnen Algorithmenaufrufe zu schaffen und deren Effektivität grafisch darzustellen.

5.3.3.2 Einstellmöglichkeiten

Der Plotter verarbeitet außer den allgemeinen Parametern folgende Kommandozeilenparameter:

- `-ggobi`: Diese parameterlose Option aktiviert die Ausgabe in einem zu *GGobi* kompatiblen Format. Sie besteht aus zwei Dateien. Eine hat die Endung „`.dat`“. Diese enthält in jeder Zeile einen Fitnesswert und die Parameterwerte des entsprechenden Algorithmenaufrufs. Die andere Datei endet mit „`.col`“. In ihr stehen die Bezeichnungen zu den Spalten der „`.dat`“ Datei.

- `-sort <Sortierung>`: Hier kann angegeben werden, nach welchem Parameter sortiert werden soll. Bei der Angabe von mehreren Parametern wird in der angegebenen Reihenfolge sortiert. Die einzelnen Parameter werden durch Komma getrennt. So sortiert z. B. `-sort g,variance,archive` erst nach `g`, dann nach `variance` und schließlich nach `archive`.

5.3.4 MeanMetricPlotter

5.3.4.1 Beschreibung

Der `MeanMetricPlotter` visualisiert die Metrikergebnisse eines Designaufrufs in Abhängigkeit von einem Parameter in einem 2-dimensionalen Diagramm (siehe Abbildung 10). Beim Aufruf dieses Plotters muss ein Parameter des Algorithmus angegeben werden, der in den Versuchen variiert wurde. Nach diesem werden die Metrikwerte sortiert dargestellt. Gibt es mehrere Dateien zu einer Einstellung dieses Parameters,

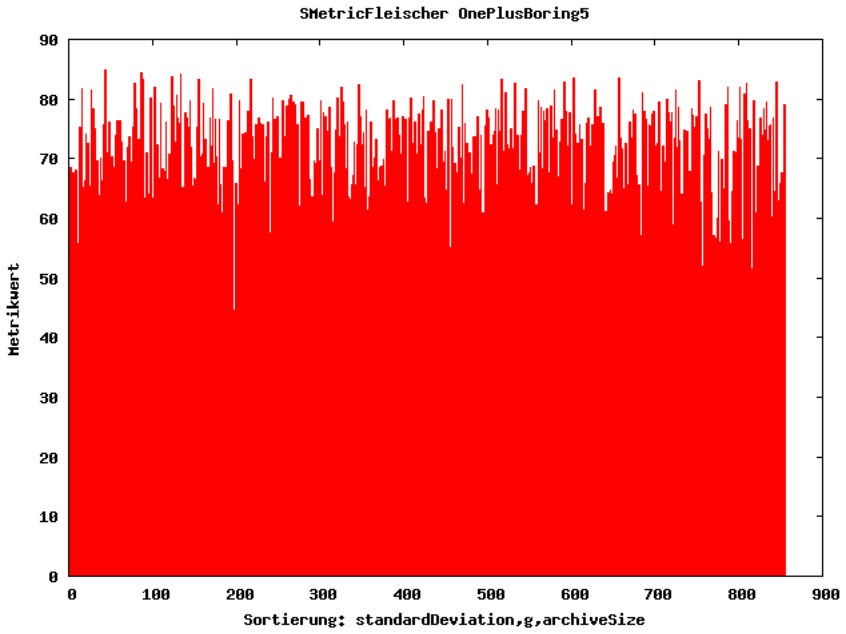

Abbildung 9: Beispiel Balkendiagram für den `MetricsPlotter`

so wird der Durchschnitt der entsprechenden Werte gebildet. Auf diese Weise können z. B. Mittelwerte über eine Reihe verschiedener Random-Seeds gebildet werden. Die entstehenden Grafiken sind dadurch aussagekräftiger und leichter zu interpretieren.

5.3.4.2 Einstellmöglichkeiten

Der Plotter verarbeitet außer den allgemeinen Parametern folgende Kommandozeilenparameter:

- `-p1 <Parametername>`: Der Plotter fasst die Ergebnisse aller Algo-Läufe zusammen, die den gleichen Wert bei dem hier angegebenen Parameter haben. Dateien, die diesen Parameter nicht im Namen tragen, werden ignoriert. Die Ergebnisse werden nach diesem Parameter entlang der X-Achse aufgetragen (z. B.: `-p1 archive`).

- `-t <Testfunktion/Simulator>`: Gibt die verwendete Testfunktion an. Dieser Parameter ist nur möglich und dann auch notwendig, wenn die *Euklid-Metrik* (Kapitel 5.2.1) verwendet wird, da bei dieser die tatsächliche Paretofront benötigt wird (z. B.: `-t testfkt.DoubleParabola`).

- `-sort <Sortierung>`: Hier kann angegeben werden, nach welchem Parameter sortiert werden soll. Bei der Angabe von mehreren Parametern wird in der angegebenen Reihenfolge sortiert. Die einzelnen Parameter werden durch Komma getrennt. So sortiert z. B. `-sort g,variance,archive` erst nach `g`, dann nach `variance` und schließlich nach `archive`.

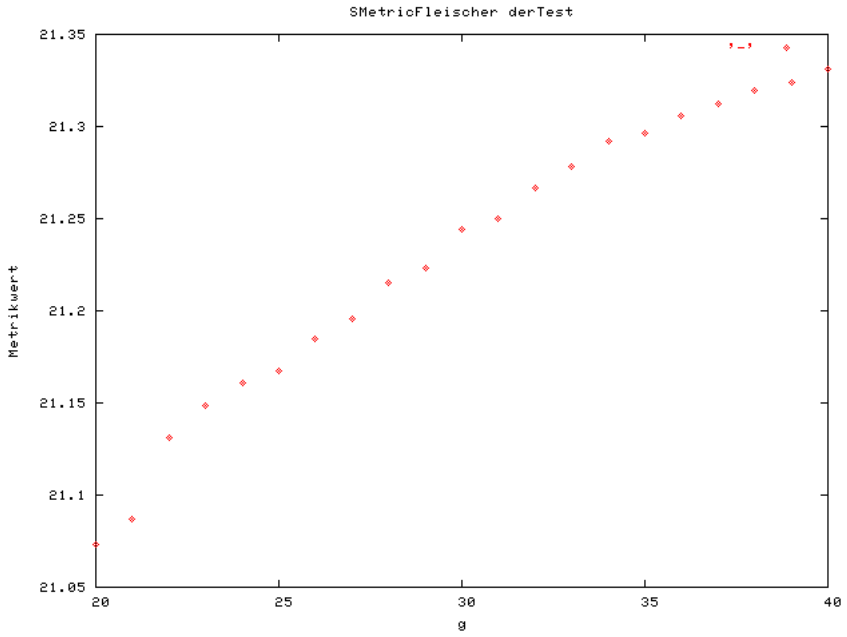

Abbildung 10: Beispiel Diagramm für den `MeanMetricPlotter`

5.3.5 SurfacePlotter

5.3.5.1 Beschreibung

Der `SurfacePlotter` basiert im Großen und Ganzen auf dem `MetricsPlotter` (5.3.3), stellt aber nicht nur ein Balkendiagramm, sondern eine Fläche (Surface) im dreidimensionalen Raum dar.
Diese Fläche stellt den Metrikenwert dar (bezüglich der Z-Koordinatenachse), und zwar in Abhängigkeit zweier Parameter, die bei den darzustellenden Algorithmenläufen variiert wurden. Diese Parameter werden durch die x- und die y-Achse des Plots dargestellt.

Es wird also beim Start des `SurfacePlotters` ein Verzeichnis angegeben, in dem sich die Ausgabedateien mehrerer Algorithmenläufe befinden. Zusätzlich werden die Namen der beiden Parameter angegeben, deren Variation die Grundlage für die Surface-Visualisierung bilden sollen. Außerdem muss eine Metrik angegeben werden, anhand deren Wert die Höhe der zu plottenden Oberfläche berechnet werden soll. Ein Beispiel-Bild ist in Abbildung 11 zu sehen. Hier wurde ein Surface-Plot von Algorithmenläufen bezüglich der `NewSMetric` ausgegeben. In den Algorithmenläufen wurden die Parameter c1 und c2 variiert; und zwar in den Grenzen $0 \leq c1 \leq 1,2$ und $0 \leq c2 \leq 1,6$.

Wurden für die Erzeugung der Ausgabedateien mehrere Algorithmenläufe durchgeführt, bei denen die Parameter zwar jeweils gleich waren, aber verschiedene Random-Seeds benutzt wurden, so wird vom SurfacePlotter zwischen diesen Läufen mit gleichen Parameter-Einstellungen arithmetisch gemittelt. So können Unterschiede in den Algorithmenläufen, die durch verschiedene interne Zufallswerte bedingt sind, heraus-

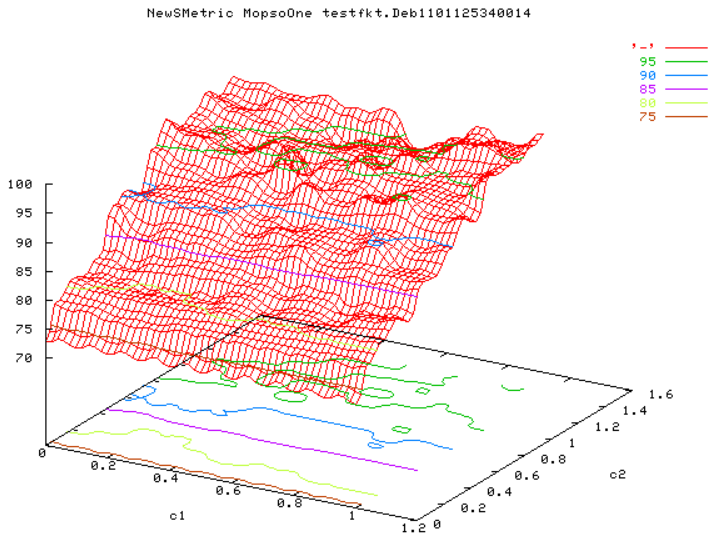

Abbildung 11: Beispiel-Bild eines Surface-Plots mit dem `SurfacePlotter` bezüglich der Parameter c1 und c2

gefiltert werden.

5.3.5.2 Einstellmöglichkeiten

Der `SurfacePlotter` akzeptiert außer den allgemeinen Parametern folgende Kommandozeilen-Parameter:

- `-p1`: der erste der beiden in den Algorithmenläufen variierten Parameter, nach denen der Surface-Plot erstellt werden soll (z. B. `-p1 generations`).
- `-p2`: der zweite der beiden in den Algorithmenläufen variierten Parameter, nach denen der Surface-Plot erstellt werden soll (z. B. `-p2 inertia`).

5.3.6 PercentageMetricPlotter

5.3.6.1 Beschreibung

Der `PercentageMetricPlotter` bereitet die Ergebnisse von Algorithmen auf etwas andere Weise als die restlichen von der PG entwickelten Plotter auf. Mittels Metrikwerten wird analysiert, wie oft Algorithmen eine bestimmte „Güte“ erreichen. Da die verwendeten Heuristiken in Abhängigkeit der verwendeten Zufallszahlen laufen, ist diese Auswertung interessant, um zu sehen, ob ein Algorithmus bei bestimmten Einstellungen tatsächlich durchgängig gute Ergebnisse liefert.

Dazu ist es notwendig, dass Algorithmen-Läufe mit konstanten Einstellungen mehrfach mit unterschiedlichen Random-Seeds durchgeführt werden.

Durch unterschiedliche Möglichkeiten, auf die im Folgenden eingegangen wird, wird eine Gütegrenze bestimmt, ein Metrikwert, den die Ergebnisse der Algorithmen mindestens erreichen müssen, um als „gut“ eingestuft zu werden. Berechnet wird nun, wie viele der Algorithmen-Läufe mit gleichen Einstellungen (abgesehen vom variierten Random-Seed) diese Gütegrenze erreicht haben. Dies erfolgt relativ zur Anzahl der jeweiligen Algorithmen-Wiederholungen in Prozent.

Setzen wir beispielsweise die Gütegrenze auf sieben fest. Werden dann zehn Wiederholungen eines Algorithmus mit bestimmen Einstellungen durchgeführt, von denen vier den Metrikwert sechs und sechs Läufe mindestens den Metrikwert 7 erreicht haben, erfolgt die Ausgabe von „60%“.

Abbildung 12 zeigt eine typische Ausgabe des `PercentageMetricPlotters`, bei dem der Parameter `evaluations` (der evaluationsbasierte LiveOutput eines Algorithmus) auf der X-Achse aufgetragen ist und 90% des besten Metrikwerts als Gütegrenze gewählt wurde. Es ist zu erkennen, dass bereits bei wenigen Evaluationen die geforderte Güte von einigen wenigen Läufen erreicht wird und die Anzahl dieser Läufe sehr stark ansteigt. Ab ungefähr 12000 Evaluationen sinkt die Steigung der Kurve enorm. Ab dort erreichen aber bereits ca. 80% der Läufe die geforderte Güte. Das leichte Ansteigen der Kurve bis 50000 Evaluationen zeigt, dass die restlichen Läufe des Algorithmus zum Erreichen der Gütegrenze mehr Evaluationen benötigen. Ob alle Läufe die Gütegrenze erreichen, ist dem Diagramm nicht zu entnehmen, weil die Experimente nur bis 50000 Evaluationen durchgeführt wurden.

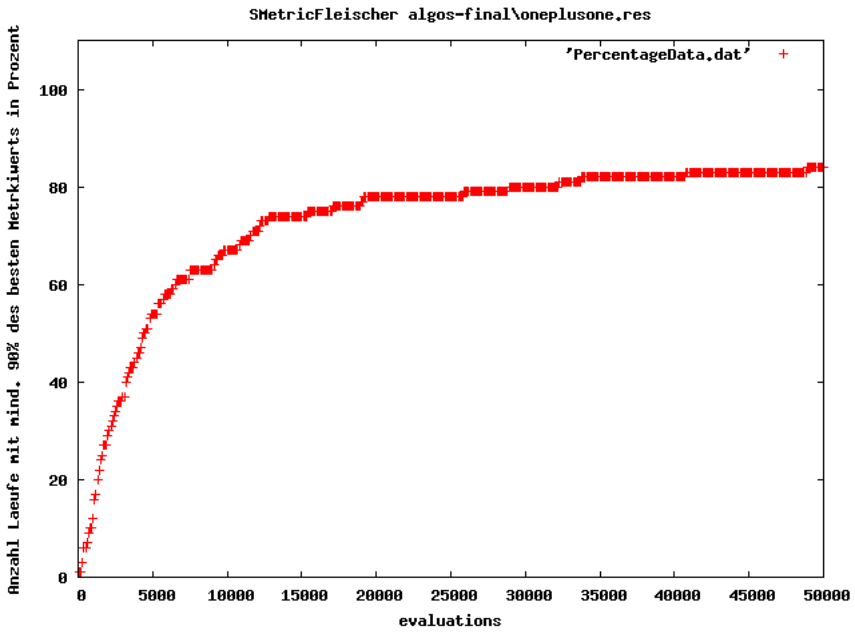

Abbildung 12: Beispiel-Ausgabe des `PercentageMetricPlotter` mit Punkten

5.3.6.2 Einstellmöglichkeiten

Die Gütegrenze kann mit den folgenden Kommandozeilenparametern entweder manuell gesetzt oder aufgrund einer Angabe in Prozent vom größten auftretenden Metrikwert automatisch bestimmt werden. Der Plotter verarbeitet außer den allgemeinen Parametern folgende Kommandozeilenparameter:

- `-p1` <`Wert`>: Der Plotter fasst die Ergebnisse aller Algorithmen-Läufe zusammen, die den gleichen Wert bei dem hier angegebenen Parameter haben. Dateien, die diesen Parameter nicht im Namen tragen, werden ignoriert. Die Ergebnisse werden nach diesem Parameter entlang der X-Achse aufgetragen (z. B.: `-p1 20`).

- `-boundary` <`Wert`>: Die Gütegrenze kann mit diesem Parameter auf einen festen Wert gesetzt werden (z. B.: `-boundary 18.8`).

- `-percentage` <`Wert`>: Mit diesem Parameter kann die Gütegrenze automatisch auf einen bestimmten Prozentanteil des besten Metrikwerts gesetzt werden.
  Diese Funktion steht jedoch nur zur Verfügung, wenn der Parameter `-boundary` nicht verwendet wird. Er ist mit dem Parameter `-bestMetricValue` kombinierbar. Wird weder der `-percentage`, noch der `-boundary`-Parameter angegeben, wird automatisch 90% des besten Metrikwerts als Gütegrenze verwendet (z. B.: `-percentage 90`).

- `-bestMetricValue` <`Wert`>: Der Plotter berechnet den besten Metrikwert, auf den sich der Parameter `-percentage` bezieht, standardmäßig automatisch, indem der höchste Metrikwert aller zu verarbeitenden Dateien verwendet wird. Es ist jedoch mit diesem Parameter möglich, einen besten Metrikwert manuell vorzugeben. Dies ist insbesondere bei Darstellung mehrerer Plots in einem Diagramm notwendig, damit die Gütegrenze bei allen Plots übereinstimmt (z. B.: `-bestMetricValue 20`).

5.3.6.3 Variationsmöglichkeiten

Um Abbildungen mehrerer Läufe in einem Diagramm darstellen zu können, muss der Plotter für jeden Algorithmen-Lauf einzeln aufgerufen und die Ergebnisse anschließend zusammen visualisiert werden.
Die Daten müssen mit gleichen Einstellungen generiert werden, weil die Darstellung sonst verfälscht würde. Relevant sind dazu der Referenzpunkt, der manuell gewählt werden muss und die Gütegrenze, die entweder mit `-boundary` oder `-bestMetricValue` nach Betrachtung aller Plotter-Aufrufe festgesetzt werden muss. Dazu sollte der Plotter zuerst ohne diese Einstellungen für jeden Algorithmen-Lauf gestartet werden, wodurch u. a. der gewählte Referenzpunkt und der größte Metrikwert berechnet und ausgegeben werden. Hierzu erstellt der Plotter auch eine Datei mit der Endung `.dump`, in der zusätzlich alle Konsolen-Ausgaben stehen, damit diese nicht abgeschrieben werden müssen.

Da die Ergebnisse in eine eigene Datei mit der Endung `.dat` geschrieben werden, können schließlich die Datendateien aller darzustellenden Algorithmen-Läufe zusammen an Gnuplot zur Visualisierung gegeben werden.

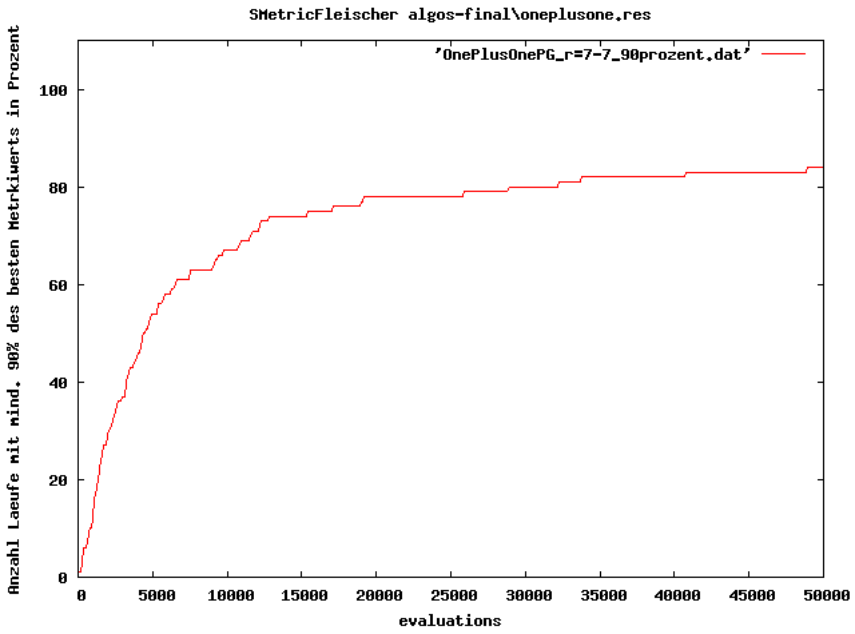

Abbildung 13: Beispiel-Ausgabe des `PercentageMetricPlotters` mit Linien

Um das Diagramm wie in Abbildung 13 mit Linien statt Punkten darzustellen ist es neben der Anpassung des *Gnuplot*-Befehls notwendig, die Zeilen in der Datei mit der Endung `.dat` nach der ersten Spalte zu sortieren.

5.3.7 BohrVisualizer

5.3.7.1 Beschreibung

Der `BohrVisualizer` kann die Individuen des Temperierbohrungsproblems (siehe 5.4.3) grafisch darstellen. Es handelt sich dabei um die Eckpunkte eines Polygonzuges, der sich idealerweise in einem vorgegebenen Quader befindet und eine darin enthaltene Sphäre nicht schneidet. Da die schriftliche Darbringung solcher Polygonzüge nicht sehr intuitiv ist, wurde dieses Tool entwickelt.

Die Visualisierung geschieht mit *Gnuplot*. Die Ausgabe erfolgt entweder interaktiv in *Gnuplot* oder über Grafikdateien im PNG-Format (siehe Abbildung 14). Diese Grafikdateien können dann in einer animierten GIF-Datei zusammengefasst werden. Die Darstellung erfolgt wahlweise mit oder ohne den Quader, der die äußeren Dimensionen des Werkzeuges markiert.

5.3.7.2 Einstellmöglichkeiten

Der `BohrVisualizer` verarbeitet folgende Kommandozeilenparameter:

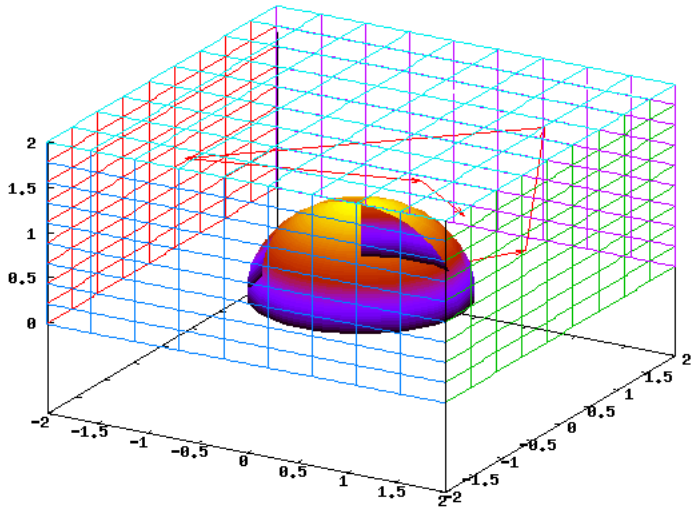

Abbildung 14: Ein gültiger Polygonzug, der im Modus `gnuplotboxedpng` visualisiert wurde

- `-d` <`Eingabedatei`>: Der BohrVisualizer benötigt eine Datei, in der die Polygonzüge gespeichert sind. Meist sind dies Dateien mit der Endung „`.in`“.
- `-o` <`Präfix`>: Um die Ausgaben mehrerer Aufrufe des `BohrVisualizers` unterscheiden zu können, kann mit diesem Parameter das Ausgabeverzeichnis und ein Präfix für die Namen der Ausgabedateien gewählt werden. Standard ist hier „bohrviz“.
- `-mode` <`gnuplot(boxed)(png/anigif)`>: Hier kann die Ausgabe bestimmt werden. Sie ist derzeit nur mit `gnuplot` möglich. Der Zusatz `boxed` aktiviert die Darstellung des Quaders. `png` steht für die Ausgabe in PNG-Dateien, `anigif` erzeugt PNG-Dateien und gibt die Parameter aus, die für den Aufruf des `AnimatedGifEncoder` notwendig sind. Ohne `anigif` oder `png` wird der interaktive Modus gestartet. Standard ist hier „gnuplot“ (z. B.: `-mode gnuplotboxedanigif`).

5.3.8 Tool zur Erstellung animierter GIF-Bilder

5.3.8.1 Beschreibung

Durch die Klasse `AnimatedGifEncoder` ist es möglich, kleinere Animationen zu erstellen und so beispielsweise darzustellen, wie sich Individuen in einem Algorithmen-Lauf über die Zeit hin der Pareto-Front annähern. Die Klasse benutzt die beiden Klassen `LZWEncoder` von Jef Poskanzer und J. M. G. Elliott sowie `NeuQuant` von Anthony Dekker.

Die Handhabung der Klasse ist recht simpel. Man gibt lediglich einen Verzeichnisnamen an, in dem die Grafikdateien für die Animation liegen. Erlaubte Formate für die Grafiken sind `*.gif`, `*.jpg` und `*.png`. Aus den gesamten Daten im festgelegten Verzeichnis wird dann eine animierte GIF-Datei erstellt.

5.3.8.2 Einstellmöglichkeiten

Der Encoder akzeptiert folgende Kommandozeilen-Parameter:

- `-d`<`Verzeichnis`>: Legt den Namen des Quellverzeichnisses fest, in dem sich die Grafiken befinden.
- `-o`<`Ausgabe`>: Mit diesem optionalen Parameter kann man einen Namen für die Ausgabedatei angeben. Der Defaultname ist `AnimatedGIF.gif`
- `-delay`<`Verzögerung`>: Hier läßt sich einstellen, wie viele Millisekunden die Verzögerung zwischen den einzelnen Frames betragen soll. Gibt man diesen Parameter nicht an, werden 500 Millisekunden verwendet.
- `-repeat`<`Wiederholungen`>: Durch diesen Parameter läßt sich festlegen, wie oft die Animation wiederholt werden soll. Lässt man den Parameter wegfallen, wird ein animiertes GIF mit Endlosschleife generiert.
- `-sort`<`Sortierung`>: Zusätzlich kann spezifiziert werden, nach welchem Kriterium im Namen die Grafikdateien sortiert werden. Auf diese Weise wird die Reihenfolge der einzelnen Frames in der Animation bestimmt.

5.3.9 R-Klassen

5.3.9.1 Beschreibung

Die R-Klassen `RScatterplot, RHistogram` und `RBoxplot` sollen die Anwendung von `R` erleichtern. Es wird jeweils ein R-Script erzeugt und die Eingabe ausgegeben, die für einen Aufruf des Scripts mit `R` benötigt wird.

Das Script von `RScatterplot` gibt dann eine Grafikdatei mit dem Scatterplot und ein Textfile mit den Korrelationen zwischen den Variablen aus. `RBoxplot` gibt eine entsprechende Boxplotgrafik aus und `RHistogram` eine Grafik eines Histogramms. Wahlweise in einem Balkendiagramm, Graphen oder beiden Darstellungsformen.

5.3.9.2 Einstellmöglichkeiten

Die R-Klassen akzeptieren folgende Kommandozeilen-Parameter:

- `-d`<`Datei`>: Legt den Namen der Quelldatei fest, in dem sich die Daten befinden.
- `-o`<`Ausgabe`>: Mit diesem Parameter legt man einen Namen für die Ausgabedatei fest.

- `-col<Spalte>`(Nur RHistogram): Hier muss eingestellt werden, welche Spalte ausgewertet werden soll.
- `-mode<0=Balkendiagram, 1=Kontinuierlicher Graph, 2=Beides>`(Nur RHistogram): Hier kann man einstellen, welcher Darstellungsmodus verwendet werden soll.

5.4 Testfunktionen und Praxis-Anwendungen

Im folgenden werden die Optimierungsprobleme vorgestellt, mit denen die Projektgruppe gearbeitet hat. Dabei handelt es sich zum einen um Testfunktionen (5.4.1), zum anderen um Problemstellungen aus der Praxis (5.4.1.10 und 5.4.3). Die Testfunktionen sind künstlich konstruierte, mathematische Probleme, die allein dazu dienen, das Verhalten von Algorithmen zu untersuchen. Dagegen stammen die beiden vorgestellten Praxis-Problemstellungen aus der Industrie; hier gilt es, „echte“ Lösungen zu finden, die dann auch tatsächlich in wirtschaftlich bedeutsamen Produkten umgesetzt werden.

5.4.1 Testfunktionen

Die Aufgabe der PG bestand darin, Heuristiken zur Lösung von Problemstellungen zu implementieren, zu optimieren und zu verbessern, sowie ggf. neue zu entwerfen. Doch um Probleme lösen zu können, benötigt man erst einmal welche. Hier kommen die Testfunktionen ins Spiel.

Testfunktionen sind standardisierte Probleme, deren Lösungen bekannt sind. Dies gilt insbesondere für die Optima. Sie ermöglichen es uns, unsere Algorithmen zu testen, ihr Verhalten zu beobachten und die Ergebnisse mit den bekannten Optima zu vergleichen. Um dies zu erleichtern, handelt es sich hier meist um relativ einfache Funktionen, die leicht zu verstehen, zu implementieren und zu berechnen sind.

Im ersten Semester haben wir drei Testfunktionen implementiert; die `DoubleParabola` (5.4.1.1), die `Fonseca`-Testfunktion (5.4.1.2) und das `Knapsack`-Problem (5.4.1.3). Diese sind noch recht einfach und konnten sehr gut von unseren Algorithmen gelöst werden. Um die Implementierung dieser und weiterer Testfunktionen zu vereinfachen und zu standardisieren, hatten wir ein Interface geschrieben, welches von allen Testfunktionen implementiert werden musste.

Im zweiten Semester der Projektgruppe wollten wir die Algorithmen nun auf schwierigere Probleme anwenden, daher haben wir weitere, anspruchsvollere Testfunktionen implementiert. So entstanden die `Deb`-Testfunktion (5.4.1.4), die `Schaffer`-Funktion (5.4.1.5), sowie die Funktionen `ZDT1` (5.4.1.6) bis `ZDT4` (5.4.1.9) und `ZDT6` (5.4.1.10).

5.4.1.1 DoubleParabola

Die Doppel-Parabel ist eine zweikriterielle Testfunktion mit einer Variable. Sie ist auf den reellen Zahlen definiert und wird in Java auf den Datentyp `double` abgebildet.

Beide Kriterien sind zu minimieren.

Die Doppel-Parabel ist die eindimensionale und zweikriterielle Version der bekannten *Sphere* -Testfunktion.
Die Funktion ist, wie der Name schon sagt, eine doppelte Parabel, wobei die Funktionen f_1 und f_2 jeweils eine Normalparabel sind. Die Funktion f_1 hat ihr Minimum in $x = 0$, die Funktion f_2 hat ihr Minimum in $x = 2$.

Somit sieht das Testproblem `DoubleParabola` wie folgt aus:

$$\begin{aligned} Minimiere: f_1 &= x^2 \\ Minimiere: f_2 &= (x-2)^2 \end{aligned}$$

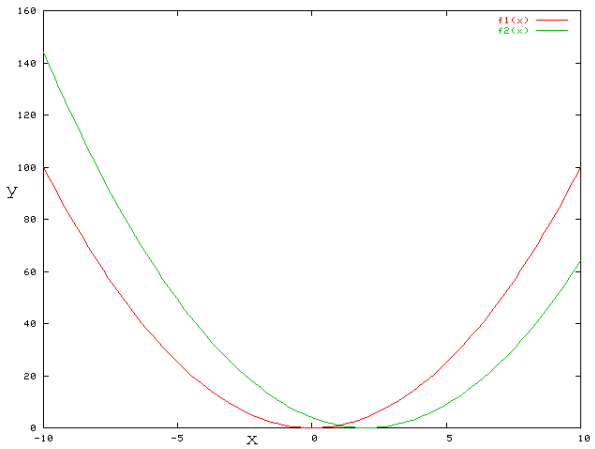

Abbildung 15: Die DoubleParabola im Objektvariablen-Raum

Die Pareto-Front der Doppel-Parabel ist die eindimensionale Verbindungsstrecke zwischen den beiden Einzel-Minima der Funktionen f_1 und f_2. Somit liegen die Pareto-optimalen Punkte bei $0 \leq x \leq 2$.

5.4.1.2 FonsecaF1

Die `FonsecaF1` ist eine zweikriterielle Testfunktion auf zwei Variablen. Sie ist auf den reellen Zahlen definiert und wird in Java auf den Datentyp `double` abgebildet. Beide Kriterien sind zu minimieren.
Die Funktion ist eine Kombination von zwei einkriteriellen `Fonseca`-Funktionen, die grob einer *Sphären*-Funktion ähneln. Die Funktion f_1 hat ihr Minimum in $x = (1, -1)$ mit dem Wert 0. Je weiter man sich von x entfernt, desto größer werden die Funktionswerte, so dass bei einem Abstand von 5,5 vom Minimum wegen der Rundungsfehler das eigentlich nicht vorhandene Maximum mit Wert 1 erreicht wird. Die Funktion f_2 ist eine Verschiebung von f_1 und hat das Minimum an der Stelle $(-1, 1)$.

Formal sieht das Optimierungsproblem so aus:

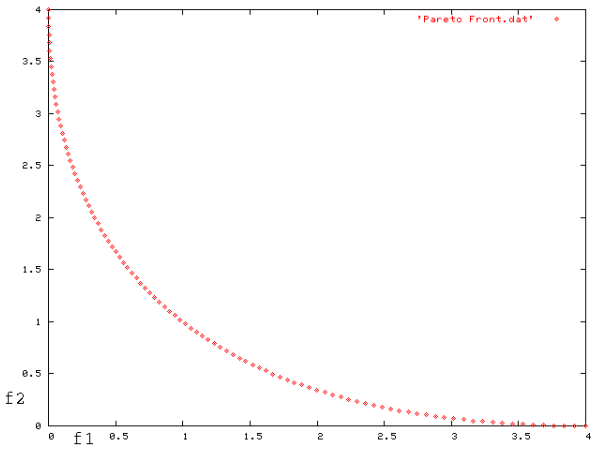

Abbildung 16: Die Pareto-Front der DoubleParabola

$$Minimiere : f_1(x_1, x_2) = 1 - \frac{1}{\exp((x_1 - 1)^2 + (x_2 + 1)^2)}$$
$$Minimiere : f_2(x_1, x_2) = 1 - \frac{1}{\exp((x_1 + 1)^2 + (x_2 - 1)^2)}$$

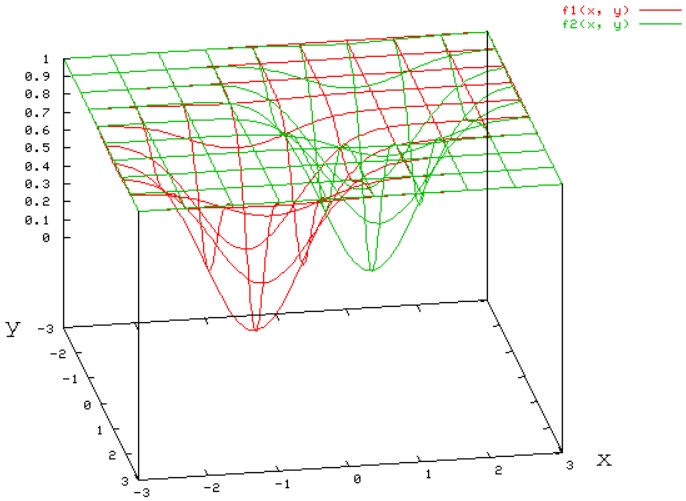

Abbildung 17: FonsecaF1

Die pareto-optimalen Punkte im Definitionsbereich liegen auf der Geraden zwischen den Minima (1,-1) und (-1,1). Die Pareto-Front der `FonsecaF1` ist im Zielbereich eine bogenförmige Verbindungsstrecke zwischen den Punkten $(0, \frac{1}{\exp(8)})$ und $(0, \frac{\exp(8)}{1})$.

5.4.1.3 Knapsack Problem

Die Testfunktion `KnapsackProblem` ist eine Implementierung des mehrkriteriellen 0/1 Rucksackproblems. Es handelt sich um ein kombinatorisches Problem, wodurch sich diese Testfunktion von den anderen absetzt. Sowohl die Dimension, als auch die Anzahl der Kriterien ist variabel.
Beim klassischen Rucksackproblem geht es darum, einen Rucksack möglichst gut zu bepacken. Es gibt einen Rucksack mit einem bestimmten Fassungsvermögen (einkriteriell) und einer bestimmten Anzahl von Gegenständen. Diese haben jeweils eine Größe und einen Nutzwert. Jeder Gegenstand kann nun in den Rucksack gepackt werden oder nicht. Die Summe der Größen aller im Rucksack befindlichen Gegenstände darf das Fassungsvermögen des Rucksacks nicht überschreiten. Die Güte einer Bepackung ist die Summe der Nutzwerte aller Gegenstände im Rucksack.
Dieses einkriterielle Problem kann sehr einfach zu einem mehrkriteriellen Problem erweitert werden, indem man jedem Gegenstand mehrere Nutzwerte zuweist. Die Kriterien sind dann die Summen der jeweiligen Nutzwerte aller im Sack befindlichen Gegenstände. Die Nutzwerte in unserer Implementierung sind negativ, damit wir ein Minimierungsproblem haben, welches ohne Probleme mit unseren Tools funktioniert.
Unsere Implementierung ist sehr variabel gehalten. Es ist möglich, alle Eigenschaften zu ändern. Konkret sind dies:

- Größe des Rucksacks
- Anzahl der Gegenstände
- Anzahl der Nutzwerte pro Gegenstand (Anzahl der Kriterien)
- maximaler Wert eines einzelnen Nutzwertes
- maximales Gewicht eines Rucksackgegenstandes
- Random Seed zur Erzeugung der Gewichte und Nutzwerte

Da es extrem viele Kombinationsmöglichkeiten dieser Eigenschaften gibt, werden die Nutzwerte und das Gewicht der Gegenstände zufällig erzeugt. Dazu wird ein Random Seed verwendet. Dadurch können immer gleiche Rucksackprobleminstanzen erzeugt werden oder für gleiche Rahmenbedingungen (Größe, Anzahl Gegenstände/Nutzwerte etc.) andere Gegenstände verwendet werden.
Das Problem hierbei war, dass noch keine Parameterübergabe an die Testfunktionen spezifiziert wurde. Die Parameter müssen deshalb direkt im Quellcode über den Konstruktor der Testfunktion gesetzt werden. Da im Allgemeinen nur der leere Konstruktor verwendet wird, setzt dieser Standardwerte für den variablen Konstruktor.
Die variable Erzeugung eines Rucksackproblems bringt ein Problem mit sich. Wir kennen die Pareto-optimalen Werte nicht. Diese müssen, da es sich um ein NP-vollständiges Problem handelt, durch „ausprobieren“ aller Bepackungen ermittelt werden. Bei den bisher betrachteten Probleminstanzen gab es zu wenige Pareto-optimale Werte, um eine durchgängige Front zu erzeugen.

5.4.1.4 Deb-Testfunktion

Die `Deb`-Funktion ist eine zweikriterielle Testfunktion mit zwei Variablen. Sie ist auf den reellen Zahlen definiert und wird in Java auf den Datentyp `double` abgebildet. Beide Kriterien sind zu minimieren.

Diese Test-Funktion geht auf Kalyanmoy Deb ([25]) zurück.
Die Paretofront der Testfunktion besteht aus 4 einzelnen, nicht miteinander verbundenen Stücken.

Formell wird die Funktion beschrieben durch:

$$\begin{aligned} Minimiere: f_1(x_1, x_2) &= x_1 \\ Minimiere: f_2(x_1, x_2) &= (1 + 10x_2) \times (1 - (\frac{x_1}{1 + 10x_2})^{\alpha} \\ &\quad - \frac{x_1}{1 + 10x_2} sin(2\pi q x)) \end{aligned}$$

Die Pareto-Front der Deb-Funktion besteht aus 4 einzelnen Teil-Fronten, die untereinander nicht verbunden sind.

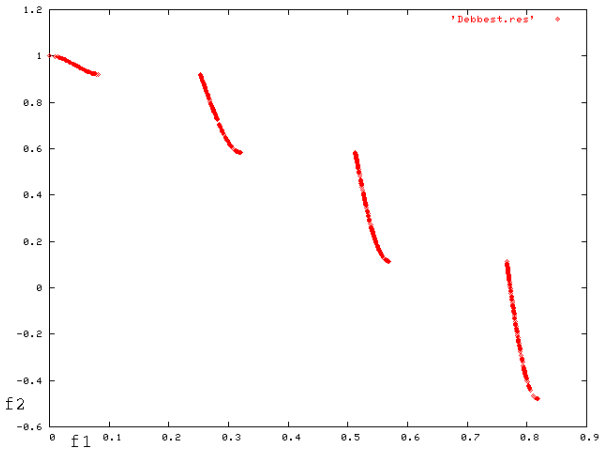

Abbildung 18: Die Deb-Testfunktion im Funktionen-Raum

5.4.1.5 Schaffer-Testfunktion

Die `Schaffer`-Funktion ist eine zweikriterielle Testfunktion mit einer Variablen. Sie ist auf den reellen Zahlen definiert und wird in Java auf den Datentyp `double` abgebildet. Beide Kriterien sind zu minimieren.

Diese Testfunktion geht auf J. David Schaffer zurück.
Die Paretofront der Testfunktion besteht aus 2 einzelnen, nicht miteinander verbundenen Stücken.

Formell wird die Funktion beschrieben durch:

$$Minimiere : f_1(x) = \begin{cases} -x & : \quad x \leq 1 \\ -2+x & : \quad 1 < x \leq 3 \\ 4-x & : \quad 3 < x \leq 4 \\ -4+x & : \quad x > 4 \end{cases}$$

$$Minimiere : f_2(x) = (x-5)^2$$

Die Pareto-Front der Schaffer-Funktion ist eine gebogene Kurve aus 2 einzelnen Teilen, die untereinander nicht verbunden sind.

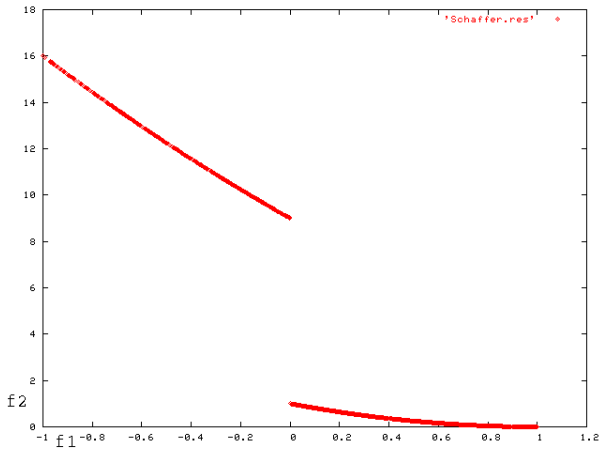

Abbildung 19: Die Schaffer-Testfunktion im Funktionen-Raum

5.4.1.6 ZDT1

Die ZDT-Testfunktionen wurden 2000 von Zitzler, Deb und Thiele entwickelt, für weitere Details siehe [18].
Die ZDT1-Funktion hat 30 Variablen und eine konvexe pareto-optimale Menge.
Alle Variablen x_i liegen im Bereich [0,1]. Somit sieht das Testproblem ZDT1 wie folgt aus:

$$ZDT1 = \begin{cases} f_1(x) = x_1 \\ g(x) = 1 + \frac{9}{n-1} \sum_{i=2}^{n} x_i \\ h(f_1, g) = 1 - \sqrt{\frac{f_1}{g}} \end{cases}$$

Die pareto-optimale Region entspricht $0 \leq x_1 \leq 1$ und $x_i = 0$, für $i = 2, 3, ..., 30$.

5.4.1.7 ZDT2

Die ZDT2-Funktion hat 30 Variablen und eine konkave pareto-optimale Menge.

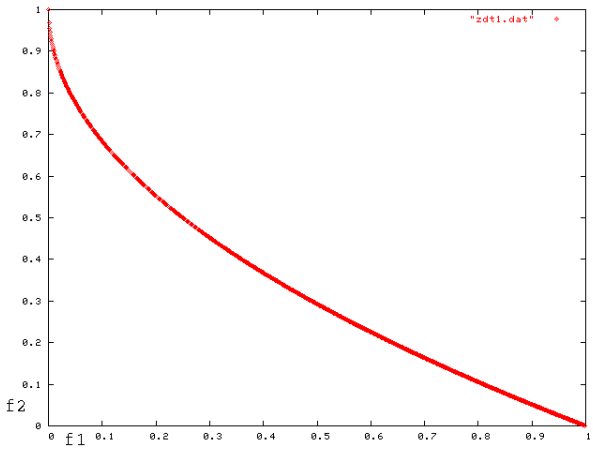

Abbildung 20: Die pareto-optimalen Werte der ZDT1

Alle Variablen liegen im Bereich [0,1]. Somit sieht das Testproblem ZDT2 wie folgt aus:

$$ZDT2 = \begin{cases} f_1(x) = x_1 \\ g(x) = 1 + \frac{9}{n-1} \sum_{i=2}^{n} x_i \\ h(f_1, g) = 1 - (\frac{f_1}{g})^2 \end{cases}$$

Die pareto-optimale Region entspricht $0 \leq x_1 \leq 1$ und $x_i = 0$, für $i = 2, 3, ..., 30$.

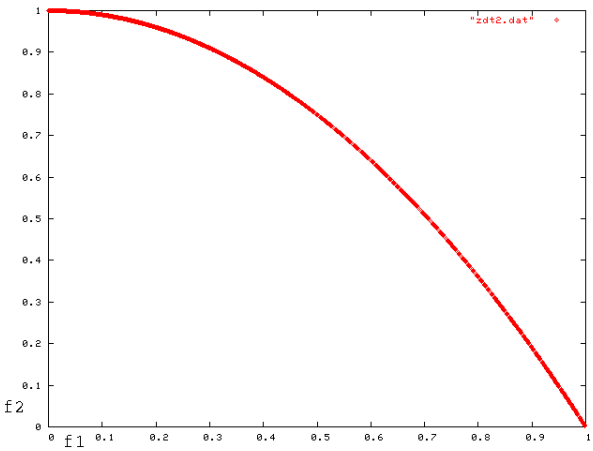

Abbildung 21: Die pareto-optimalen Werte der ZDT2

5.4.1.8 ZDT3

Die ZDT3-Funktion hat 30 Variablen und fünf getrennte konvexe Fronten.
Alle Variablen x_i liegen im Bereich [0,1]. Somit sieht das Testproblem ZDT3 wie folgt aus:

$$ZDT3 = \begin{cases} f_1(x) = x_1 \\ g(x) = 1 + \frac{9}{n-1} \sum_{i=2}^{n} x_i \\ h(f_1, g) = 1 - \sqrt{\frac{f_1}{g}} - (\frac{f_1}{g}) \sin(10\pi f_1) \end{cases}$$

Die pareto-optimale Region entspricht $x_i = 0$, für $i = 2, 3, ..., 30$ und daher liegen nicht alle Punkte, die die Bedingung $0 \leq x_1 \leq 1$ erfüllen auf der pareto-optimalen Front.

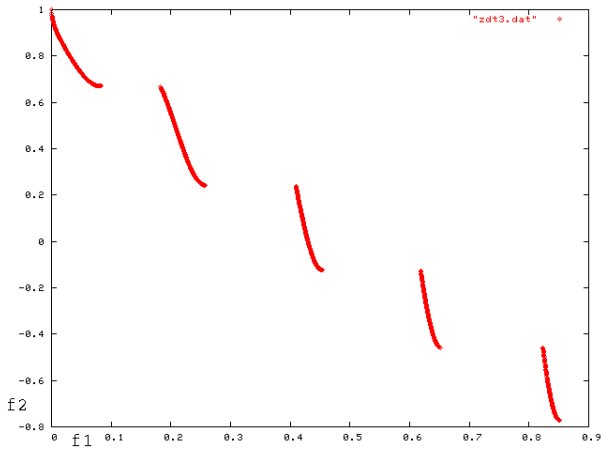

Abbildung 22: Die pareto-optimalen Werte der ZDT3

5.4.1.9 ZDT4

Die ZDT4-Funktion hat 10 Variablen und eine konvexe pareto-optimale Menge.
Die Variable x_1 liegt im Bereich $[0, 1]$ und alle anderen im Bereich $[-5, 5]$. Somit sieht das Testproblem ZDT4 wie folgt aus:

$$ZDT4 = \begin{cases} f_1(x) = x_1 \\ g(x) = 1 + 10(n-1) + \sum_{i=2}^{n} (x_i^2 - 10\cos(4\pi x_i)) \\ h(f_1, g) = 1 - \sqrt{\frac{f_1}{g}} \end{cases}$$

Die globale pareto-optimale Front entspricht $0 \leq x_1 \leq 1$ und $x_i = 0$, für $i = 2, 3, ..., 10$, es existieren jedoch eine Vielzahl an lokalen pareto-optimalen Fronten. Die globale pareto-optimale Front entspricht der von der ZDT1.

5.4.1.10 ZDT6

Die ZDT6-Funktion hat 10 Variablen und eine konkave pareto-optimale Menge.
Alle Variablen x_i liegen im Bereich $[0, 1]$. Somit sieht das Testproblem ZDT6 wie folgt aus:

$$ZDT6 = \begin{cases} f_1(x) = 1 - \exp(-4x_1) \sin^6(6\pi x_1) \\ g(x) = 1 + 9[\sum_{i=2}^{10} x_i/9]^{0.25} \\ h(f_1, g) = 1 - (\frac{f_1}{g})^2 \end{cases}$$

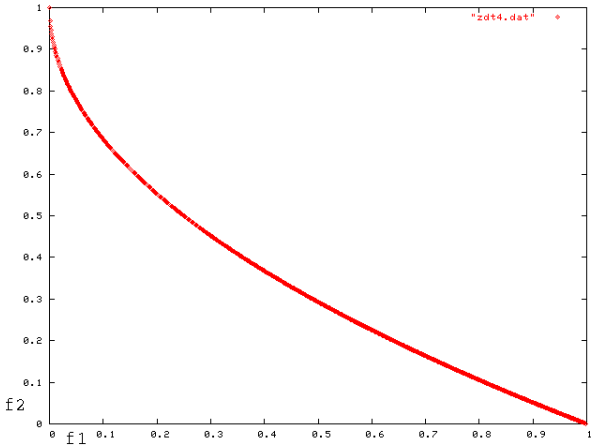

Abbildung 23: Die pareto-optimalen Werte der ZDT4

Die globale pareto-optimale Front entspricht $0 \leq x_1 \leq 1$ und $x_i = 0$, für $i = 2, 3, ..., 10$. Die Besonderheit der ZDT6-Funktion besteht darin, dass die Punkte-Dichte auf der pareto-optimalen Front nicht gleichmäßig ist.

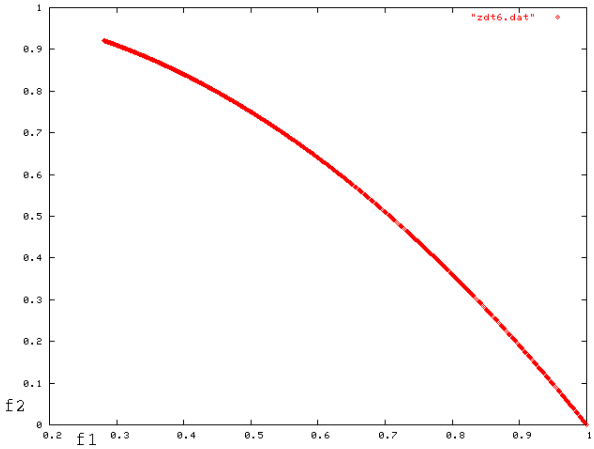

Abbildung 24: Die pareto-optimalen Werte der ZDT6

5.4.2 Fahrstuhlsimulationen und das S-Ring-Modell

5.4.2.1 Motivation

Das eigentliche Vorhaben dieser Arbeitsgruppe war die Anbindung eines in der Industrie eingesetzten Fahrstuhlsimulators. Anhand dessen sollte die Relation der Laufzeit zur Qualität der Ergebnisse der im Projekt implementierten Algorithmen untersucht werden, um Erkenntnisse zur Realzeit-Berechnung in Fahrstuhlsystemen zu gewinnen.
Um erste Erfahrungen zu sammeln und schnell Experimente mit vorhandenen Algorithmen und Metriken durchführen zu können, wurde zunächst ein einfacher an das S-Ring-Modell angelehnter Simulator eingebunden.
Im Hinblick auf das noch verbleibende PG-Semester und die langen Laufzeiten des komplexen Simulators erschien es wenig praktikabel, diesen noch einzubinden und anschließend ausführliche Designs anzuwenden und auszuwerten. Daher konzentrierten sich die Bemühungen dann weiterhin auf das S-Ring-Modell.

5.4.2.2 Die Fahrstuhlproblematik - eine Übersicht

Bei der Konzeption von Fahrstuhlsystemen müssen die verschiedensten (auch gegenläufigen) Faktoren berücksichtigt werden. So soll das Fahrstuhlverhalten zum Beispiel die verschiedenen Tagesereignisse meistern können: Im Großen und Ganzen unterscheidet man drei Szenarien, nämlich den Uppeak-Traffic (morgens, wenn alle Personen vom Erdgeschoss in ihre Büroetagen fahren müssen), den Downpeak-Traffic (am Feierabend, wenn die Beschäftigten von ihren Büros ins Erdgeschoss möchten) sowie den Interfloor-Traffic (verursacht durch Mitarbeiter, die während der Arbeitszeit zum Beispiel in Büros oder Besprechunsräume auf anderen Etagen wechseln). Andere Muster (zum Beispiel Lunchtime-Traffic zur Mittagszeit) resultieren aus verschiedenen Anteilen der drei vorgestellten Tagesereignisse [26].
Unter Berücksichtigung dieser (und anderer) Faktoren gilt es nun, das System hinsichtlich der Fahrgastwartezeit (Waiting-Time), der Fahrzeit (Riding-Time) oder anderer Ziele zu optimieren [27].
Um eine Lösung auch in der Praxis einsetzen zu können, müssen zudem die (häufig irrationalen) menschlichen Interaktionen (zum Beispiel Aufhalten einer Tür, falsche oder überflüssige Stockwerkwahl) oder die unterschiedliche Beschaffenheit von Gebäuden (Höhe, Art und Lage von Sozial-, Geschäfts- oder Wohnbereichen etc.) wie auch unterschiedliche Anforderungen (zum Beispiel im Sicherheits- oder Pflegebereich) bedacht werden [28].
Daraus ergibt sich schnell die Erkenntnis, dass die Konzeption eines solchen Systems individuell für jede Fahrstuhlinstallation neu durchgeführt werden muss und nur schlecht übertragbar ist, was wiederum zu hohen Kosten auf der Seite der Hersteller führt.
Das Ziel der Fahrstuhlindustrie ist es also, möglichst effiziente Lösungen für die Programmierung von Fahrstuhlsystemen zu entwickeln bzw. Fahrstuhlsysteme zu schaffen, die sich selbst an die Umgebung und die aktuellen und voraussichtlichen Anfragen von Fahrgästen anpassen.
Zu diesem Zweck werden umfangreiche Simulatoren entwickelt, mit welchen in Verbindung mit verschiedenen Verfahren (zum Beispiel neurale Netze, evolutionäre Algo-

rithmen) Lösungen entwickelt bzw. die Entwicklung solcher Lösungen erforscht werden können.

5.4.2.3 Der verwendete Simulator

Ein einfacher Simulator für Fahrstühle ist das S-Ring-Modell. Hierbei wird davon ausgegangen, dass die Fahrstuhlwagen sequenziell (S-Ring-Model = Sequential Ring-Model) immer vom Untersten zum obersten Stockwerk auf- und anschließend wieder ab fahren. In der Abbildung 25 kann man die Ring-Struktur dieses Modells erkennen. So gibt es für jedes Stockwerk (mit Ausnahme des untersten und des obersten) je eine „Station“ für Fahrstühle in Aufwärts- und in Abwärtsrichtung. Der Zustand einer „Station“ wird definiert über das Vorhandensein mindestens eines wartenden Passagiers und ob an dieser Position ein Fahrstuhlwagen vorhanden ist, oder nicht. Daher wird jede „Station“ durch zwei Bits beschrieben, sodass sich der Zustand eines S-Rings zum Zeitpunkt t durch einen binären Vektor beschreiben lässt.
In jeder Iteration der Simulation errechnet der Simulator den Zustand des gesamten Systems zum Zeitpunkt $t + 1$, beeinflusst durch die Ankunftswahrscheinlichkeit neuer Fahrgäste (Hall-Call-Probability), der Anzahl der Fahrstuhlwagen im System und der Anzahl der Stockwerke im Gebäude.
Das Verhalten des Systems während der Simulation wird durch einen Gewichtsvektor beschrieben, der für jedes Bit jeder „Station“ eine Gewichtung enthält.
Zunächst ist dieses Anwendungsproblem einkriteriell. Sobald man aber das Verhalten des Fahrstuhlsystems zu verschiedenen Stoßzeiten (= mit verschiedenen Fahrgast-Ankunftswahrscheinlichkeiten) optimieren möchte, wird das Problem für n Hall-Call-Probabilities n-kriteriell.
Der konkret in diesem Projekt eingesetzte und ausgelieferte Simulator ist eine auf [29] basierende und an die Anforderungen dieses Projekts angepasste Implementierung, die jedoch zum Zeitpunkt der Durchführung dieser Versuche noch nicht umfassend getestet und ausgearbeitet war. Interessierte können per Mail [30] nähere Informationen sowie eine aktuelle Version anfordern.

5.4.2.4 Anwendung des Simulators/der Schnittstelle

Der im Lieferumfang befindliche S-Ring-Simulator kann mit allen notwendigen Optionen – unabhängig von *NObELTJE* – von der Kommandozeile aus aufgerufen werden. Es werden die folgenden Parameter in genau dieser Folge erwartet (Erklärung in Tabelle 3):
`ring n m n_iteration l sd ts` w_1 w_2 w_3 `...` w_k

Das Ergebnis wird als Gleitkommazahl per Standard-Output zurückgegeben. Da kleinere Werte zu bevorzugen sind, handelt es sich hierbei um ein Minimierungsproblem.

Um das S-Ring-Modell in das *NObELTJE*-Framework integrieren zu können, existiert eine in Java implementierte Schnittstelle (`simulator.elevatorcon.SRingCon`). Diese kann bei einem Aufruf über das Design oder einen Algorithmus zur Steuerung des Simulators die folgenden Optionen erhalten:

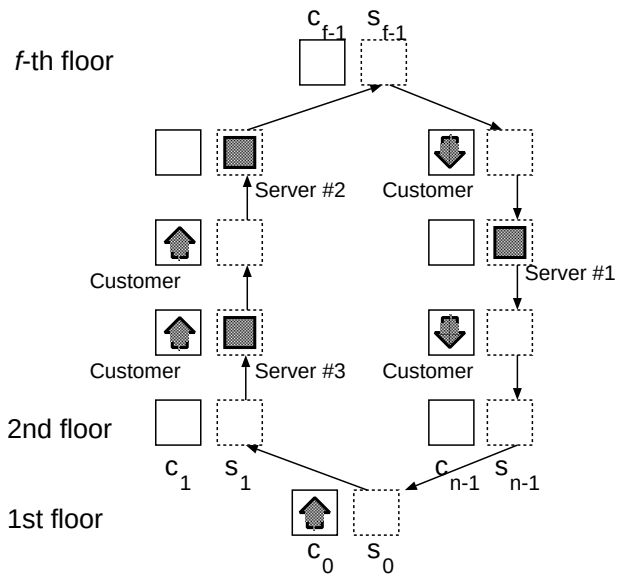

Abbildung 25: Schematische Darstellung des SRing-Modells [31]: Der binäre Zustand C_n beschreibt die Anwesenheit wartender Fahrgäste für die angegebene Fahrtrichtung, während das Bit S_n bestimmt, ob in dem jeweiligen Stockwerk ein Fahrstuhlwagen vor Ort ist, der in diese Richtung fährt.

- `-tti`: Anzahl der vom Simulator zu verwendenden Iterationen (Standard: 10000)
- `-ttn`: Anzahl der vom Simulator zu verwendenden Stockwerke (Standard: 6)
- `-ttm`: Anzahl der Fahrstuhlwagen im zu simulierenden Gebäude (Standard: 2)
- `-ttp`: Wahrscheinlichkeiten für Ankunft einer Person auf einem Stockwerk. Mehrere Angaben müssen mit „:“ getrennt werden (z. B. `-ttp 0.3:0.4`). Hiermit wird gleichzeitig die Anzahl der Kriterien festgelegt (Standard: `0.3:0.4`)
- `-ttr`: Random-Seed (Standard: randomisiert)
- `-ttd`: Debug-Modus ($0-7$, Standard 2)
- `-ttpath`: Pfad zum Simulator (nur notwendig, wenn nicht die *NObELTJE*-Verzeichnisstruktur gegeben ist oder ein anderer Simulator verwendet werden soll).

5.4.2.5 Besonderheiten der Schnittstelle

Ein wesentlicher Unterschied zur Anbindung einer klassischen Testfunktion stellt die Verfügbarkeit des Simulators im C-Quellcode (im Dateisystem: `simulator/FahrStuhl/ringR.c`) dar. Während die Testfunktionen einfach als Java-Klasse eingebunden werden können, muss dieser Simulator separat kompiliert

Tabelle 3: Kommandozeilenparameter des S-Ring-Simulators

Position	Parameter	Beschreibung
1	`n`	Anzahl der Stockwerke
2	`m`	Anzahl der Wagen
3	`n_iteration`	Anzahl der Iterationen
4	`l`	Wahrscheinlichkeit für Hall-Call auf einem Stockwerk
5	`sd`	Random-Seed
6	`ts`	Debug-Ausgaben-Flag (1=ja, 0=nein)
7...k+6	w	Gewichte des Gewichtsvektors

und über eine Schnittstellenklasse angesteuert werden. Ein beiliegendes Makefile erlaubt jedoch einen unkomplizierten Aufruf von „make“ (sofern ein C-Compiler installiert wurde), um eine ausführbare Binary-Datei zu erhalten. Alternativ versucht die Schnittstellenklasse des Simulators, eine ausführbare Datei zu erzeugen, falls noch kein Kompilat vorliegt. Auf Linux-Systemen oder unter Windows mit Cygwin konnte der Code problemlos übersetzt werden. Detailliertere technische Informationen und Hinweise zum verwendeten Modell und der Schnittstelle können in einem README, welches sich im *NObELTJE*-Paket im Verzeichnis des Simulators befindet, nachgelesen werden.

5.4.2.6 Nachbearbeitung der Ergebnisse

Da der Simulator selbst stochastisch arbeitet, ergeben sich bei unterschiedlicher Initialisierung der Zufallszahlen Ergebnisse, die bei geringerer Iterationsanzahl im Simulator zunehmend ungenauer werden. Um fundierte Ergebnisse zu erlangen, ist es nach Beendigung des Algorithmus notwendig, die gefundenen Lösungen erneut, aber aufwändiger, zu simulieren. Details zu dieser Problematik und die implementierte Lösung werden in Kapitel 7.4.2.1 näher untersucht.

5.4.3 Temperierbohrung

Eine weitere Praxisanwendungen, der sich mit den Algorithmen aus dem *NObELTJE*-Paket genähert werden sollte, war das Temperierbohrungsproblem.
In diesem Abschnitt soll zunächst das Temperierbohrungsproblem allgemein vorgestellt werden. Damit ist eine grobe Problemdarstellung gegeben, welche gleichzeitig die Motivation darstellt, dieses Problem möglichst gut zu lösen.
Im darauf folgenden Abschnitt wird der Simulator vorgestellt, mit dem gearbeitet wurde. Der letzte Abschnitt beinhaltet, wie der Simulator mit dem Namen „*Evolver*“ in *NObELTJE* eingebunden wurde und welche Probleme es dabei gab.

5.4.3.1 Die Problematik der Temperierbohrungen

Das Temperierbohrungsproblem taucht bei der Herstellung und Fertigung von Formteilen auf. Es ist häufig im Maschinenbau zu finden, wo sich mit der Fertigung von Werkstücken beschäftigt wird. Zur Herstellung werden Stahlformen mit Negativ der zu erzeugenden Form verwendet. Diese Form besteht meistens aus zwei Hälften, die aufeinandergelegt werden und somit einen Hohlraum bilden. In der Abbildung 26 ist eine dieser Stahlformen zu sehen. Füllt man diesen Hohlraum mit dem flüssigen Werkstoff auf, so erhält man das gewünschte Werkstück nach dem Auskühlen und Heraustrennen aus der Gussform. Das Problem dabei ist, dass die Formeinsätze in der Industrie oft und vor allen Dingen auch schnell wieder einsatzfähig sein müssen, um möglichst viele Werkstücke in kürzester Zeit herzustellen. Die Zeit, wie lange die Gussform benutzt wird, hängt von der Zeit ab, die der Werkstoff zum Auskühlen braucht. Erst wenn der Werkstoff abgekühlt und hart geworden ist, kann man ihn aus den Formeinsätzen nehmen. Um die Form, und damit das Formteil zu kühlen, wird in der Praxis im Allgemeinen eine Kühlflüssigkeit aus einem Wasser-Öl-Gemisch verwendet, die durch Kühlschleifen in der Stahlform fliesst. Diese Kühlschleifen nennt man daher auch Temperierbohrungen. Diese werden mit einem Spezialbohrer in die Stahlform eingebracht. Die Bohrungen werden zum Teil von außen wieder verschlossen. Die nicht verschlossenen Bohrlöcher werden mit Anschlüssen für die Kühlflüssigkeit versehen. Somit erhält man ein Leitungssystem (siehe auch Abbildung 26).
Eine beliebige Kühlschleife in die Stahlform zu bohren, ist aus verschiedenen Gründen nicht praktikabel. Die Güte einer Kühlschleife hängt von verschiedenen Faktoren ab:

- Die Kühlung sollte gleichmäßig sein, da sich sonst das Werkstück verziehen kann oder Spannungen auftreten können, wodurch das Werkstück unbrauchbar wird.
- Das Werkstück sollte möglichst schnell gekühlt werden, damit die Formteile schnell wieder einsetzbar sind.
- Trivialerweise sollte die Kühlbohrungen nicht durch das Werkstück verlaufen.
- Die Kosten in der Herstellung der Kühlschleife sollten möglichst gering sein. Die Kosten sind dabei über die Bohrungslänge definiert. Je länger die Bohrungen gemacht werden müssen, umso mehr nutzt sich der Bohrer ab. Gerade bei Bohrungen durch Stahl kommt es manchmal vor, dass die Bohrer abbrechen oder schnell abgenutzt werden.

– Die Form darf bei der Einfüllung des flüssigen Werkstoffes eine Mindesttemperatur nicht unterschreiten, da sonst die Werkstoffflüssigkeit zu schnell abkühlt und damit nicht mehr flüssig genug ist, um den Hohlraum komplett auszufüllen.

Die Parameter wie der Abstand des Formteils zu der Kühlschleife, Durchmesser der Bohrungen, Temperatur der Kühlflüssigkeit und die Fließgeschwindigkeit sind weitere ausschlaggebende Faktoren für die Güte der Kühlung.

Ziel ist es, die Abkühlzeit mittels der Temperierbohrungen auf ein Minimum zu reduzieren, um die Formeinsätze möglichst schnell wieder benutzbar zu machen. Dabei soll die Temperierbohrung so angelegt sein, dass die Qualität des Werkstücks durch die gezielte Kühlung nicht sehr beeinträchtigt wird.

Ein Beispiel für eine Spritzgussform (entnommen aus [32]) ist in der Abbildung 26 zu sehen. Auf diesem Bild ist der eine Teil der Gussform zu erkennen. Unten sieht man die Anschlüsse für die Kühlflüssigkeit. Das fertige Werkstück, eine Abdeckung für ein Autoradio, ist ebenfalls mit abgebildet.

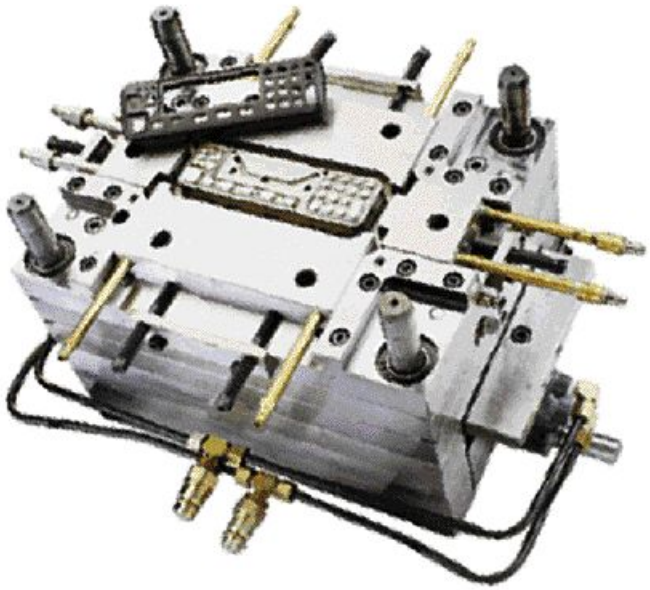

Abbildung 26: Gussform für eine Autoradioabdeckung

5.4.3.2 Der Evolver

Um dieses Problem mittels Algorithmen approximativ lösen zu können, hat das Institut für Spanende Fertigung (ISF) einen Simulator für das Temperierbohrungsproblem namens *Evolver* zur Verfügung gestellt. Der *Evolver* ist eine Blackbox. Diese Blackbox versucht Kühlwirkung der Temperierbohrungen zu simulieren, ist in C++ geschrieben und nur unter Windows ausführbar. Die Eingabe erfolgt über eine Schnittstelle, auf

die über einen Socket zugegriffen werden kann. Um diese Schnittstelle auch für Java-Programme zugänglich zu machen, wurde von einer studentischen Hilfskraft des Lehrstuhls des Instituts für Spandende Fertigung eine Implementierung einer Schnittstellenklasse vorgenommen, die weiterentwickelt und für *NObELTJE* lauffähig gemacht worden ist.

Die Eingabe für den *Evolver* ist eine mögliche Temperierbohrung, die als ein Polygonzug abstrahiert wird. Die Anzahl der Eingabewerte ist von der Bohrungsanzahl abhängig. Für jeden Bohrungsendpunkt sind drei Werte als Eingabe notwendig, da jeder Bohrungsendpunkt im drei-dimensionalen Raum angegeben wird. Dieses bedeutet bei n Bohrungen, dass insgesamt $3n + 3$ Werte für den Polygonzug übergeben werden müssen. Als Ergebnis wird ein 12-dimensionaler Vektor zurückgegeben, der aus den in der Tabelle 4 dargestellten Rückgabewerten besteht.

Tabelle 4: Rückgabewerte des *Evolvers*

Kühlungwerte		
1	thermal effect aggregated	aggregierte Kühlungswerte
4	thermal effect wort value	schlechtester Kühleffekt
5	thermal effect average	durchschnittlicher Kühleffekt
6	thermal effect variance	Varianz der Kühlungseffekte
7	thermal effect best value	bester Kühlungswert
Restriktionen		
3	cost aggregated	aggregierte Kosten
8	restriction angle of drilling holes	Ist der Winkel zwischen 2 benachbarten Bohrungen zu flach?
9	restriction angle of outer surface	Ist der Winkel zwischen 2 Bohrungen und Aussenfläche zu flach?
10	restriction bounding box	Ist der Polygonzug im Werkstück?
11	restriction drilling	Schneidet sich der Polygonzug selbst?
12	restriction object	Schneidet der Polygonzug das Werkstück?
Kosten		
2	cost aggregated	aggregierte Kosten

Die erste Spalte gibt die Komponente des Rückgabewertes im Vektor an. Hier wäre z. B. die „restriction bounding box“ im Vektor der zehnte Wert.
Wichtig ist noch, dass die Restriktionen nur bei bestimmten Werten einen gültigen, also zulässigen Polygonzug angeben. Die „restriction aggregated“ muss den Wert 1 annehmen, die anderen Restriktionswerte (im Rückgabevektor die Koordinaten 8, 9, 10, 11, 12) müssen 0 sein, wenn der Polygonzug gültig ist.

Der *Evolver* hat eine Einheitshalbkugel (siehe auch Abbildung 27) als Standard-Werkstück. Diese Halbkugel liegt in einem 3-dimensionalen Raum, der die Ausmaße von 4x4x2 Längeneinheiten hat. Die x-Koordinate eines jeden Bohrungspunktes muss im Intervall $[-2, 2]$ liegen, ebenso die y-Koordinate. Die z-Koordinaten eines jeden Bohrungspunktes müssen im Intervall $[0, 2]$ liegen.

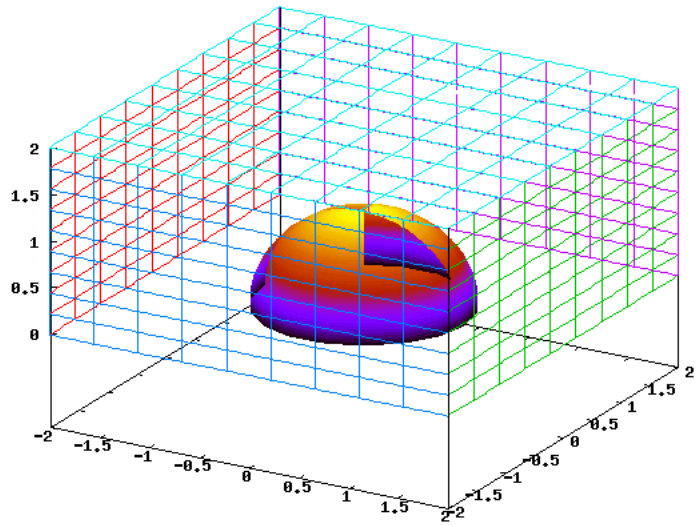

Abbildung 27: Einheitstestwerkstück des *Evolvers*

5.4.3.3 Anbindung des Simulators

Kommunikation mit dem Simulator Der Simulator für das Temperierbohrungsproblem ist ein unter Windows laufender Server. Die Kommunikation läuft über eine Socket-Schnittstelle über die Signale sowie Daten an den Server gesendet werden können. Der Server kann dabei auf einem beliebigen Windowsrechner im Netzwerk laufen und wird von einer Schnittstellenklasse via IP und Portnummer angesprochen. Der Standardport des Server ist 6201, kann aber beliebig angepasst werden. Es ist nicht empfehlenswert, den Server auf einem anderen Rechner laufen zu lassen als den Algorithmus, da selbst ein 100 Mbit Netzwerk die Berechnung sehr verlangsamt.

Folgende Funktionalität wird vom Server angeboten:

- Um die Verbindung zu überprüfen, kann ein Ping an den Server geschickt werden.
- Die Fitnesswerte zu einem Polygonzug können berechnet werden.
- Ein Startindividuum, also ein gültiger Polygonzug, wird vom Server geliefert.
- Ein ungültiger oder schlechter Polygonzug, kann, wenn er in gewissen Grenzen liegt, repariert werden.
- Der Server liefert die Namen der zu optimierenden Fitnesswerte. Diese sind in der zweiten Spalte der Tabelle 4 nachzulesen.
- Eine Verbindung zum Server kann beendet und der Server bei Bedarf heruntergefahren werden.

Die folgenden Kapitel gehen näher auf die Details der wesentlichen Funktionen ein.

Evaluation Der Server empfängt das Signal 1003 und weiß damit, dass ein Polygonzug folgt, der bewertet werden soll. Der Polygonzug wird dem Simulator als eine einfache Folge der x-, y- und z-Koordinaten eines jeden Eckpunktes des Polygonzugs übergeben. Danach folgt für jede Bohrung ein Boolean-Wert, der die Richtung der Bohrung angibt. Die Optimierung dieser Boolean-Werte wird in *NObELTJE* dem Server überlassen, weswegen in diesem Fall immer nur false als Dummy-Wert geschickt wird. Die Optimierung der Boolean-Werte geschieht mittels der Repair-Funktion (siehe auch 5.4.3.3).
Vom Server werden dann die 12 Fitnesswerte zurückgeschickt, die bereits vorgestellt wurden.

Erzeugen von Startindividuen Ein Startindividuum wird vom Server erzeugt, wenn er das Signal 1004 empfängt. Der Polygonzug wird zufällig berechnet und hält sich an die Begrenzungen des Werkstücks. Somit ist garantiert, dass der Server fähig ist, zu diesem Polygonzug die entsprechenden Fitnesswerte zu berechnen. Auf zu weit außenliegende Punkte, wie sie ein Algorithmus rein zufällig ohne Beschränkungen erstellen würde, kann der Server mit Abstürzen reagieren.

Die Repair-Funktion Das Reparieren eines Individuums ist nur begrenzt möglich. Zwei wesentliche Bedingungen werden dabei überprüft:

1. Die Richtung der Bohrungen wird verändert, falls dadurch ein gültiger Polygonzug entsteht. Auf diese Weise können z. B. unerwünschte Schnittpunkte von Bohrungen vermieden werden.

2. Die Anfangs- und der Endpunkte des Polygonzugs, der die Bohrung darstellt, werden an den Rand des Werkstücks gelegt.

Diese Funktion wird genutzt, um die Bohrungsrichtungen vom Server automatisch korrekt erzeugen zu lassen. Die Eckpunkte des Polygons werden nicht repariert, da keine Möglichkeit besteht, den Algorithmus mit den veränderten Punkten weiter rechnen zu lassen. Somit hätte die Möglichkeit bestanden, dass ein schlechtes Individuum aufgrund der Reperatur einen guten Fitnesswert bekommt, der Algorithums aber weiterhin mit dem schlechten Individuum rechnet. In der in *NObELTJE* implemtierten Fassung der Simulatoranbindung für das Temperierbohrungsproblem wird ein vom Algorithmus generiertes Individuum, bevor es simuliert wird, immer erst repariert, um die korrekten Bohrungsrichtungen zu erhalten.

Einbindung des Servers Aus dem Projekt KEA exisitierte bereits eine grobe, noch nicht funktionstüchtige Anbindung einer älteren Version des Servers an ein Java-Projekt. Diese Klasse wurde als Basis genutzt und angepasst. Dabei wurden wesentliche Teile der ursprünglichen Klasse stark verändert und teilweise komplett neu geschrieben.
Eine Anforderung an die Schnittstelle zwischen den Algorithmen und dem Simulator war, einen beliebigen, uneingeschränkten Polygonzug senden zu können und für diesen die entsprechenden Fitnesswerte zurück zu bekommen, ohne das sich der Algorithmus um Definitionsbereiche oder Bohrrichtungen kümmern muss. Außerdem sollte

zur einfachen Anbindung des Simulators an sämtliche bereits entwickelten Algorithmen das Interface von Simulatoren und Testfunktionen das selbe sein.
Sowohl die IP-Adresse des Servers, als auch der Port und die Anzahl der Bohrungen kann einfach durch die Konfigurationsdatei Temp.param eingestellt werden, die im gleichen Verzeichnis wie der Simulator liegt. Nur die Anzahl der Bohrungen läßt sich einem Algorithmus auch direkt als Parameter übergeben, sodass z. B. in einem Design auch für mehrere Bohrungsanzahlen optimiert werden kann. Parameter, die von dem Algorithmus nicht selber interpretiert werden, sondern nur an den Simulator weitergegeben werden sollen, fangen immer mit `-tt` an. Um die Anzahl der Bohrungen zu setzen, muss also `--ttboringCount` in der Komandozeile auf den gewünschten Wert gesetzt werden.
Der Server verlangt nach dem Durchlauf eines Algorithmus, dass die Verbindung beendet wird. Andererseites wartet er weiter auf Anfragen, wobei er nach kurzer Zeit abstürzt. Um also mehrere Algorithmenläufe, z. B. für ein Design, hintereinander starten zu können, war es notwendig, am Ende vom Algorithmus eine finalize-Methode aufrufen zu lassen. Diese übernimmt dann den Verbindungsabbau und erzwingt eine gewisse Wartezeit, damit sich der Server neu starten kann, bevor der nächste Algorithmenaufruf wieder versucht, sich zu verbinden.

5.4.3.4 Probleme bei der Anbindung

Um einen Algorithmus so einzurichten, dass er nicht nur mit den Testfunktionen, sondern auch mit dem Simulator funktioniert, sind einige zusätzliche Handgriffe nötig. Folgende Probleme ergaben sich:

Initialisierung des Servers Die Methode `init()` der Schnittstellen-Klasse `Temp.java` muss aufgerufen werden, sobald der Algorithmus erfragt hat, mit welchen Parametern er gestartet werden soll und seine eigene Initialisierung abgeschlossen ist. Das sollte nach dem Aufruf des Konstruktors, bzw. am Anfang der Methode `startAlgo()` geschehen, die von jedem Algorithmus aufgrund des Interfaces implementiert werden muss. Über die init-Methode werden dann z. B. die Anzahl der Bohrungen, die in diesem Lauf genutzt werden sollen, an den Server übergeben.

finalize-Methode Die finalize-Methode ist immer dann wichtig, wenn mehr als ein Algorithmusaufruf automatisiert gestartet werden soll. Wie oben bereits erwähnt, wird hier die Verbindung zum Server beendet und auf einen Neustart gewartet. Dieser Neustart wird vom Server automatisch ausgeführt und beginnt mit einer missverständlichen Fehlermeldung, die besagt, dass der *Evolver* sich nicht an seinem Port konnektieren kann. Diese Meldung ist aber nur ein Test, ob der Port auch frei ist. Danach nimmt der Server diesen freien Port für seine Kommunikation und startet normal. Die finalize-Methode muss als letzter Befehl in der Methode `startAlgo()` des Algorithmus stehen. Nur so kann garantiert werden, dass der Simulator weiterhin fehlerfrei läuft.

Nachkommastellen Leider können von dem Simulator nur Individuen simuliert werden, die gewissen Mindestansprüchen genügen. Speziell muß darauf geachtet werden, dass zwei Punkte eines Polygons nicht übereinander liegen. Dieser Fall wird in

der Schnittstellenklasse bereits überprüft und mit einem schlechten Fitnesswert bestraft (5.4.3.4). Da der Simulator aber nur mit maximal drei Nachkommastellen Genauigkeit arbeitet, liegen für ihn auch Punkte übereinander, die sich nur in den hinteren Nachkommastellen voneinander unterscheiden (Beispiel sind die Punkte 0-0-0 und 0-0-0.002 identisch). In den gestarteten Versuchen war der BasisGA der einzige Algorithmus, der in so kleinen Schritten gesucht hat, sodass diese Situation auftreten konnte. Die Schnittstellenklasse wurde daraufhin entsprechend erweitert und kann mit solchen Fällen umgehen. Der Algorithmus selbst muss sich also an keine Beschränkungen halten.
Da es sich bei dem im Server simulierten Werkstück um einen Quader mit den Seitenlängen von $40 \cdot 40$cm handelt, wird mit drei Nachkommastellen immer eine Genauigkeit von $1/10$mm erreicht, was für die Praxis ausreichend ist.

Bestrafungsfunktion Die Algorithmen aus dem *NObELTJE*-Paket halten sich nur während der Erzeugung der Startindividuen an die von dem Simulator vorgegebenen Quader-Schranken. Sobald die Suche gestartet ist, werden diese Grenzen nicht mehr berücksichtigt. Da es nicht sinnvoll ist, Polygonzüge zu betrachten, deren Punkte außerhalb des Werkstücks liegen, wurde für diese Punkte eine Bestrafungsfunktion eingefügt.
Für Bestrafungsfunktionen gibt es viele Möglichkeiten. Nach einiger Literaturrecherche ([18], Kapitel 7) fiel die Entscheidung für eine Bestrafungsfunktion, die sich leicht in *NObELTJE* integrieren ließ. Umgesetzt wurde letztendlich folgende Lösung:
Sobald ein Punkt außerhalb des Werkstücks liegt, wird seine Distanz zum Werkstück in x-, y- und z-Richtung berechnet. Der zurückgegebene Fitnessvektor enthält dann ($2^{DistanzX_1} + 1$, $2^{DistanzY_1} + 1$, $2^{DistanzZ_1} + 1$, $2^{DistanzX_2} + 1$, ...), wobei X_n der Wert der n-ten x-Koordinate des Polygonzugs ist.
Diese Bestrafungsfunktion wächst exponentiell mit der Distanz und erreicht so schnell hohe Bestrafungswerte. Außerdem bleibt auch für Algorithmen, die nach Fronten selektieren, eine interpretierbare Paretofront erhalten. Diese Bestrafungsfunktion wurde mit den in *NObELTJE* zur Verfügung stehenden Algorithmen getestet und hat gute Ergebnisse erzielt. Schon nach wenigen Durchläufen werden keine – aufgrund von außenliegenden Punkten – ungültigen Individuen mehr erzeugt.

5.5 Design und Tool-Klassen

Folgende Tools wurden entwickelt, um das Zusammenspiel der Module zu verbessern oder überhaupt erst zu ermöglichen.

5.5.1 Design

5.5.1.1 Motivation

Mit Hilfe des Design-Moduls kann der Anwender einen Satz von vielen Algorithmen-Aufrufen mit systematisch variierten Parametern veranlassen. Dies ist hilfreich, weil für die Erforschung des Verhaltens randomisierter Heuristiken viele Experimente notwendig sind, da allgemeine theoretische Ergebnisse schwierig zu erlangen sind. Insbesondere bei den natur-inspirierten Verfahren, die oft viele Parameter enthalten, ist

es schwierig, Algorithmen zu vergleichen, denn die Qualität der Optimierung hängt – neben dem Problem, auf das sie angewendet werden – auch stark von der gewählten Parameterbelegung ab. Die Problematik, ein geeignetes Qualitätsmaß für eine von einem Algorithmus produzierte Lösungsmenge zu finden, soll durch die Metriken gelöst werden (siehe 5.2). Hier soll nun durch das Design der Experimente ein faires Szenario geschaffen werden, damit der tatsächlich beste Algorithmus auch die besten Ergebnisse liefern kann. Dazu muss man zunächst herausfinden, wie die besten Parametereinstellungen für einen Algorithmus sind. Will man zwei Algorithmen vergleichen, muss man die Testfunktion, bzw. das zu optimierende Problem und die zur Verfügung stehenden Ressourcen (z. B. Anzahl der Zielfunktionsauswertungen) geeignet wählen. Unter den Ergebnissen der via Parametereinstellung variierten Algorithmen kann die günstigste Einstellung ermittelt werden. Zusätzlich können durch eingehende Analyse Zusammenhänge zwischen den Parametern aufgedeckt werden.

5.5.1.2 Syntax für die Benutzung

Das Design-Modul ermöglicht dem Anwender, Algorithmen mit verschiedenen Parametereinstellungen automatisiert im Sinne einer statistischen Versuchsplanung aufzurufen. Hierzu steht eine Syntax zur Verfügung, mit der Parameterbelegungen als Mengen oder in Form von Intervallen angegeben werden können. Bei der Intervallschreibweise kann zwischen der Angabe eines Abstandes der Elemente und der Anzahl der Elemente gewählt werden.
Diese Syntax ist Teil des Aufrufs des Design-Moduls und so kann durch einen Aufruf ein ganzer Satz von Experimenten angestoßen werden.
Bei jedem Aufruf schreibt der benutzte Algorithmus seine Ergebnisse in eine Datei, die vom Design-Modul mit einem Namen versehen wird, der seine Parameter-Belegung wiedergibt. Durch Auslesen dieses Namens kann dann das Analyse-Modul die Parameter-Belegung nachvollziehen (siehe Kapitel 5.3).
Beispiel-Aufruf:

```
java design.Design --t testfkt.FonsecaF1 --a OnePlusOnePG
-g s 100 1000 -archive i 10 100 step 10 -deviation i 0.1 1
count 5
```

Dieser Aufruf würde dann den Algorithmus `OnePlusOnePG` auf der Testfunktion `FonsecaF1` aufrufen und die Parameter des Algorithmus entsprechend variieren, so dass `g` die Werte 100 und 1000, `archive` die Werte von 10 bis 100 in Zehnerschritten und `deviation` insgesamt 5 Werte zwischen 0.1 und 1 annimmt.

5.5.1.3 Anwendung im Projekt

Das Design-Modul wurde im zweiten PG-Semester genutzt, um das Verhalten von Algorithmen bei bestimmten Parametereinstellungen zu analysieren und so allgemeine Erkenntnisse zu gewinnen, die vielleicht die Verbesserung von Verfahren ermöglichen. Erweiterungen zu diesem Tool waren geplant, wurden aber nicht mehr, oder nur indirekt, umgesetzt. Eine Erweiterung in Richtung der folgenden Punkte könnte aber weiterhin sinnvoll sein.

- Planung der Experimente
  Für das Factorial-Design wurde diese Klasse nicht mehr extra erweitert sondern

vom Benutzer erwartet, dass er die zu variierenden Parameter bereits im Sinne eines Factorial Designs angibt. Auf diese Weise lassen sich die Algorithmenaufrufe für ein Factorial Design auch mit Hilfe der aktuellen Designklasse starten.

- Constraints
  Da nicht alle Parameterkombinationen, die durch Permutation der Angaben entstehen, auch von den Algorithmen realisiert werden können, könnte ein Constraint-System eingeführt werden um zu überprüfen, ob die Parametereinstellung für das entsprechende Verfahren gültig ist. Eine einfaches Beispiel für eine Bedingung bei einer (μ, κ, λ)-ES wäre: $\mu \leq \lambda$, falls $\kappa = 1$.
  Die Notwendigkeit diese Erweiterung umzusetzen hat sich nicht ergeben, da es sich bei dem oben genannten Beispiel bis jetzt um das Einzige handelt. Verletzungen solcher Constraints liegen dabei wohl auch meistens in Bereichen, deren Untersuchung nicht interessant ist, da dort keine guten Werte vermutet werden.

- Batch-Modus
  Da die Experimente unter Umständen sehr rechenintensiv sind, könnte man das Design-Modul auch im Batch-Modus ausführbar machen.
  Diese Funktionalität musste nicht von der PG selbst umgesetzt werden, da am Lehrstuhl 11 bereits ein Batchsystem existiert, dem man solche Aufgaben übertragen kann. Dieses Batch-System wurde für die von uns ausgeführten Experimente ausführlichst genutzt.

5.5.2 Die Tool-Klassen

Viele Algorithmen nutzen gleiche Mechanismen. So wird z.B. jeder Algorithmus eine Dateiausgabe benötigen oder die bereits bewerteten Individuen bzgl. Pareto-Dominanz untersuchen.
Um hier unnötige Fehlerquellen auszuschließen und doppelte Arbeit zu vermeiden, haben wir bereits zu Beginn unseres Projekts ein eigenes Paket namens `tools` angelegt, in dem wir vermeintlich wiederverwertbare Teile für die Algorithmen, sowie allgemeine Hilfsklassen gesammelt haben. Dadurch ist eine Sammlung entstanden, deren Funktionen im Folgenden näher erklärt werden sollen.

5.5.2.1 Allgemeine Hilfsklassen

Individual Fast sämtliche der von uns implementierten Algorithmen arbeiten in irgendeiner Weise mit Populationen von Individuen. Diese bestehen meistens aus einer Liste von Objektvariablen, den dazugehörigen Fitnesswerten und eventuellen Zusatzdaten, wie z.B. dem Alter des Individuums in Form der Generation oder einer Liste von Strategievariablen.
Diese Daten wurden von uns gekapselt und mit einfachen Zugriffsfunktionen versehen, welche die komfortable Einbindung in die Algorithmen erleichtert. Zusätzlich arbeiten viele weitere Hilfsklassen, wie z.B. das `ParetoArchive`, auch mit dieser Darstellung eines Individuums, so dass auf diese Weise eine einheitliche Kommunikationsschnittstelle zwischen den Klassen gewährleistet ist.

5.5.2.2 I/O-Tools

Im Speziellen gab es bei der Entwicklung unseres Projekts viele Stellen, an denen sich mit der Ein- und Ausgabe beschäftigt werden musste. Um hier einheitlich und damit untereinander kompatibel zu bleiben, wurden die hierfür benötigen Klassen ins Paket `tools.io` ausgelagert und von allen Algorithmen gemeinschaftlich genutzt.

FilterAndDelete.java Diese Klasse hat sich im wesentlichen aus zwei Gründen entwickelt. Da wir schon während der Entwicklungsphase einige Experimente gestartet haben, hat sich an einigen Stellen ein inkompatibles Zahlenformat gezeigt. Um solche Daten trotzdem hinterher nutzen zu können, mussten sie gefiltert und gegebenfalls konvertiert werden. Unser Projekt arbeitet ausschließlich mit einem Punkt als Dezimaltrennzeichen. Die Tausenderstellen werden nicht durch ein Trennzeichen markiert. Also sollten in den zu bearbeitenden Dateien keine Kommata vorkommen.
Außerdem werden beim Temperierbohrungsproblem (siehe 7.4.1) teilweise ungültige Individuen erzeugt, deren Punkte z. B. außerhalb der Gussform liegen, oder die durch das zu gießende Werkstück gehen. Diese Bohrungen könnten das Ergebniss verfälschen und sollten somit ausgefiltert werden. Leider war es nicht möglich, die Ergebnisse aus dem Temperierbohrungsproblem bezüglich aller 12 Fitnesswerte zu interpretieren, da die einzige Metrik, die von uns für mehr als 2-dimensionale Fitnesswerte implementiert wurde, leider eine zu hohe Laufzeit hat.
Da diese Klasse teilweise Daten ohne Rückfrage und Sicherheitskopie löscht, ist es wichtig, vorher eine eigene Sicherheitskopie anzulegen.
Aus den genannten Problemen ergaben sich folgende Funktionen:

- `-d`
  Angabe des zu bearbeitenden Verzeichnisses.
- `-del_Comma`
  Entfernt sämtlich Kommata! Das ist bei Zahlen im Format 1,234.567 sinnvoll.
- `-del_Invalid`
  Entfernt ungültige Individuen vom Temperierbohrungsproblem und ist somit für sämtliche anderen Probleme nicht interessant. Ungültig heißt in diesem Fall, dass in der dritten Spalte ein Wert größer als eins oder in den Spalten acht bis zwölf ein Wert größer als null steht.
- `-convert point` oder `-convert comma`
  Es werden alle Kommata zu Punkten oder bei Verwendung des Parameters `point` alle Punkte zu Kommata konvertiert (evtl. sollte zuvor erst `del_Comma` ausgeführt werden).
- `-select 1,2,3,6`
  Aus den Fitnesswerten werden die angegebenen Spalten gelöscht. Es gibt keine Überprüfung, ob die Spalten existieren. Die erste Spalte hat die Nummer eins.
- `-undo`
  Kopiert alle `.old`-Dateien zurück zu ihrem ursprünglichen Dateinamen.

Natürlich ist das nur ein echtes „Undo“ wenn in den `*.old` noch die unveränderten Daten stehen.

Help.java Diese Klasse ruft nur die Methode `toString()` der angegebenen Klasse auf und gibt dessen Rückgabe auf der Konsole aus. Auf diese Weise können zu fast jeder Klasse Hilfen abgerufen werden.
Dazu muss die Klasse beim Aufruf mit Paketnamen angegeben werden, also mit Punkt-Trennung zwischen Paketen, z. B.:
`java tools/io/Help algo.MueRhoLES`.
Die Hilfen enthalten z. B. bei den Algorithmen die korrekten Aufrufe sowie Beispielparameter in sinnvollen Bereichen.

6 NObELTJE in der Praxis

Neben einer Einleitung, wie Algorithmen aufgerufen und ganze Versuche mit *NObELTJE* durchgeführt und analysiert werden, soll in diesem Kapitel näher auf die Möglichkeiten eingegangen werden, das System zu erweitern. Abschließend wird vorgestellt, wie *NObELTJE* zu beziehen ist und unter welchen lizenzrechtlichen Bedingungen es genutzt und erweitert werden darf.

6.1 Durchführung von Experimenten

Im Folgenden wird exemplarisch erklärt, wie man einen Versuch mit unserer *NObELTJE*-Software durchführen kann. Dabei wird Schritt für Schritt gezeigt, wie ein Algorithmus gestartet wird und welche Ausgaben man erhält. Es wird auch vorgestellt, wie ganze Versuchsreihen einfach gestartet werden können und wie man die umfangreichen Ausgaben mit den Analysetools analysieren kann.

6.1.1 Aufruf eines Algorithmus

Am Beispiel des `OnePlusOnePG` wollen wir exemplarisch den Aufruf eines Algorithmus zeigen. Wird eine Entwicklungsumgebung, wie z. B. *Eclipse* benutzt, ist es von dieser Umgebung natürlich abhängig, wo die entsprechenden Befehle und Parameter-Werte eingegeben werden müssen. Im Folgenden werden die Beispiele mit dem Kommandozeilen-Befehl `java` veranschaulicht. Mit der Klasse `Help` kann zu jeder Klasse im Projekt eine kurze Beschreibung und Bedienungsanleitung ausgegeben werden, zu der auch die benötigten Parameter gehören:

```
java tools/io/Help algo.OnePlusOnePG
```

Die erste Möglichkeit einen Algorithmus zu starten, ist der direkte Aufruf:

```
java algo.OnePlusOnePG -t testfkt.Deb -g 100 -archive 500
-deviation 1.0
```

Es ist bei jedem Algorithmen-aufruf notwendig, die gewünschte Testfunktion mit dem Parameter `-t` anzugeben. Abhängig vom ausgewählten Algorithmus müssen noch weitere, jeweils andere Parameter, wie hier `-g`, die Anzahl der Generationen, angegeben werden. Wenn die Parameter `-archive` und `-deviation`, die die Größe des Archivs und die Standardabweichung der Mutation beschreiben, nicht gesetzt sind, werden sie beim `OnePlusOnePG` mit ihren Default-Werten belegt, die fest im Algorithmus implementiert sind und den Einstieg im Umgang mit *NObELTJE* erleichtern. Bei einigen Algorithmen gibt es für manche Parameter jedoch keinen Default-Wert. In diesen Fällen ist die Angabe eines Parameters notwendig. Sollen die praktischen Probleme verwendet werden, so sind ggf. weitere Parameter möglich oder auch notwendig (siehe Kapitel 5.4).
Hat man den Algorithmus mit den zuvor gezeigten Aufruf gestartet und ist die Berechnung beendet, so erhält man folgende Ausgabe-Dateien:
`2005-01-18_02-56-39.best.in und 2005-01-18_02-56-39.best`

`.res`. Die Dateinamen zeigen das Datum und die aktuelle Zeit des Aufrufs. In der ersten befinden sich die Koordinaten der besten Individuen und in der zweiten ihre Fitnesswerte. Mit dem zusätzlichen Parameter `-o` kann der Beginn der Dateinamen selbst gewählt werden:

```
java algo.OnePlusOnePG -t testfkt.Deb -g 100 -o one
```

Nun haben wir in unserem Projektordner zwei Dateien: `one.best.in` und `one.best.res`. Diese Ergebnisse können, z. B. mit einem unserer Plotter oder direkt mit *Gnuplot*, geeignet visualisiert und analysiert werden (siehe hierzu auch Kapitel 5.2 und 5.3).

6.1.2 Vom Design zur Analyse

Soll nicht nur ein Lauf, sondern mehrere Durchläufe eines Algorithmus untersucht und die Parameter der Algorithmen variiert werden, so steht die in Kapitel 5.5.1 vorgestellte Klasse `Design` zur Verfügung. Diese bietet folgende Einstellmöglichkeiten:
Jeder Parameter des Algorithmus kann entweder als Einzelwert, als Liste/Menge oder als Intervall mit Schrittweite/Schrittanzahl angeben werden. Das Design ruft den angegebenen Algorithmus auf der gewählten Testfunktion sequentiell mit allen möglichen Kombinationen der bestimmten Parameter auf:

Liste: `-Parametername s Wert1 Wert2 ...`

Intervall mit Schrittweite: `-Parametername i Intervallgrenze1 Intervallgrenze2 step Schrittweite`

Intervall mit Schrittanzahl: `-Parametername i Intervallgrenze1 Intervallgrenze2 count Anzahl`

Im Gegensatz zu den Aufrufen ohne der `Design`-Klasse, werden die Testfunktionen hier mit `--t` und die Algorithmen mit `--a` angegeben (siehe nächster Befehl). Es können auch verschiedene Durchläufe mit unterschiedlichen Random-Seeds durchgeführt werden. Der Parameter, der hier zu variieren ist, heißt bei allen Algorithmen `-r`. Möchte man Läufe des `OnePlusOnePG` auf der Testfunktion `Schaffer` mit zehn verschiedenen Random-Seeds durchführen, so ergibt sich folgender Befehl:

```
java design.Design --a OnePlusOnePG --t testfkt.Schaffer
-g s 100 -deviation s 1.0 -r i 1 10 step 1
```

Soll auch der Parameter `deviation` verändert werden, so schreibt man:

```
java design.Design --a OnePlusOnePG --t testfkt.Schaffer
-g s 100 -deviation i 1 4 step 1 -r i 1 10 step 1
```

Die `step` Angabe bestimmt, dass `-deviation` mit einer Schrittweite von „eins“ pro Lauf verändert wird, hier werden also pro Random-Seed noch zusätzlich vier wei-

tere Läufe mit verschiedenen `deviation` Einstellungen gestartet. Besonders muss man hier auf die Schreibweise achten. Werden die Angaben mit `i` bzw. `s` vergessen, bekommt man keine Fehlermeldung, sondern das Design wird sofort beendet.
Die Ausgabe, die von `Design` produziert wird, ist ein Verzeichnis, das hier `OnePlusOnePG_testfkt.Schaffer1106014263983` heißt. Der Verzeichnisname besteht aus dem Namen des Algorithmus, der Testfunktion und eines „Zeitstempels“, um ein Überschreiben von früheren Ausgaben zu verhindern. Die Dateien, die in dem Verzeichnis erstellt werden, tragen jeweils die Parameternamen und deren verwendeten Werte im Namen. Für den vorigen Aufruf heißt eine beispielsweise `g=100_deviation=1.0_r=1.0.best.res`.

War die Berechnung erfolgreich und wurde das Verzeichnis mit den Ergebnissdateien gefüllt, kann man mit diesen Daten nun die Ergebnisse analysieren. Hier gibt es das `analysis`-Paket, welches mehrere Plotter enthält. Als Eingabe erwarten alle Analyse-Tools den vollständigen Verzeichnisnamen des jeweiligen Experiments. Im Folgenden wird das Verzeichnis angegeben, welches bei unserem vorigen Design-Aufruf produziert wurde:

```
java analysis.MeanMetricPlotter -m SMetricFleischer
-d OnePlusOnePG_testfkt.Schaffer1106014263983 -p1 deviation
```

Weiterhin muss mit `-m` die Metrik, die die Läufe bewertet, und beim `MeanMetricPlotter` mit `-p1` der Parameter, über den jeweils gemittelt werden soll, angegeben werden. Welche Plotter und Einstellungsmöglichkeiten zur Verfügung stehen, kann im Kapitel 5.3 nachgesehen werden.
Als weiteres Beispiel nehmen wir den `MetricsPlotter`, der folgendermaßen aufgerufen wird:

```
java analysis.MetricsPlotter -m SMetricFleischer
-d OnePlusOnePG_testfkt.Schaffer1106014263983 -sort r
```

Durch den Parameter `-sort` werden die Dateien in dem Verzeichnis nach dem jeweiligen Wert, hier `r`, sortiert. Die Plotter versuchen zur Visualisierung *Gnuplot* automatisch zu starten. Falls dies nicht funktioniert, muss *Gnuplot* mit den erhaltenen *Gnuplot*-Scripten manuell gestartet werden. Die Grafik zeigt dann an, welche Random-Seeds welchen Metrik-Wert erzeugt haben.

6.1.3 Beispielaufrufe des SRing-Modells

Beispiel für einen einfachen Aufruf mit dem Algorithmus OnePlusOnePG

```
java algo.OnePlusOnePG -t simulator.elevatorcon.SRingCon
-g 500 -ttn 4
```

Für den Algorithmus wird die Generationenanzahl 500 und für den Simulator ein vierstöckiges Gebäude gewählt.

Beispiel für ein komplettes Design mit dem Algorithmus BasisGAPG

```
java design.Design --a BasisGAPG
--t simulator.elevatorcon.SRingCon -r i 1000 1010 step 1
-psize i 400 500 step 20 -g s 125 -live s 100 --time
-ttpath s /home/tux/simulator -ttm i 2 10 step 2
-ttp s 0.2 0.5 0.9
```

In diesem Aufruf unterscheidet man drei Arten von Parametern: Die Parameter, die nur vom Design verwendet werden, erhalten ein „`--`“ vorangestellt. Optionen, die das Design an die zu verwendende Testfunktion/den Simulator weitergibt, werden durch das Präfix „`-tt`“ gekennzeichnet. Die übrigen, durch ein „`-`“ benannten, Parameter dienen zur Steuerung des gewählten Algorithmus. Weitere Informationen zur Designplanung und -durchführung, insbesondere zur Parametrisierung, stehen im Kapitel 5.5.1. Die in diesem Beispiel verwendeten Simulator- und Algorithmenspezifischen Parameter werden im Kapitel 5.4.2.4 (S-Ring-Simulator) und der Online-Hilfe (siehe dazu Kapitel 5.5.2.2) näher beschrieben.

In dem Beispielaufruf werden die Populationsgröße (`-psize`) und der Random-Seed (`-r`) des Algorithmus variiert. Die Anzahl der Generationen (`-g`) wird fest gewählt. Als Zusatzoptionen sollen alle 100 Simulatoraufrufe das Zwischenergebnis abgespeichert (`-live`, ermöglicht später eine Analyse des Verlaufs) und die benötigte Zeit des Designaufrufs festgehalten werden (`--time`). Als Parameter für den Simulator werden ein besonderer Standort für den Simulator (`-ttpath`) sowie die Anzahl der Fahrstuhlwagen (`-ttm`) und die verschiedenen zu simulierenden Ankuftswahrscheinlichkeiten (`-ttp`) gewählt. In diesem Fall ist das Fahrstuhlproblem drei-kriteriell, da für die Wahrscheinlichkeiten 0,2, 0,5 und 0,9 optimiert wird.

6.2 Erweiterungsmöglichkeiten

Die gesamte von der Projektgruppe implementierte Software des *NObELTJE*-Projekts ist bewusst modular gehalten; es ist großer Wert auf die Erweiterbarkeit derselben gelegt worden. So ist es möglich, das Paket der Algorithmen, der Metriken oder der Anwendungsfälle (in unserem Fall hauptsächlich Testfunktionen, sowie zwei Simulatoren) beliebig zu erweitern. Ein Anwender kann beliebige Komponenten zum Software-Paket der Projektgruppe hinzufügen, es müssen lediglich Interfaces, bzw. die Methoden der abstrakten Klassen implementiert werden.

Für die Algorithmen muss die `AbstractAlgo.java` im Paket `algo` implementiert werden, für die Metriken ist dies die `Metrics.java` unter `metrics` und für die Testfunktionen ist dies `AbstractTestfkt` im Paket `testfkt`; zudem gibt es für die Testfunktionen noch das Interface `TestFunctions`. Für konkrete Anwendungsfälle, z. B. aus der Praxis wird auch die `TestFunctions` gewählt. Entspricht ein neu implementiertes Modul diesen abstrakten Klassen, so kann es in der Software-Umgebung wie alle anderen benutzt werden.

6.2.1 Beispiel

Wir nehmen an, es soll die neue Testfunktion `Dummy` implementiert werden. Dafür muss zunächst im Verzeichnis `testfkt` die Datei `Dummy.java` angelegt werden. In dieser Datei wird die neue Testfunktion dann implementiert. Es muss per `package testfkt;` dort angegeben werden, dass sie zum Paket `testfkt` gehört, zudem per `extends AbstractTestfkt`, dass die neue Testfunktion diese abstrakte Testfunktion implementiert.

In der Methode `execute` wird nun die eigentliche Funktion implementiert. Eingabewert an die Methode ist eine Datenstruktur von Java-Typ `List`, in der die n Punkte der n-dimensionalen Eingabemenge ($x_1 ... x_n$) als Objekte der Eingabewerte (z. B. Objekte vom Typ `Double` oder `Int`) übergeben werden. Die Methode gibt dann analog eine Liste vom Typ `List` mit m Objekten (wiederum z. B. `Double` oder `Int`, je nach Art der Testfunktion) zurück, die den zugehörigen Funktionswerten der m Kriterien der Testfunktion entsprechen.

Die Methode `writeParetoPoints` wird so implementiert, dass sie die zu der Testfunktionen gehörenden Punkte der „wahren“ Pareto-Menge ausgibt. Dies kann Methoden-intern entweder rechnerisch (falls eine analytischer Ausdruck zum direkten Berechnung dieser Punkte existiert) oder auch über eine externe Datei, in der diese Punkte enthalten sind, geschehen. Übergeben werden die Anzahl der gewünschten Punkte von der Paretofront, sowie ein Dateiname (vom Java-Typ `String`), der angibt, in welche Datei diese Punkte gespeichert werden sollen. Diese Methode wird für die Metriken benötigt, die zum Errechnen des Metrik-Wertes auf eine Referenz-Menge angewiesen sind.

Durch die Methode `getOptimalBounds` kann einem Algorithmus ein Gebiet von „guten“ Punkten im Suchraum übergeben werden, damit sich der Algorithmus für den Start bereits in der Nähe guter Lösungen befindet. Ausgegeben wird wieder eine Liste mit den Grenzen, Index 0 der Liste enthält dabei die untere Grenze für die Variable x_1; Index 1 enthält die obere Grenze für x_1; Index 2 enthält dann die untere Grenze für x_2, und so weiter.

Die Methode `getStartPoint` gibt, in Anlehnung an `getOptimalBounds`, einen einzigen Punkt in der Nähe von guten Lösungen zurück.

Mittels `getBounds` kann die Testfunktion ausgeben, auf welchen Zahlenbereichen die Funktion überhaupt definiert ist; ihre Implementierung und Benutzung verlaufen analog zu `getOptimalBounds`.

Ferner gibt es noch die Methode `setPenalty`, welche eine „Straffunktion“ implementiert. Sie sollte benutzt werden, falls der Definitionsbereich der Testfunktion nicht dem gesamten Wertebereich des benutzten Datentyps entspricht. So kann eine Funktion z. B. nur auf den reellen Zahlen zwischen 0 und 1 definiert sein. Ein evolutionärer Algorithmus z. B. würde nun bei Verlassen dieses Definitionsbereiches nur ungültige Individuen von der `execute`-Methode zurückbekommen, hätte also keinen An-

haltspunkt, in welches Bereich des Suchraumes er steuern sollte, um wieder gültige Individuen zu erhalten. Die Methode `setPenalty` bekommt als Liste die Koordinaten eines solchen Individuums und gibt eine Liste (die so viele Einträge hat, wie die Testfunktion Kriterien) zurück, in der „Strafwerte“ stehen. Diese Strafwerte für das Individuum fallen um so höher aus, je weiter das Individuum von den Definitionsgrenzen der Funktion entfernt ist. Somit erhält der Algorithmus wieder einen Hinweis, in welchem Bereich des Suchraumes er weitersuchen soll.

Die Methoden `getDimension` und `getNumberOfObjectives` geben schließlich an, wie viele Dimensionen n der Eingabevektor ($x_1..x_n$) und wie viele Kriterien die Testfunktion hat.

Letztendlich sollte jede Testfunktion eine `toString`-Methode implementieren, welche über die jeweilige Testfunktion im speziellen Aufschluss gibt und Erklärungen liefert.

6.3 Lizenz

Die Teilnehmer der PG möchten allen Leuten, die daran interessiert sind, *NObELTJE* zur Verfügung stellen. Dazu wurde auf der Homepage des **Instituts für Spanende Fertigung** der Universität Dortmund eine Projektseite (siehe [1]) eingerichtet.

Der Quellcode steht unter der *GNU General Public License Version 2* (siehe [33]) und kann im Rahmen dieser Lizenz genutzt werden. Eine deutsche Übersetzung der Lizenz ist unter [34] zu finden. Sollten Sie *NObELTJE* nutzen oder weiterentwickeln, würden wir uns sehr über Ihr Feedback freuen.

Dieses Paket enthält Code der *KEA Toolbox* (siehe [35]), der für die Anbindung des Simulators für das Temperierbohrungsproblem verwendet wurde. Dieses Projekt ist auch unter der GPL verfügbar, wie auch der von uns eingebundene Code. Die Distribution enthält den Simulator des Temperierbohrungsproblems jedoch nicht. Der *Evolver* wurde uns für unsere Forschungsarbeit vom ISF zur Verfügung gestellt, ist aber nicht öffentlich zugänglich. Das Fahrstuhlproblem wurde mit einem Simulator untersucht, der dem S-Ring Modell aus [29] angelehnt ist, und für *NObELTJE* angepasst wurde. Er kann beispielsweise mit *GCC* kompiliert werden. Weiterhin verwendet *NObELTJE* externen Code zur Gif-Kompression.

7 Ergebnisse aus der eigenen Forschung

Die Mitglieder der PG447 gruppierten sich im zweiten PG-Semester nach Forschungsinteressen und stellten sich selbst herausfordernde Aufgaben. Dieses Kapitel ist den Ergebnissen unserer selbständigen Forschung gewidmet.

Zusätzlich zu den in Kapitel 5.3 beschriebenen erstellten Werkzeugen zur graphischen Darstellung und zur statistischen Auswertung, werden in Kapitel 7.1 weitere bekannte Visualisierungs- und Auswertungsverfahren multidimensionaler Daten aufgeführt, insbesondere die Darstellungsmöglichkeiten des Tools *GGobi*.

Eine Weiterentwicklung des SMS-EMOA wird in Kapitel 7.2 vorgestellt, wobei die Wirksamkeit und Effizienz eines modifizierten Selektionsoperators untersucht und mit der originalen Version verglichen wird.

Eine Arbeitsgruppe analysierte die Sensibilität einzelner Algorithmen gegen veränderte Parametrisierungen, wobei intensiv gegenseitige Abhängigkeiten der Parameter des `MopsoOne` untersucht wurden. Diese Ergebnisse, sowie Vergleiche von optimal parametrisierten Algorithmen bei gleicher Anzahl von Zielfunktionsauswertungen sind Inhalt von Kapitel 7.3.

In Kapitel 7.4 werden zwei praktische Optimierprobleme analysiert, in Unterkapitel 7.4.1 das Temperierbohrungsproblem und in 7.4.2 das Fahrstuhlproblem. Hierbei werden die produzierten Lösungsmengen verschiedener Algorithmen mit Hilfe von Metriken verglichen.

Die Gruppen nutzten *NObELTJE* intensiv für ihre Arbeit und führten einige Erweiterungen des Softwarepakets durch.

7.1 Visualisierung

7.1.1 Einleitung

Bei der Analyse von Ergebnissen macht es oft Sinn, die Daten in einem ersten Schritt grafisch darzustellen, bevor man sie statistisch auswertet. Auf diese Weise kann man sich schnell einen Überblick verschaffen; oft lassen sich so schon Zusammenhänge und Strukturen erkennen.

Wir haben uns nun die Frage gestellt, welche Möglichkeiten überhaupt bestehen, um Daten mit mehr als drei Dimensionen darzustellen und welche dieser Darstellungsweisen besonders intuitiv und gut lesbar sind. Weiterhin haben wir aus den zahlreichen Verfahren dann einige ausgewählt, um sie in der Projektgruppe zu realisieren. Dabei konnten wir teilweise auf bereits implementierte Tools zurückgreifen, wie beispielsweise *GGobi* und *R*.

Leider ist die Darstellung am Monitor auf zwei Dimensionen beschränkt, die dritte Dimension lässt sich noch auf die Ebene projizieren. Gerade bei mehrkriteriellen

Problemen stößt man also schon bald an seine Grenzen. Es existieren zahlreiche Verfahren zu diesem Thema, die sich grob in zwei Gruppen einteilen lassen: Zum einen kann man versuchen, für höhere Dimensionen alternative Darstellungsformen zu finden, oder man geht den Weg der Dimensionsreduktion und versucht die Daten mit möglichst geringem Informationsverlust auf zwei oder drei Dimensionen abzubilden. Beide Vorgehensweisen haben ihre Vor- und Nachteile. Oft findet auch eine Mischung beider Methoden statt.

Im Folgenden werden einige Möglichkeiten aufgeführt, die wir näher untersucht haben. Einen vertiefenden Einblick, auch in Themen, die hier nicht näher erläutert werden, geben Yu und Stockford [36].

7.1.2 Alternative Darstellung höherer Dimensionen

Will man die Ergebnisse nicht herunterrechnen, muss man für die Veranschaulichung weiterer Dimensionen andere Möglichkeiten finden. Der klare Vorteil dieser Methode ist, dass es zu keinem Datenverlust kommt, die Informationen bleiben hundertprozentig erhalten. Allerdings lassen sich solche Grafiken gewöhnlich intuitiv nicht mehr so schnell und gut erfassen. Es erfordert einige Übung diese Visualisierungen zu interpretieren.

Farbe als weitere Dimension

Die vierte Dimension darzustellen, gestaltet sich noch relativ einfach. Eine gebräuchliche Methode ist es, Farbe hinzuzunehmen. Ist die hinzugefügte Dimension geordnet, muss man allerdings beachten, dass die Farbpalette keine natürliche Ordnung besitzt. In diesem Fall bietet sich eine Grauskala an, oder man stellt die Ordnung durch die Saturierung, d. h. die Farbsättigung, dar.

Eine Anwendung von Farbe wäre beispielsweise, jedem Individuum eines Evolutionären Algorithmus in der nullten Generation eine eindeutige Farbe zuzuordnen. Durch Reproduktion entstehen neue Individuen mit der gleichen Farbe, durch Rekombination entstehen Individuen mit Mischfarben.
Oder man färbt bei Mutation die betroffenen Bereiche eines Individuums weiß ein. Auf diese Weise lässt sich gut beobachten, wie viel von dem ursprünglichen Genmaterial erhalten bleibt.

Zeit als weitere Dimension

Die fünfte Dimension lässt sich durch die Zeit darstellen. Man erhält also eine animierte Darstellung der Daten. Hier sieht man schon, dass die Visualisierungen immer schwerer lesbar werden. Allerdings eignet sich diese Methode noch relativ gut dazu, Interaktionen zwischen den einzelnen Variablen zu entdecken. In einem typischen 3D-Plot zeigt die Form der Fläche die Abwesenheit oder Präsenz von Relationen zwischen Variablen an.

Die folgenden Abbildungen erfordern etwas Fantasie, da das Ausgabemedium bedauerlicherweise keine Animationen zulässt.

Ist die Fläche eben, geht man davon aus, dass keine Abhängigkeiten existieren, so wie in Abbildung 28. Hier werden die ersten drei Variablen auf den drei Achsen dargestellt und eine vierte Variable durch Farbe. Eine fünfte Variable kann durch Zeit dargestellt werden. Eine Interaktion mit der fünften Variable ließe sich daran erkennen, dass sich die Neigung der Fläche mit der Zeit verändert.

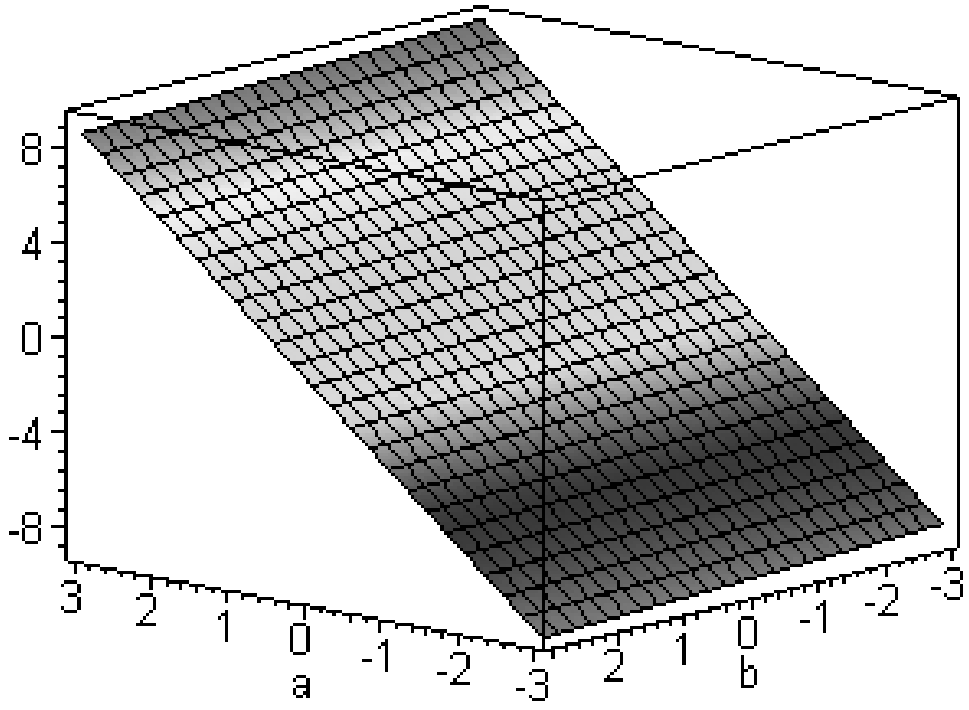

Abbildung 28: Ein 3D-Plot ohne Interaktionen

Ist die Fläche gekrümmt, kann man normalerweise von einer 2-Wege-Interaktion ausgehen. Zeigt die animierte Grafik eine Fläche, die sich in Abhängigkeit von der fünften Variable bewegt, existiert sogar eine 3-Wege-Interaktion. Siehe Abbildung 29.

Verfolgt man diese Vorgehensweise weiter, steht man bei jeder weiteren Dimension immer wieder aufs Neue vor dem Problem, dass man für sie eine weitere Darstellungsform entwickeln muss.

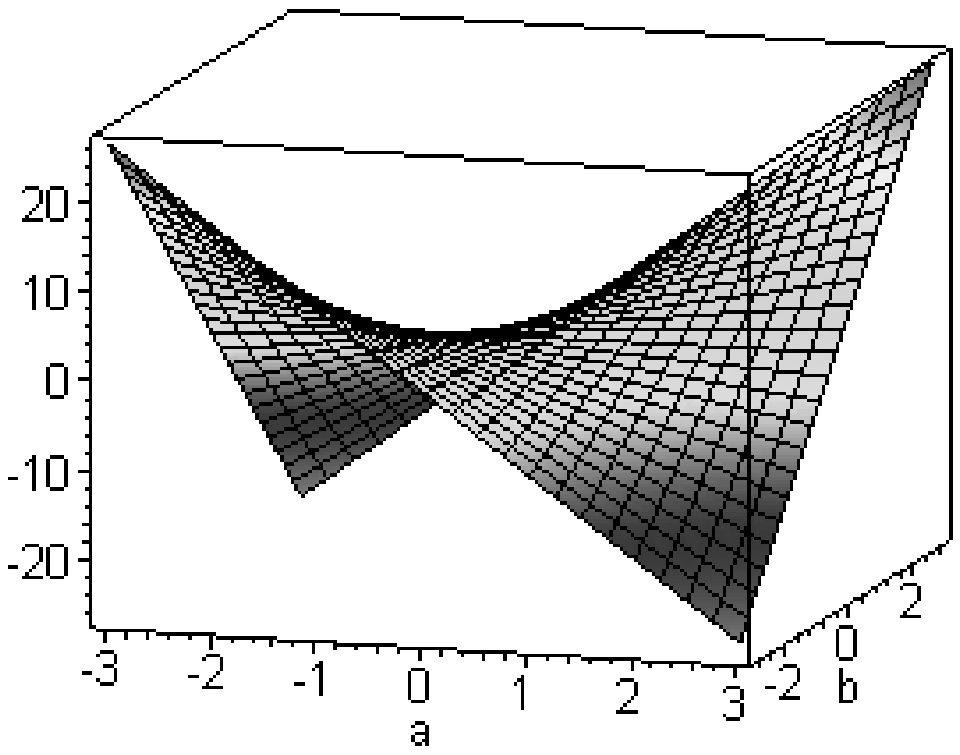

Abbildung 29: Ein 3D-Plot mit 2-Wege-Interaktion

Glyphen

Eine andere gebräuchliche Methode zur Visualisierung multidimensionaler Daten ist die Verwendung von so genannten Glyphen, wie sie auch bei Spears [37] beschrieben werden. Glyphen sind kleine Symbole, die abhängig von den Daten, die sie darstellen, ihr Erscheinungsbild wie z. B. Form, Farbe, Größe, Orientierung etc. ändern. Dabei wird für jeden mehrdimensionalen Punkt eine Glyphe gezeichnet. Auf diese Weise können vorhandene Strukturen in den originalen Daten durch Gemeinsamkeiten oder Unterschiede in den Merkmalen der einzelnen Glyphen dargestellt werden. Zwei gängige Arten sind „Star Plots“ und „Chernoff Faces“.

Star Plots

Star Plots, oder auch Sunflower Plots, sind Punkte, von deren Mitte Linien ausgehen, wie die Speichen eines Rads. Für jede Dimension wird eine Speiche gezeichnet, wobei die Länge den Wert der jeweiligen Variable darstellt. Es gibt zahlreiche Variationen von Star Plots, eine einfache zeigt Abbildung 30.

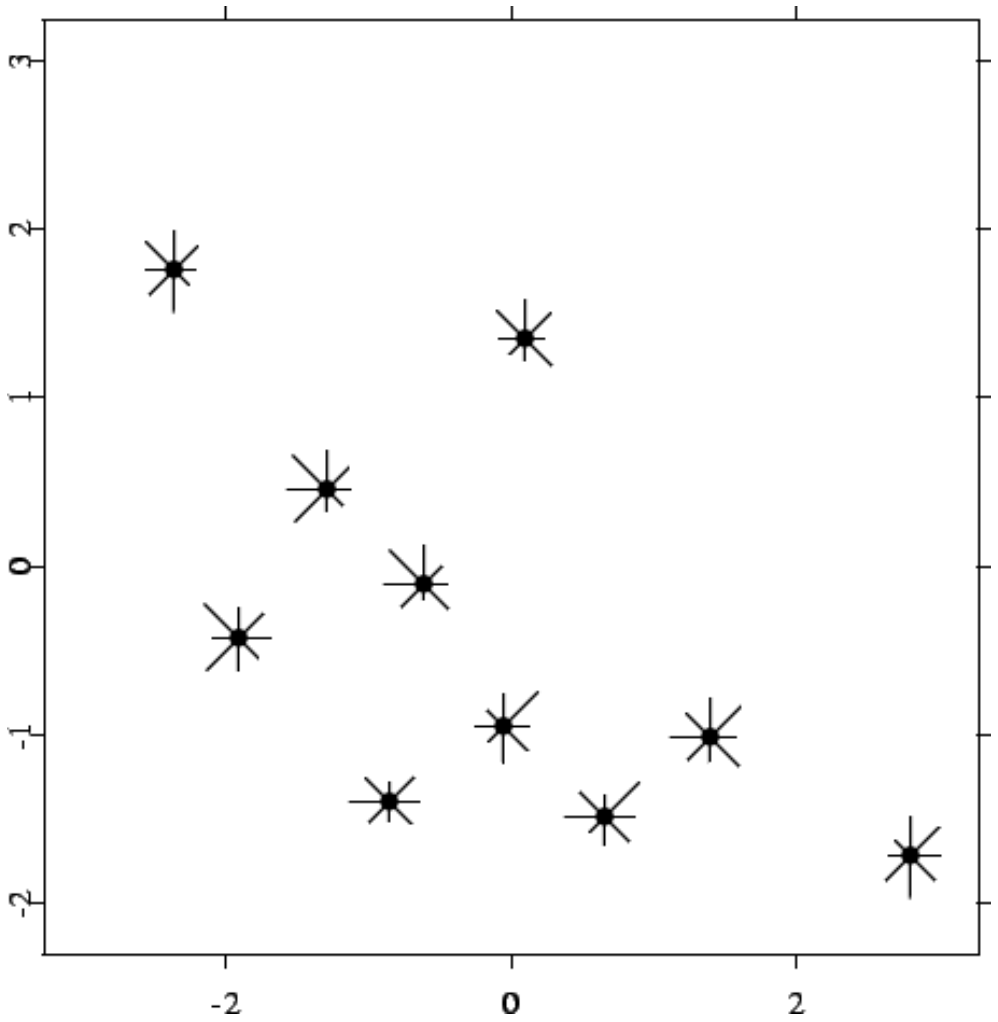

Abbildung 30: Ein Star Plot, welches zehn Dimension darstellt. Zwei Dimensionen durch das verwendete Koordinatensystem und acht durch die Speichen an jedem der Punkte.

Chernoff Faces

Ein etwas komplexeres Beispiel für Glyphen sind die so genannten Chernoff Faces. Jeder Datenpunkt wird durch ein stilisiertes menschliches Gesicht dargestellt. Dabei werden durch die Größe, Form oder den Abstand einzelner Gesichtsmerkmale die Werte der verschiedenen Variablen dargestellt wie beispielsweise in Abbildung 31. Motiviert ist diese etwas ausgefallene Darstellungsform durch das Fakt, dass der Mensch sehr schnell in der Lage ist, zwischen unterschiedlichen Gesichtern zu differenzieren und sie wieder zu erkennen. Weitergeführt wurde dies mit den *Chernoff Bodies*, wo jedes Symbol zusätzlich zum Gesicht auch noch einen Körper mit Armen und Beinen besitzt.

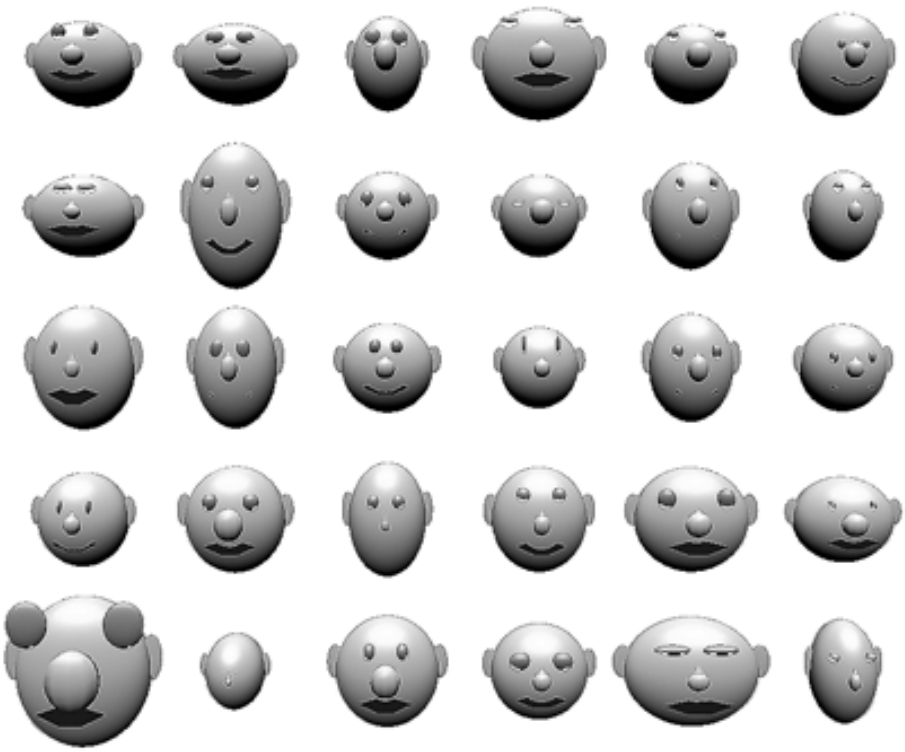

Abbildung 31: Verschiedene Chernoff Faces, die eine Vielzahl an Dimensionen darstellen. Unter anderem durch Abstand, Größe und Form von Mund, Augen und Nase.

Parallelprojektion

Ein ganz anderer Ansatz wird bei der Parallelprojektion verfolgt. Bisher wurden Daten immer in orthogonalen Koordinatensystemen dargestellt. Bei dieser Art der Visualisierung werden die Achsen für alle Variablen parallel unter- bzw. nebeneinander aufgetragen. Der Wert einer Variablen wird auf der zugehörigen Achse als Punkt dargestellt; die Punkte eines Lösungsvektors werden durch Linien miteinander verbunden. Auf diese Weise erhält man für jede Lösung einen Polygonzug im Diagramm. Abbildung 32 zeigt eine solche Parallelprojektion. Bei sehr vielen Lösungen kann diese Darstellung schnell unübersichtlich werden. Hier hilft es, wenn man einzelne Polygonzüge farbig markiert um sie vom Rest abzuheben.
Die Parallelprojektion haben wir in der Projektgruppe eingesetzt, um die Daten zu analysieren. Zum Teil ließ sich sehr schön erkennen, welche Auswirkungen die Änderung eines Parameters auf die von ihm abhängigen Variablen nach sich zog. Dabei kam *GGobi* zum Einsatz.

Andrews Curves

Bei Andrews Plots werden ähnlich der Parallelprojektion einzelne Lösungsvektoren

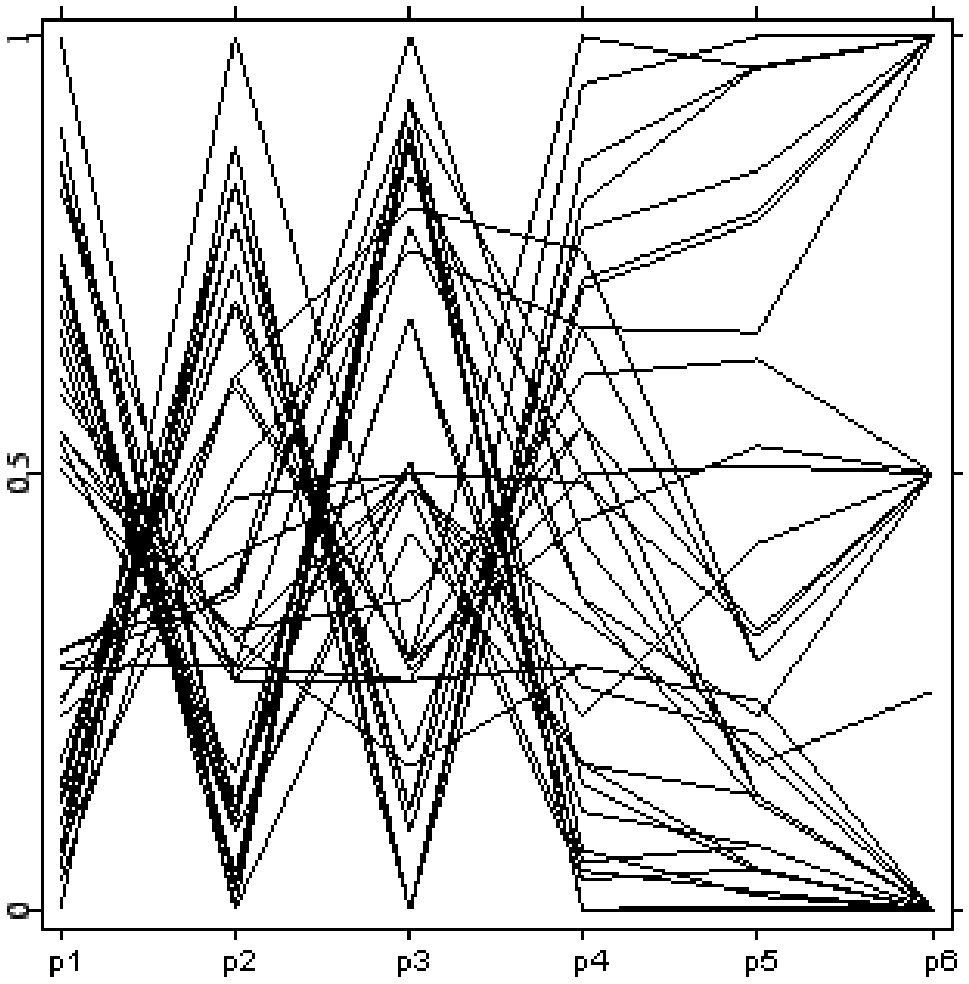

Abbildung 32: Ein durch Parallelprojektion dargestellter Datensatz. Man kann beispielweise erkennen, dass ein hoher Wert von p1 bei einem Individuum gleichzeitig einen niedrigen Wert von p2 bedeutet.

durch eine Linie dargestellt. Allerdings werden hier nicht einzelne Punkte zu einem Polygonzug verbunden. Stattdessen wird jeder Datenpunkt auf eine Kurve abgebildet.

Für den Lösungsvektor $v = (v_1, ..., v_n)$ ist die Kurve durch folgende trigonometrische Funktion definiert:

$$f_v(t) = \frac{u_1}{\sqrt{2}} + u_2 * sin(t) + u_3 * cos(t) + u_4 * sin(2t) + u_5 * cos(2t) + ...$$

für $t \in (-\pi, \pi)$

Die wichtigsten Variablen sollten mit den niederfrequenten Termen der Funktion assoziiert werden, da sie das Gesamterscheinungsbild der Kurve am stärksten bestimmen. Siehe hierzu Abbildung 33.
Diese Projektionstechnik ist nützlich, um Cluster in der Datenstruktur zu finden. Mehr-

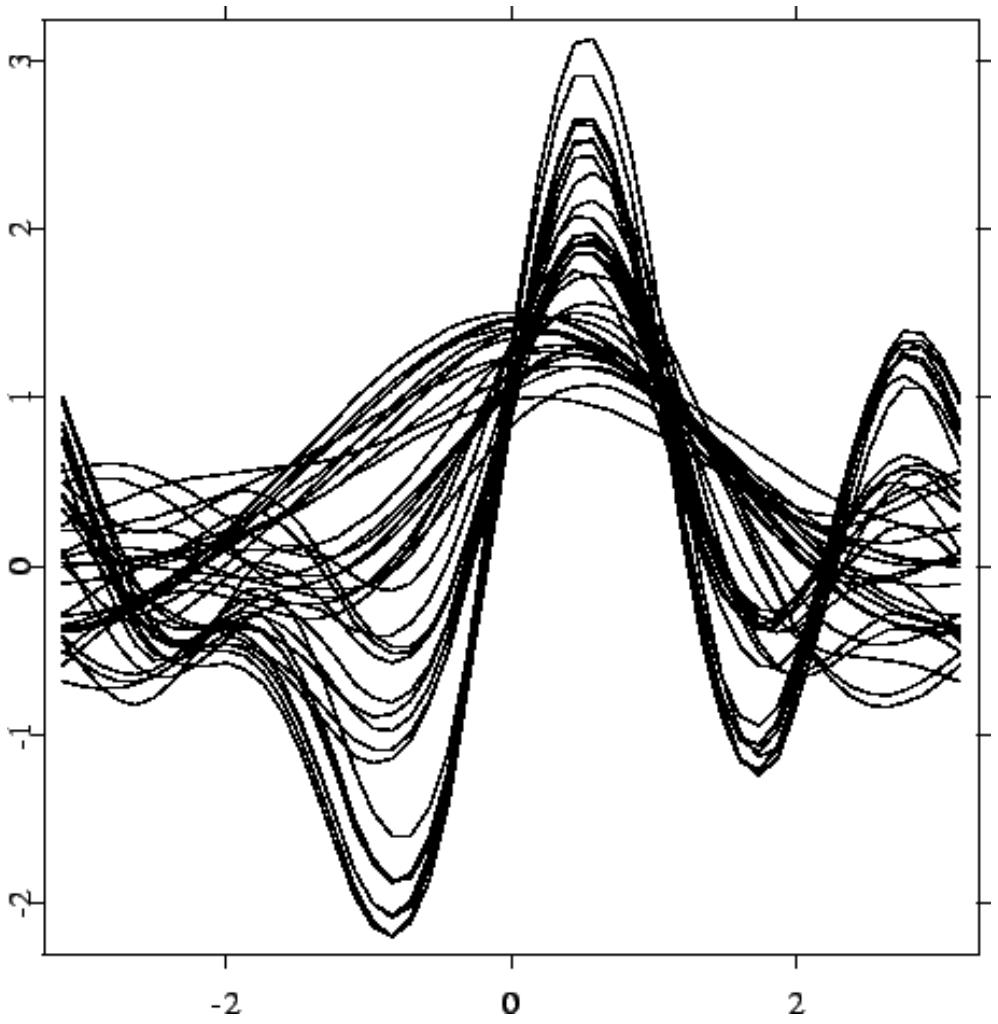

Abbildung 33: Beispiel für Andrews Curves

dimensionale Punkte, die nahe zusammenliegen, haben tendenziell auch Kurven, die eng beieinander verlaufen.

7.1.3 Dimensionsreduktion

Während es sich bei den bisher vorgestellten Methoden zur Visualisierung mehrdimensionaler Daten um Darstellungsverfahren ohne Informationsverlust handelte, werden im Folgenden einige Methoden vorgestellt, bei denen die betrachteten Daten reduziert werden. Entweder wird nur ein Teilausschnitt der Datenmenge berücksichtigt, um ganz gezielt bestimmte Zusammenhänge zu untersuchen, oder die Daten werden auf weniger Dimensionen abgebildet, so dass sie leichter darstellbar sind.

Beim typischen 3D-Plot kommt diese Methodik bereits zum Einsatz. Eine dritte Variable wird so auf zwei Dimensionen projiziert, dass die Abbildung dreidimensional wirkt. Hier ist es schon nicht immer eindeutig, wo sich die dargestellten Punkte genau im Raum befinden, und man muss sich bereits einiger Hilfsmittel bedienen. Eine

Möglichkeit, um etwas mehr Klarheit zu schaffen, ist, die Punkte mit einer senkrechten Linie auf eine der Grundebenen zu verbinden (Abbildung 34). Oder man animiert die Grafik indem man sie um eine Achse rotieren lässt.

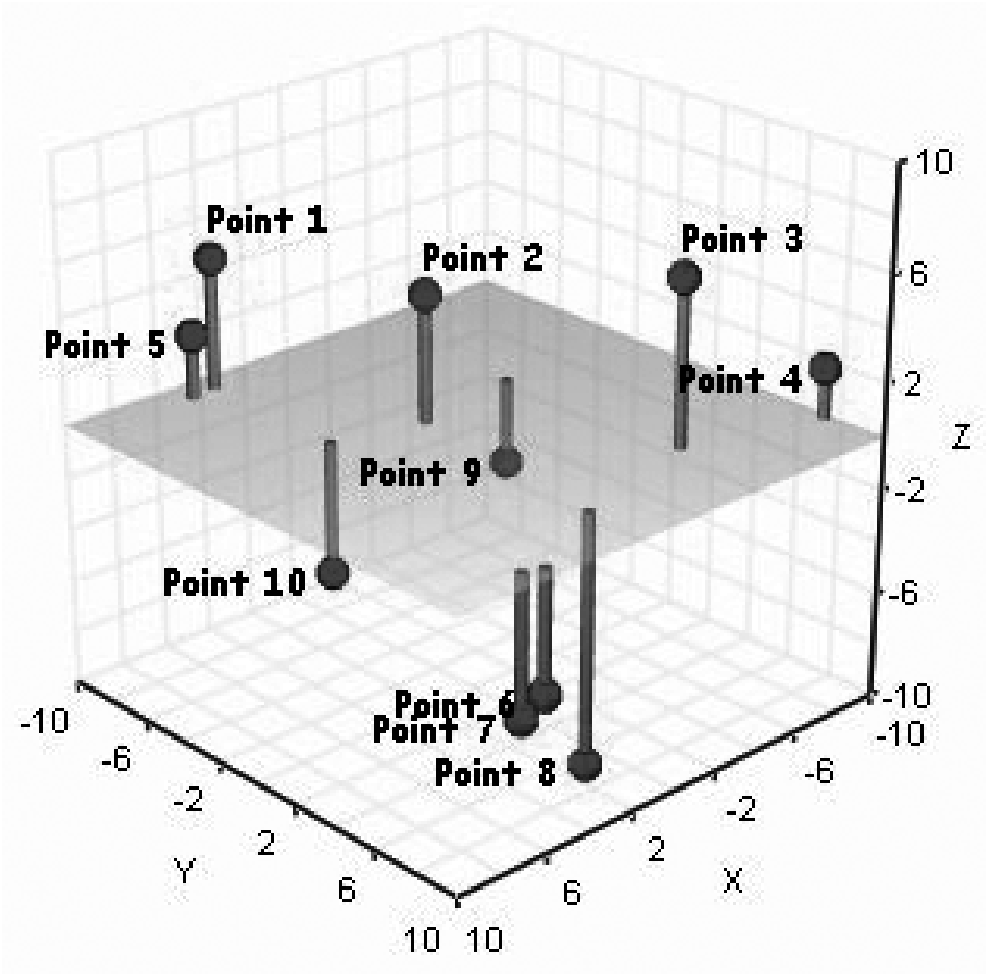

Abbildung 34: Ein einfaches 3D-Plot, bei dem die Punkte zusätzlich mit der Grundebene verbunden sind. Dadurch soll der Informationsverlust durch die Reduktion von drei auf zwei Dimensionen ausgeglichend werden.

Multidimensional Scaling

Beim multidimensionalen Skalieren rechnet man hochdimensionale Daten so auf eine geringere Anzahl von Dimensionen herunter, dass beide Darstellungen immer noch ähnlich zueinander sind. Das bedeutet in anderen Worten, eine solche Transformation sollte ein Bild liefern, in dem die Unterschiede zwischen den einzelnen Datenpunkten im niederdimensionalen Raum mit den Originaldaten aus dem hochdimensionalen Raum korrespondieren.

Um diese Unterschiede zwischen den Datenpunkten zu messen, verwendet man die Distanz. Diese Distanz kann der echte Abstand, z. B. der euklidsche Abstand, zwischen den Originaldaten aus dem hochdimensionalen Raum sein oder auch einen Ersatz für den Abstand darstellen, wenn direkte Werte nicht berechenbar sind. Zwei gebräuchliche Methoden für multidimensionales Skalieren sind das Sammon-Mapping und Genotype-Space-Mapping. Vertiefende Literatur zu diesem Thema findet man bei Pohlheim [38] [39].

Typische Beispiele für die Anwendung dieser Methodik ist die Visualisierung der Variablen der besten Individuen eines Optimierungslaufs oder die Visualisierung von nicht-dominierten Lösungen bei mehrkriteriellen Problemen. Die Diagramme, die sich auf diese Weise ergeben, liefern oft ein klareres Bild, als wenn man die Daten direkt dargestellt hätte. Zusätzlich lassen sich so die Daten mehrerer Optimierungsläufe einfacher vergleichen, was mit Standard-Visualisierungstechniken eine komplizierte Aufgabe darstellt.

Scatterplot

Beim Scatterplot wird lediglich die Beziehung zwischen zwei der Variablen untersucht, alle anderen werden vernachlässigt. Die Daten werden als Punkte in ein Diagramm eingetragen mit der ersten Variable auf der Abszisse und der zweiten auf der Ordinate. Scatterplots können Korrelationen zwischen den Variablen aufdecken, z. B. Linearität. Ergibt sich eine lose Punktewolke in der Grafik, korrelieren die beiden Werte nicht.

Üblicherweise verwendet man Scatterplot-Matrizen, um sich schnell einen ersten Überblick zu verschaffen. Eine Scatterplot-Matrix ist eine Sammlung von einzelnen Scatterplots, die genau wie eine Kovarianz-Matrix aufgebaut ist. Die Variable i wird gegen die Variable j aufgetragen in der i-ten Zeile und j-ten Spalte. Allerdings wird diese Darstellung für viele Variablen schnell unübersichtlich. Abbildung 35 zeigt eine Scatterplot-Matrix mit drei Variablen.

Regressionsanalyse

Geht man den Weg der Scatterplot-Matrizen weiter, führt einen dies zur Regressionsanalyse. Bei einem Scatterplot lassen sich zwar bereits Korrelationen zwischen zwei Variablen erkennen, allerdings werden die Daten wirklich nur rein grafisch dargestellt und nicht weiter interpretiert.

Bei der Regressionsanalyse hingegen versucht man, die Beziehung zwischen zwei Variablen durch eine Funktion zu bestimmen oder wenigstens zu approximieren. Im einfachsten Fall geschieht dies durch eine Gerade, wie in Abbildung 36 angedeutet.

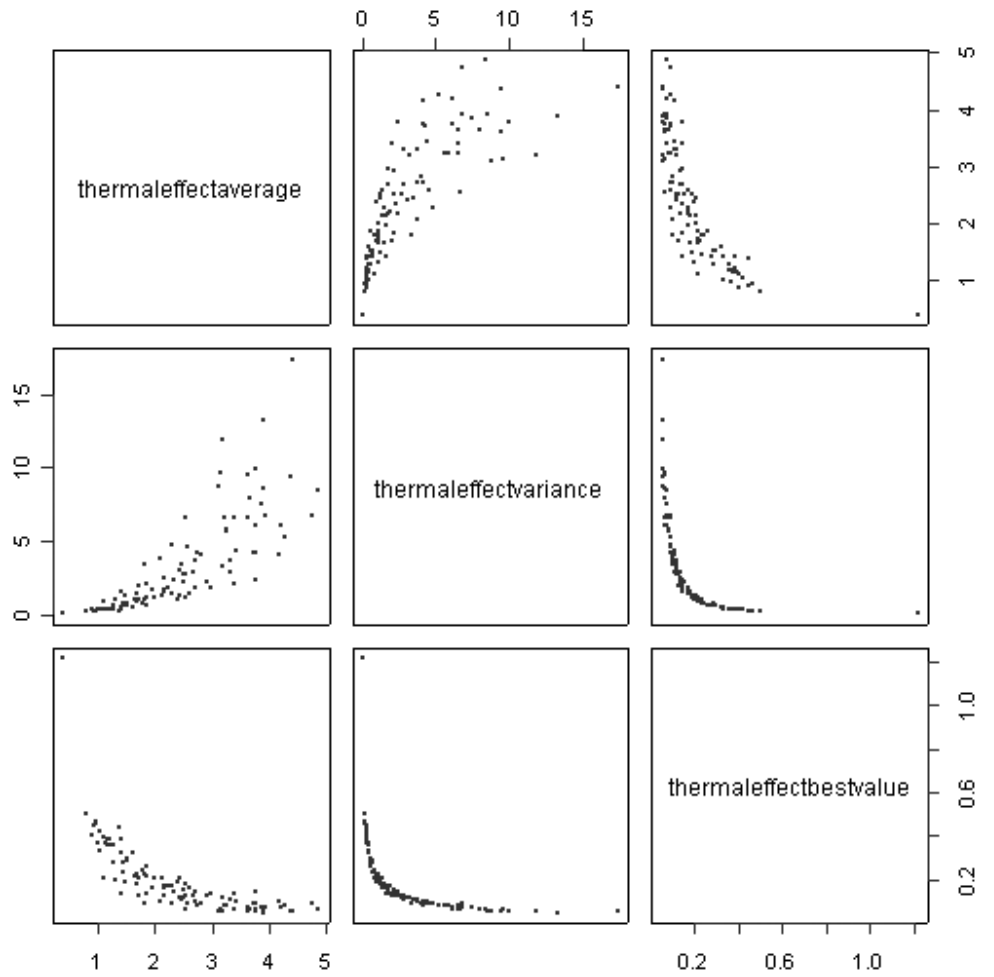

Abbildung 35: Eine Scatterplot-Matrix für 3 Dimensionen. Es werden dabei jeweils 2 Dimensionen in einem Scatterplot in Beziehung gebracht.

Histogramm

Ist man an der Häufigkeit interessiert, mit der Werte einer bestimmten Variablen bei den Individuen vorkommen, bietet sich ein Histogramm an. Bei dieser Darstellungsart wird lediglich eine Variable bei allen Individuen betrachtet. Auf der Abszisse wird der Wert der Variablen abgetragen, auf der Ordinate die Anzahl der Individuen, die mit diesem Wert vorkommen. Es gibt mehrere Arten, auf die ein Histogramm dargestellt werden kann, z. B. als Balkendiagramm (Abbildung 37) oder als kontinuierliche Linie (Abbildung 38).

Boxplot

Das Boxplot stellt ähnliche Informationen wie das Histogramm dar. Allerdings werden diese hier ein wenig anders aufgeschlüsselt. Die Verteilung der Werte für eine

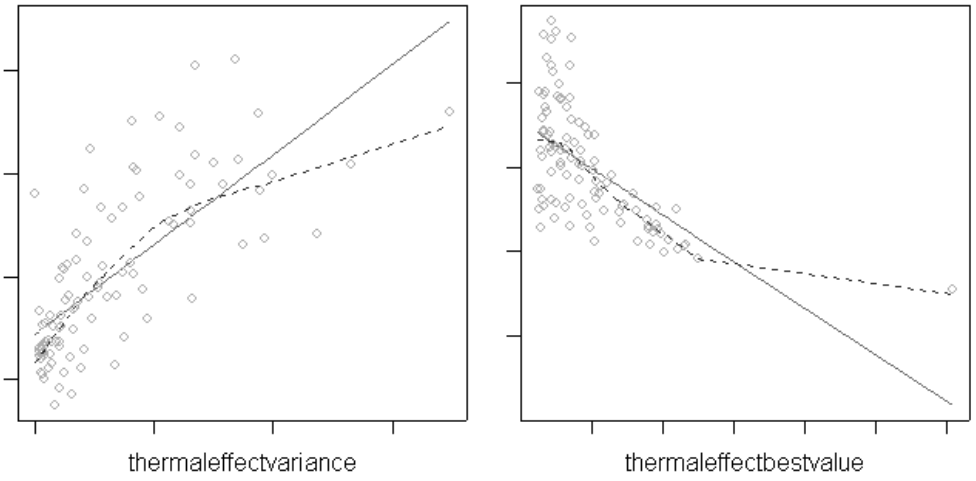

Abbildung 36: Beispiel für Regressionsanalyse. Die Linien stellen den Verlauf einer approximierten Funktion zu den Werten dar.

Variable wird durch fünf Rahmenwerte dargestellt. Vom ersten bis zum dritten Quartil wird eine Box gezeichnet. Erstes Quartil bedeutet hier, dass 25 Prozent aller Werte darunterliegen, beim dritten Quartil entsprechend 75 Prozent. Innerhalb der Box wird der Median durch eine Linie dargestellt. Alle Werte, die außerhalb der Boxen liegen, werden als Extreme bezeichnet und durch vertikale, gestrichelte Linien ober- und unterhalb der Boxen eingezeichnet, die an sogenannten Zäunen enden. Schließlich gibt es noch Ausreißer, die extra durch Punkte angegeben werden. Als Ausreißer gelten alle Punkte, die mehr als das $1,5$-fache der Kastenhöhe von den Quartilen entfernt sind. Zur Veranschaulichung siehe Abbildung 39.

7.1.4 GGobi

GGobi ist ein Tool zur Visualisierung mehrdimensionaler Daten. Viele der hier vorgestellten Visualisierungstechniken gehören zum Funktionsumfang von *GGobi*. Das besondere ist, dass sich verschiedene Darstellungen gleichzeitig anzeigen lassen und man Individuen markieren kann. Diese Markierungen werden dann in allen Darstellungen angezeigt. So lässt sich sehr schnell ein Überblick über eine große Datenmenge gewinnen, bevor mit einer exakten statistischen Auswertung begonnen wird. Wir konnten dadurch beispielsweise sehr schnell den Einfluss von Parametern auf einen Metrikwert bestimmen. Hier soll nun exemplarisch gezeigt werden, wie so ein Vorgehen mit *NObELTJE* und *GGobi* für einen Designaufruf aussehen kann.

Die Daten in diesem Beispiel stammen vom `MopsoOne`-Algorithmus und dem Temperierbohrungsproblem. Den `MopsoOne` haben wir mit den Parametern `g=100`, `g=200` und `g=400` laufen lassen, zudem noch `quantityParticle` von 50 bis 100 in Schrittweite 10, `c1` und `c2` von 1 bis 5 in Schrittweite 1, `scalingInertia=1` und `maxInertia=0.9` gewählt, sowie drei verschiedene Random-Seeds verwendet. Es wurden drei bis fünf Bohrungen simuliert. Die Daten wurden danach gefiltert und auf 3 Dimensionen reduziert. Das genaue Vorgehen bei den Daten des Evolvers wird

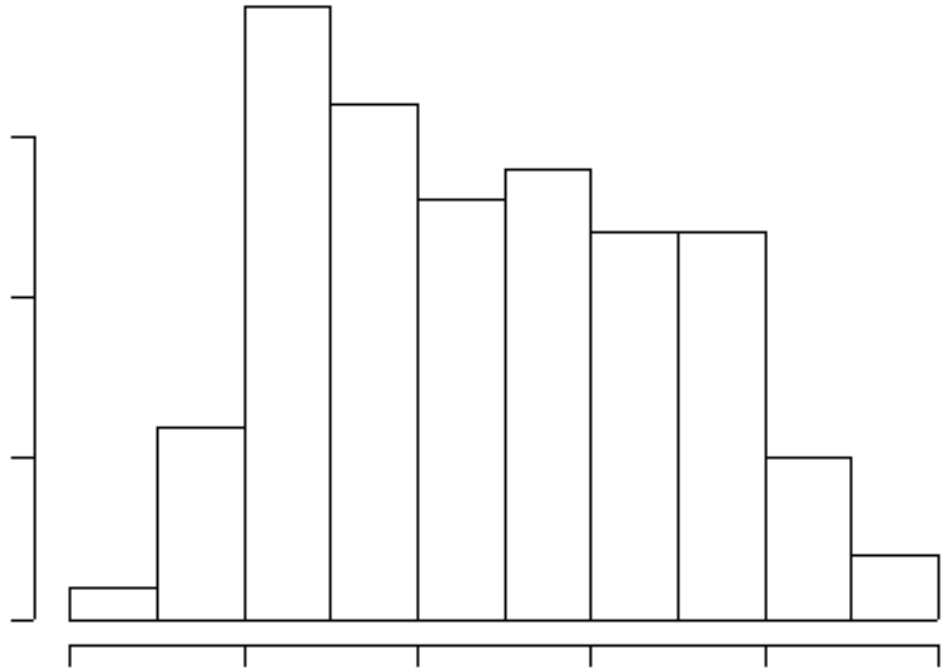

Abbildung 37: Ein Histogramm als Balkendiagramm. Die Höhe der Balken repräsentiert die Anzahl der Individuen, die in dem Wertbereich liegen, der durch die Breite der Balken vorgegeben wird.

in Kapitel 7.4.1.1 beschrieben. Es soll nun die Abhängigkeit des Metrikwertes von einigen Paramtern untersucht werden. Dazu müssen die Metrikwerte ermittelt und mit den Parametern des entsprechenden Aufrufs in einem von *GGobi* verwertbaren Format gespeichert werden. Dies kann unser Tool MetricsPlotter mit der Option `-ggobi` (siehe Kapitel 5.3.3). Die dadurch erzeugten Daten lassen sich direkt mit den richtigen Bezeichnungen in *GGobi* einlesen.

GGobi zeigt erst einmal ein wenig aussagekräftiges Scatterplot. Von den vielen Darstellungsmöglichkeiten sind insbesondere die *Scatterplot Matrix* und das *parallel coordinates display* interessant. Hier wird das *parallel coordinates display* verwendet, aus dem die konstanten Parameter ausgeblendet werden sollten. Zusätzlich wird dann noch ein *Barchart* für die Verteilung der Metrikwerte eingeblendet. Jetzt kann man mit der `Draw`-Funktion die zu untersuchenden Parameter farblich hervorheben. Dies zeigt Abbildung 40 für die drei unterschiedlichen Random-Seeds.

Man kann hier auf einen Blick erkennen, wie stark der Zufall die Metrikwerte beeinflusst und entsprechende Konsequenzen ziehen. Mit wenig Aufwand läßt sich aber auch noch der Einfluss der Parameter `c1` und `c2` auf den Metrikwert darstellen. Abbildung 41 zeigt, dass sich diese Parameter beim Temperierborungsproblem völlig anders verhalten, als in den Beobachtungen unserer Robustheitsgruppe bei den Testfunktionen (siehe Kapitel 7.3.2). Ein weiteres Barchart zeigt den Einfluss der Anzahl der Bohrun-

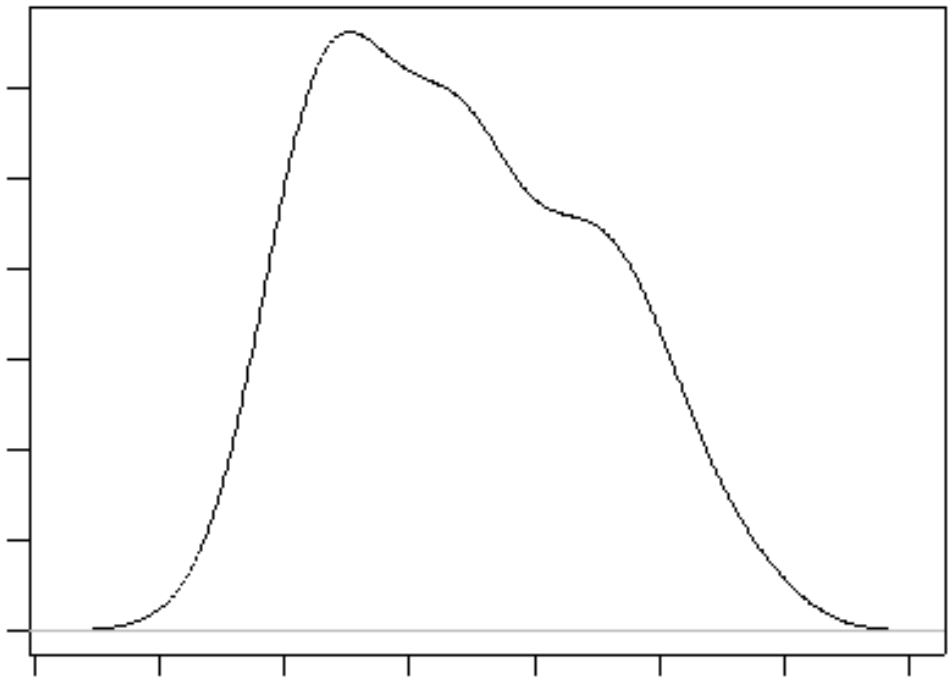

Abbildung 38: Wie 37, nur in einer Darstellung durch eine geglättete Linie.

gen auf den Metrikwert.

GGobi ermöglicht es, mit geringem Aufwand mögliche Zusammenhänge zu erkennen, für die man sonst erheblich mehr Zeit benötigen würde. Somit eignet es sich sehr gut für die erste Analyse eines unbekannten Datensatzes.

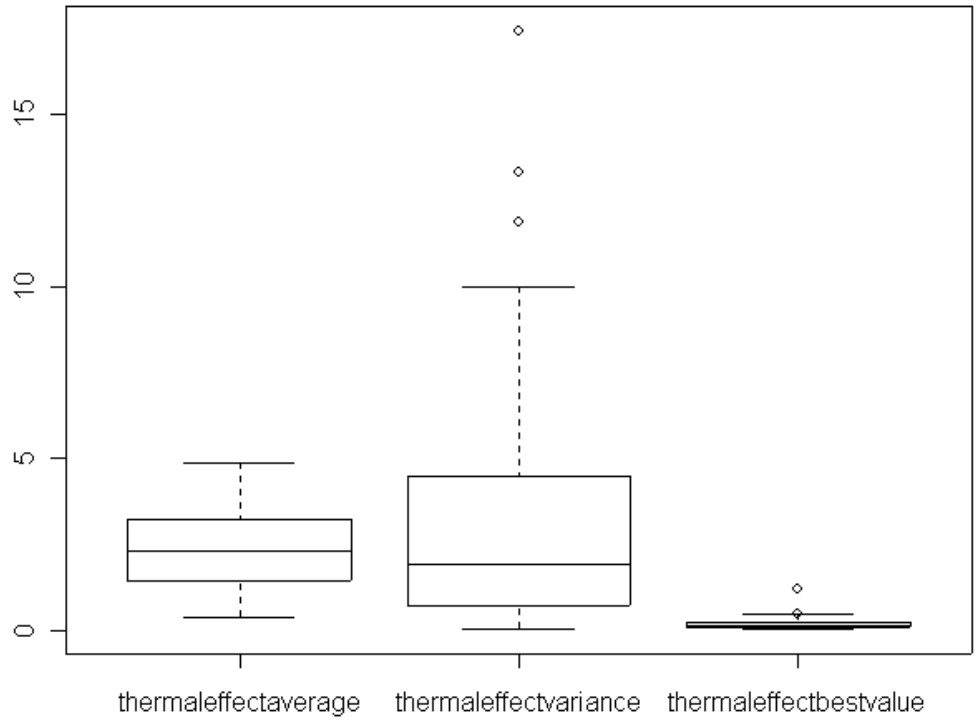

Abbildung 39: Beispiel für ein Boxplot (siehe referenzierenden Text)

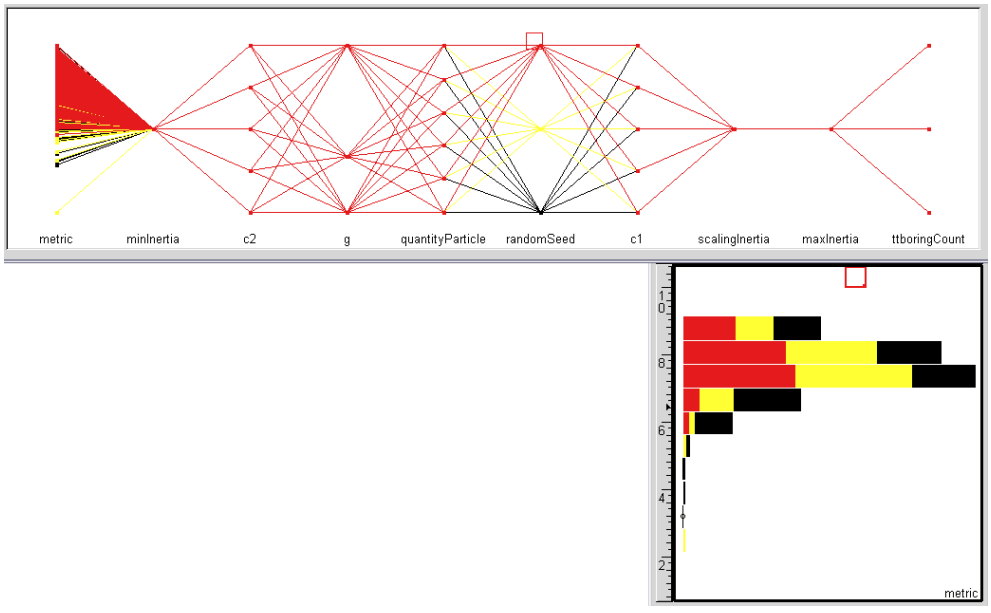

Abbildung 40: *GGobi*-Visualisierung unterschiedlicher Random-Seeds. Die drei verschiedenen Random-Seeds sind farbig bzw. in Graustufen markiert, was auf alle Individuen mit dem entsprechenden Random-Seed übertragen wird. Dies gilt auch in anderen Darstellungsfenstern, wie man in dem Barchart, welches den Einfluss des Zufalls auf die Metrikwerte wiedergibt, sehen kann.

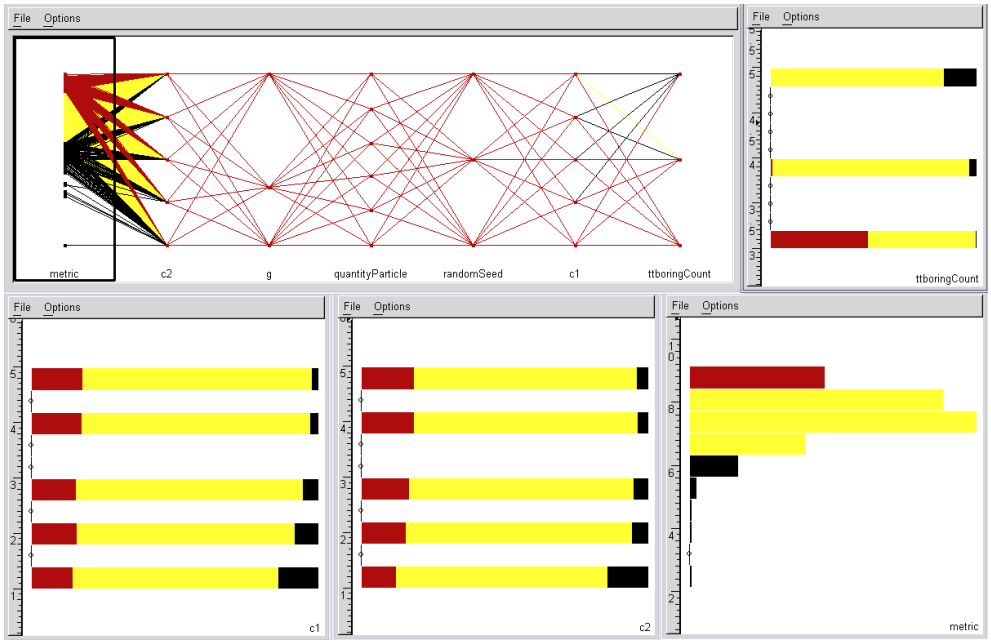

Abbildung 41: *GGobi*-Visualisierung der Abhängigkeit bestimmter Metrikwerte von den Parametern c1 und c2 beim `MopsoOne` sowie der Anzahl der Bohrungen

7.2 Entwicklung und Analyse eines modifizierten SMS-EMOA

7.2.1 Motivation und Idee

Die Arbeit der Projektgruppe baut auf dem SMS-EMOA (S metric selection evolutionary multiobjective optimization algorithm) auf, der von Emmerich, Beume und Naujoks (siehe [16]) 2004 am Lehrstuhl Informatik 11 entwickelt wurde. Der SMS-EMOA ist ein steady-state EMOA, der den Wert der S-Metrik als Selektionskriterium verwendet. Der Selektionsoperator gliedert zunächst die Population in Mengen (Fronten) von Individuen, innerhalb derer Individuen nicht-dominierend sind. Die Mengen sind hierarchisch geordnet, so dass jedes Individuen von dem einer „besseren“ Front dominiert wird (*Non-dominated Sorting*). Anschließend wird das Individuum aussortiert, welches den kleinsten Beitrag zum S-Metrik-Wert der schlechtesten Front liefert. Der Beitrag eines Individuums ist hierbei als die Größe des Hypervolumens definiert, das ausschließlich von diesem Individuum dominiert wird und von keinem anderen. Im Vergleich mit etablierten Algorithmen, wie NSGA-II oder SPEA2 zeigt sich, dass der SMS-EMOA auf der Familie der ZDT-Funktionen diesen Verfahren deutlich überlegen ist. Diese Beurteilung bezieht sich auf Vergleiche mittels S-Metrik und des euklidischen Abstands, wodurch sowohl die Verteilung als auch die Konvergenz der Ergebnismenge bewertet wird.
Eine Arbeitsgruppe der Projektgruppe wählte sich nun die Aufgabe, diesen Algorithmus weiter zu entwickeln. Da die Berechnung des S-Metrik-Wertes im Vergleich zu anderen Metriken sehr zeitaufwändig ist ($O(n^3k^2)$, wobei n die Größe der Paretooptimalen Menge und k die Anzahl der Zielfunktionen bezeichnet), sollte eine andere Metrik gefunden werden, die als Selektionskriterium benutzt werden kann. Die Zielsetzung war, bei geringerer Laufzeit des Algorithmus, die Qualität der Ergebnismenge nicht wesentlich zu verschlechtern.
Es wurde dazu eine neue Metrik entwickelt, die für jedes Individuum die Anzahl der Individuen zählt, die dieses dominieren. Es wird das Individuum aussortiert, welches den größten Metrik-Wert aufweist, also die größte Anzahl dominierender Lösungen. Diese Metrik wird eingesetzt, um das Individuum der schlechtesten Front zu selektieren, falls es mehr als eine nicht-dominierte Front gibt. Bei genau einer Front wird weiterhin die Selektion nach der S-Metrik angewandt. Die Idee der neuen Selektion ist, dass man keine gleichmäßige Verteilung der Lösungen auf den hinteren Fronten anstrebt, sondern Punkte in den Bereichen behält, in denen die vorderen Fronten nicht dicht besetzt sind.

7.2.2 Entwicklung des neuen Algorithmus

Die Arbeitsgruppe versuchte eine geeignete Metrik zu finden, die als Selektionskriterium verwendet werden könnte. Diese sollte dann auch in den SMS-EMOA Algorithmen integriert und getestet werden. Als geeignete Quelle für verschiedene Metriken stellte sich das Buch *Multiobjective Optimization von Siarry und Collette* [19] heraus, das eine große Sammlung verbreiteter Metriken beinhaltet. Die enthaltenen Metriken wurden hinsichtlich der folgenden Kriterien untersucht, die aus Sicht der Projektgruppe eine Metrik erfüllen sollte, um als Selektionskriterium geeignet zu sein.

1. Die Berechnung des Metrik-Wertes sollte effizient sein.

2. Die Berechnung des Metrik-Wertes sollte ohne Referenzmenge möglich sein.

3. Die Metrik sollte kompatibel zu den Outperformance Relationen sein.

Die Tabelle 5 zeigt, die Beurteilung einiger bekannter Metriken nach den obigen Kriterien, nämlich die Coverage-(C-), die Gerneral-Distance-(GD-), die Error-Ratio-(ER-), die Schotts-Spacing-Metrik (SchSp-) und die S-Metrik. Die Zeile *Laufzeit* unterscheidet zwischen niedrig, mittel und hoch. Eine Metrik erfüllt das Kriterium 1, wenn sie eine niedrige Laufzeit zur Berechnung eines Metrik-Wertes benötigt. Die zweite Zeile *Referenzmenge* zeigt, welche der Metriken eine Referenzmenge bzw. einen -Punkt zur Berechnung des Metrik-Wertes benötigt. Die *Kompatibilität* einer Metrik zu den drei Outperformance Relationen soll die Aussagekraft einer Metrik zeigen (siehe [23]). Dabei bedeutet 0, dass die Metrik mit keiner Relation kompatibel ist, 1, dass die Metrik schwach kompatibel mit allen Outperformance Relation ist, 2, dass die Metrik nicht mit der schwachen Relation, aber der starken und kompletten Relation kompatibel ist und 3 bedeutet, dass die Metriken mit allen Relationen kompatibel ist.

Tabelle 5: Auswertung von Metriken an Hand der Kriterien der Projektgruppe

Metriken	**C-**	**GD-**	**ER-**	**SchSp-**	**S-**
Laufzeit	niedrig	mittel	niedrig	mittel	hoch
Referenzmenge	nein	ja	ja	nein	ja
Kompatibilität	2	2	0	0	3

Die Auswertung der Tabelle 5 zeigt, dass die C-Metrik und die Error-Ratio-Metrik als Vorteil gegenüber der S-Metrik eine niedrige Laufzeit haben. Die C-Metrik hat im Gegensatz zur Error-Ratio-Metrik den Vorteil, dass sie keinen Referenzpunkt oder eine -Menge benötigt. Da die C-Metrik jedoch nur schwach kompatibel mit den Outperformancce Relationen ist, kann in dem Fall, dass zwei zu vergleichende Mengen sich nicht komplett dominieren, keine eindeutige Ordnung bestimmt werden. Im Vergleich zur S-Metrik, erwarteten wir also kein gutes Verhalten als Selektionskriterium. Die PG447 entschloss sich, auf Grund der Auswertung der Tabelle keine der betrachteten Metriken als Selektionskriterium zu nutzen.

Da wir keine Metrik gefunden haben, von der wir erwarteten, dass sie annähernd so gut wie die S-Metrik funktionieren würde, haben wir uns selbst eine Metrik ausgedacht, die in einer Abwandlung des SMS-EMOA eingesetzt werden sollte. Der Schwachpunkt der Selektion des SMS-EMOA liegt darin, dass nicht betrachtet wird, wie die Verteilung der Punkte der vorderen Front aussieht. Hier sahen wir eine Verbesserungsmöglichkeit: Unsere Idee beruht auf der Vermutung, dass dünn besetzte Teile der vorderen Fronten durch Elemente der hinteren Fronten aufgefüllt werden könnten. In Gebieten, wo die vorderen Fronten dicht besetzt sind, erscheint es dagegen nicht sinnvoll, Elemente hinterer Fronten zu behalten. Wir entwickelten zwei Lösungsansätze und entschieden uns, den zweiten zu realisieren.

Die erste Idee war, für jeden Punkt zu betrachten, wie viele dominierende Punkte in seinem „Raum“ liegen. Unter dem Raum eines Punktes verstehen wir hierbei die Fläche, die durch zwei Geraden begrenzt wird, die von seinen benachbarten Punkten

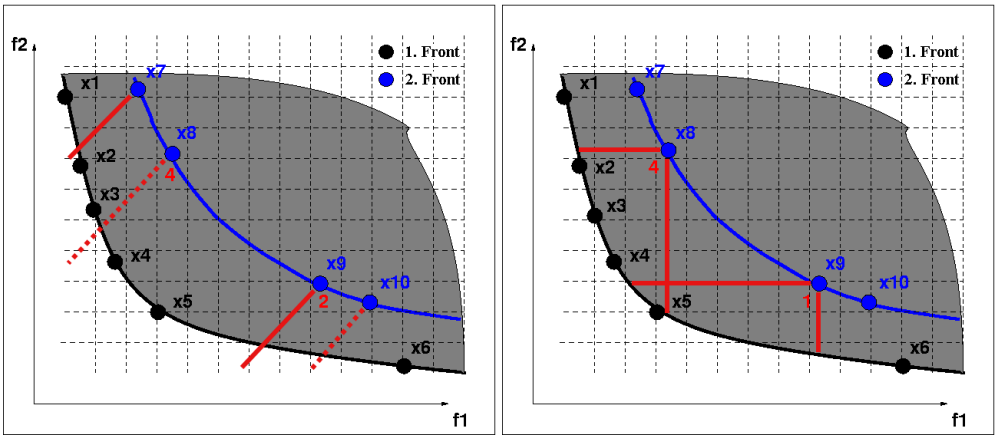

Abbildung 42: Die linke Abbildung zeigt die verworfe Idee, die Anzahl der dominierenden Individuen in einem $45°$-Raum zu berechnen. Die rechte Abbildung veranschaulicht die Funktionsweise der neuen Selektionsmethode: Für jedes Individuum der schlechtesten Front (ausgenommen der Randpunkt) wird die Anzahl dominierender Lösungen gezählt. Das Individuum mit der größten Anzahl wird aussortiert (hier $x8$).

(in derselben Front) im $45°$-Winkel zu den Achsen führen (siehe Abbildung 42). Im Fall von zwei Zielfunktionen, ist dieses Verfahren einfach und effizient. Die Idee wurde jedoch verworfen, da keine einfache Verallgemeinerung auf den Fall von mehr als zwei Zielfunktionen gefunden werden konnte.

Die zweite Idee war es, die Anzahl der Punkte zu zählen, von denen ein Punkt dominiert wird. Der betrachtete Raum entspricht dabei dem Punkt-Ursprung-Hyperkubus. Die Anzahl dominierender Individuen wird als Selektionskriterium verwendet, indem aus der letzten Front das Individuum mit der höchsten Anzahl entfernt wird (siehe Abbildung 42). Randpunkte sollen bei der Selektion möglichst nicht entfernt werden, damit die Diversität erhalten bleibt. Sie werden nur ausselektiert, wenn keine inneren Punkte vorhanden sind.
Mit dieser Methode ergaben sich im Vergleich zu der Idee mit einem $45°$-Raum zwei wichtige Vorteile. Zum einen, ist die Bestimmung der Anzahl der dominanten Individuen im Hyperkubus noch einfacher. Zweitens ist die Methode für eine beliebige Anzahl von Zielfunktionen anwendbar.

7.2.3 Implementierung

Die Implementierung des Algorithmus (siehe Kapitel 5.1.9) und der S-Metrik (siehe Kapitel 5.2.2) wurden bereits detailliert beschrieben. Hier soll daher nur noch einmal ein kurzer Überblick über die verwendeten Verfahren und Interaktionen zwischen diesen gegeben werden. Der modifizierte SMS-EMOA der PG447 entspricht der Java-Klasse `SMS_EMOA_pg`. Der Selektionsoperator des Algorithmus benutzt die Klasse `DeltaSMetric`, um den S-Metrik-Beitrag einer Lösung zu berechnen. Die Klasse `DeltaSMetric` ruft wiederum die Klasse `SMetricFleischer` auf. Diese Klasse

berechnet den S-Metrik-Wert einer Lösungsmenge nach dem Algorithmus von Fleischer [17] und kann optional so verwendet werden, dass nur der Beitrag eines Punktes berechnet wird.
Die Klasse `SMS_EMOA_pg` verwendet einige Klassen bzw. Methoden aus dem Paket `tools`. Die Klasse `Individual` wird benutzt, um die für den Algorithmus benötigen Individuen zu implementieren. Die Methode `refreshArchive` der Klasse `paretoArchive` wird beim Sortieren der Individuen nach Dominanz-Fronten benutzt. Außerdem werden Methoden der Klasse `AlgoOutput` verwendet, um Ergebnisse in Textdateien zu schreiben. Die Durchführung der Experimente wurde uns mit dem Paket `Design` wesentlich erleichtert. Für die Auswertung und Visualisierung der Resultate haben wir den `MeanMetricsPlotter` benutzt, sowie den `AnimatedGifEncoder` um Gif-Animationen zu erstellen.

Wir möchten kurz unsere persönlichen Erfahrungen und Probleme schildern, die während der Implementierung des modifizierten SMS-EMOA auftraten. Die C++-Implementierung des SMS-EMOA wurde uns von den Autoren [16] zur Verfügung gestellt und musste dann in einen Java-Code übersetzt werden. Die Implementierung der Variationsoperatoren basiert auf dem C-Code von Deb, der auf der KanGAL-Homepage [21] verfügbar ist. Verständnisprobleme enstanden dadurch, dass sich diese Implementierungen des Mutations- und Rekombinationsoperators von den theoretischen Beschreibungen in dem Buch von Deb [18] unterscheiden.
Wie in anderen Absätzen erläutert wird, führten einige Experimente auf den ZDT-Funktionen zu außerordentlich schlechten Ergebnissen. Da diese Phänomene des extremen Diversitätsverlustes (siehe 7.2.4.3) vorher unbekannt waren, haben wir den Fehler bei uns vermutet. Wir haben uns von unseren eigenen Experimenten verunsichern lassen und lange an der Korrektheit und der Qualität unseres Algorithmus gezweifelt, was uns sehr viel Zeit gekostet hat. Schließlich erkannten wir, dass diese Phänomene Eigenheiten der ZDT-Funktionen sind und auch bei anderen etablierten Algorithmen auftreten.

7.2.4 Experimente und Auswertung

7.2.4.1 Qualität nach 20.000 Generationen

Um die Qualität des modifizierten SMS-EMOA mit der des Originals zu vergleichen, wurden die Experimente aus [16] nachgestellt. Es wurde der Algorithmus `SMS_EMOA_pg` (Selektionsparameter `sm=0`) mit einer Populationsgröße von $\mu = 100$ mit 20.000 Generationen durchgeführt und die Werte aus fünf Läufen (random seed von 1 bis 5) gemittelt. Die Paramter sind dadurch genauso gewählt wie in [16]. Diese Ergebnisse sind in Tabelle 6 dargestellt und zur besseren Übersicht wurden zusätzlich einige bekannte Werte aus [16] in Tabelle 6 kopiert. Wissenswert für die Interpretation der Euklid-Metrik-Werte ist, dass hierbei der Abstand bezüglich einer Referenzmenge von 1000 Pareto-optimalen Punkten berechnet wird. Daher hat nicht einmal ein Punkt der wahren Pareto-Front einen Euklid-Metrik-Wert von genau 0, es sei denn, er entspricht genau einem der 1000 Referenzpunkte. Die Werte sind also nur bis zu einer bestimmten Genauigkeit vergleichbar. Die Tabelle zeigt, dass die Ergebnisse der beiden SMS-EMOA auf den Funktionen ZDT1, ZDT2 und ZDT3 sehr ähnlich sind. Die S-

Tabelle 6: Werte von S-Metrik und Euklid-Metrik verschiedener Algorithmen bei 20.000 Auswertungen, gemittelt über fünf Läufe. Die Abkürzung Nr. bezeichnet die Rangfolge der Algorithmen.

Funktion	Algorithmus	Euklid-Metrik			S-Metrik		
		Mittel	Std. Ab.	Nr.	Mittel	Std. Ab.	Nr.
ZDT1	NSGA-II	0.00055	6.6e-05	4	0.8701	3.9e-04	5
	SPEA2	0.00101	12.1e-05	5	0.8708	1.9e-04	3
	ϵ-MOEA	**0.00040**	1.2e-05	1	0.8702	8.3e-05	4
	SMS-EMOA	0.00044	2.9e-05	2	**0.8721**	2.3e-05	1
	SMS_EMOA_pg	0.00048	3.4e-05	3	**0.8721**	9.5e-06	1
ZDT2	NSGA-II	**0.00038**	1.9e-05	1	0.5372	3.0e-04	5
	SPEA2	0.00083	11.4e-05	5	0.5374	2.6e-04	4
	ϵ-MOEA	0.00046	2.5e-05	4	0.5383	6.4e-05	3
	SMS-EMOA	0.00041	2.3e-05	2	**0.5388**	3.6e-05	1
	SMS_EMOA_pg	0.00043	2.9e-05	3	**0.5388**	1.5e-05	1
ZDT3	NSGA-II	0.00232	14.0e-05	4	1.3285	1.7e-04	4
	SPEA2	0.00261	15.5e-05	5	1.3276	2.5e-04	5
	ϵ-MOEA	0.00175	7.5e-05	3	1.3287	1.3e-04	3
	SMS-EMOA	0.00057	5.8e-05	2	**1.3295**	2.1e-05	1
	SMS_EMOA_pg	**0.00054**	5.8e-05	1	**1.3295**	1.7e-05	1
ZDT4	NSGA-II	0.00644	0.0043	5	0.8613	0.00640	3
	SPEA2	0.00769	0.0043	4	0.8609	0.00536	4
	ϵ-MOEA	0.00259	0.0006	3	0.8509	0.01537	5
	SMS-EMOA	0.00252	0.0014	2	**0.8677**	0.00258	1
	SMS_EMOA_pg	**0.00231**	0.0010	1	0.8674	0.00261	2
ZDT6	NSGA-II	0.07896	0.0067	5	0.3959	0.00894	5
	SPEA2	0.00574	0.0009	2	0.4968	0.00117	2
	ϵ-MOEA	0.06793	0.0118	4	0.4112	0.01573	4
	SMS-EMOA	0.05043	0.0217	3	0.4354	0.02957	3
	SMS_EMOA_pg	**0.00081**	0.0001	1	**0.5036**	0.00017	1

Metrik-Werte sind genau gleich und die Unterschiede in den Werten der Euklid-Metrik geringfügig. Auch bei ZDT4 zeigen sich fast gleiche Ergebnisse, wobei sich bei dieser Funktion die eigentliche Qualität eines Algorithmus beim Aspekt der Robustheit zeigt. Bei zwei Experimenten (Random-Seed 1 und 2) auf ZDT4 traten extreme Diversitätsverluste auf. Die Tabelle 6 soll die Vergleiche von erfolgreichen Läufen zeigen, daher wurden zwei weitere Experimente (Random-Seed 6 und 7) durchgeführt und die Tabellenwerte für die Läufe mit Random-Seed 3 bis 7 berechnet. Die Ergebnisse des modifizierten SMS-EMOA sind erstaunlich gut für ZDT6. Der Wert der S-Metrik ist fast optimal und der Werte der Euklid-Metrik ist um Größenordnungen besser als bei allen anderen Algorithmen. Dieses Ergebnis kann allerdings nicht dem neuen Selektionsverfahren zugeschrieben werden, denn auch die Nachimplementierung des originalen SMS-EMOA zeigt auf ZDT6 dieses außergewöhnlich gute Verhalten. Für diese Diskrepanz zwischen der originalen C-Implementierung und der Java-Version der PG447 hat die Projektgruppe bisher leider keine Erklärung gefunden.

Der modifizierte SMS-EMOA und das Original haben die gleiche worst case Laufzeit. Falls immer nur genau eine Front existiert, wird unsere schnellere Selektionsvariante nach dominierenden Punkten (`selectDomPoints`) niemals angewendet. Diese worst case Instanz kann zwar nicht ausgeschlossen werden, ist allerdings sehr unwahrscheinlich und daher ist hier die durchschnittliche Laufzeit ein sinnvolles Maß. Es zeigt sich in den Experimenten auf den ZDT-Funktionen, dass unsere `selectDomPoints`-Methode durchschnittlich in etwa $65 - 95\%$ aller Generationen zum Einsatz kommt (siehe Tabelle 7). Die S-Metrik-Selektion (`selectDeltaS`) wird demnach in den

Tabelle 7: Anzahl Anwendungen der beiden Selektions-Methoden bei 20.000 Auswertungen, gemittelt über fünf Läufe.

Selektion	ZDT1	ZDT2	ZDT3	ZDT4	ZDT6
selectDomPoints (abs.)	12998,6	14403,4	14262,6	19036,6	15239,4
selectDomPoints (in %)	64,99	72,02	71,31	95,18	76,20
selectDeltaS (abs.)	7001,4	5596,6	5737,4	963,4	4760,6
selectDeltaS (in %)	35,01	27,98	28,69	4,82	23,80
einfache Selektion (abs.)	8948,8	8944,8	10523,2	13353,4	8421,2
einfache Selektion (in %)	44,74	44,72	52,62	66,77	42,11

übrigen nur $5 - 35\%$ der Generationen angewendet. In den beiden letzten Zeilen von Tabelle 7 ist aufgeführt, wie oft der Fall auftritt, dass sich in der schlechtesten Front nur ein einziges Individuum befindet („einfache Selektion“). In dieser Situation wird dieses Individuum selektiert, ohne dass aufwändige Berechnungen notwendig wären. Der beschriebene Fall tritt in $42 - 67\%$ von 20.000 Generationen auf, was bedeutet, dass hierdurch die Laufzeit beider Selektionsvarianten verbessert wird. Die Schlussfolgerung, dass die praktische Laufzeit des modifizierten SMS-EMOA viel besser ist als der worst case, scheint zulässig zu sein.

Zur Veranschaulichung, wann welche Selektionsmethode zum Einsatz kommt, betrachten wir die Anzahl der Mengen (Fronten) von Individuen, innerhalb derer sich Individuen gegenseitig nicht dominieren. Die Anzahl der Fronten ist für die Funktionen ZDT2 und ZDT4 im Verlauf der 20.000 Generationen in Abbildung 43 dargestellt. Der Anzahl der Fronten für ZDT1 und ZDT3 sinkt ähnlich gleichmäßig wie bei ZDT2, allerdings noch schneller (hier nicht dargestellt). Bei ZDT4 schwankt die Anzahl besonders stark und erreicht ihr Maximum etwa bei der 7.000. Generation. Auch der Verlauf von ZDT6 ist nicht monoton. Die Anzahl schwangt bis zur 2.000. Generation und fällt danach gleichmäßig ab.
Die S-Metrik-Selektion wird erstmalig eingesetzt, wenn die Individuen alle auf genau einer Front liegen. Dies ist bei den Funktionen ZDT1, ZDT2 und ZDT3 zwischen der 4.000 und 7.000 Generationen der Fall. Für ZDT4 liegt dieser Zeitpunkt erst bei etwa 15.000 Generationen und für ZDT6 bei ca. 9.000. Auf Grund des Steady-State-Ansatzes kann sich die Anzahl von Fronten in jeder Generation nur um höchstens eins ändern. Kurze Zeit nach dem erstmaligen Einsatz der S-Metrik ist zu beobachten, dass die Anzahl der Fronten nur noch zwischen eins und zwei schwankt und sich daher die S-Metrik-Selektion und die Selektion nach Anzahl dominierender Punkte häufig

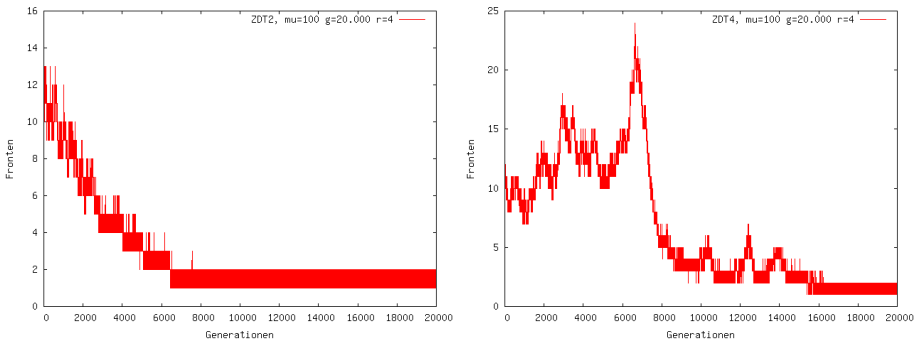

Abbildung 43: Anzahl nicht-dominierter Fronten des SMS_EMOA_pg auf den Funktionene ZDT2 (links) und ZDT4 (rechts) im Verlauf von 20.000 Generationen. Dargestellt ist jeweils der Lauf, der dem Medianwert bezüglich des S-Metrik-Wertes der fünf Läufe entspricht.

abwechseln. Viele dieser Generationen können als einfache Selektionen bezeichnet werden, da sich oftmals nur ein einziges Individuum in der schlechtesten Front befindet.

7.2.4.2 Verlauf von 20.000 Generationen

Nachdem wir die Qualität des modifizierten SMS-EMOA mit der des Originals verglichen haben, untersuchten wir den Verlauf beider Algorithmen.
Der Algorithmus wurde dazu jeweils mit den zwei Selektionsvarianten sm$=0$ und sm$=2$, auf die Testfunktionen ZDT1 (Kapitel 5.4.1.6), ZDT2 (Kapitel 5.4.1.7), ZDT3 (Kapitel 5.4.1.8), ZDT4 (Kapitel 5.4.1.9) und ZDT6 (Kapitel 5.4.1.10) angewendet. Um einen Überblick über den Verlauf dieser Algorithmen zu erhalten, führten wir ein Design mit 1.000 bis 20.000 Generationen durch. Die Läufe wurden mit fünf Random-Seeds (1, 2, 3, 4, 5) durchgeführt, um die Ergebnisse mitteln zu können. Das Experiment wurde durch den Design-Aufruf: `--a SMS_EMOA_pg --t testfkt.ZDTX -g i 1000 20000 step 1000 -mu s 100 -sm s 0 2 -r i 1 5` durchgeführt. Die Ergebnisse wurden mit der S-Metrik nach dem Algorithmus von Fleischer 5.2.2.2 und der EuklidMetrik 5.2.1 ausgewertet. Zur Auswertung mit der S-Metrik wurde der MeanMetrik-Plotter (5.3.4) zur Hilfe genommen, der für die Generationen die Ergebnisse gemittelt hat und jeweils einen S-Metrik-Wert, abhängig von dem Referenzpunkt $(1,1;1,1)$, berechnete.
Der Verlauf der SMS-EMOA-Algorithmen auf der ZDT1-Funktion zeigte, dass der Metrik-Wert des SMS-EMOA-pg nur minimal gegenüber dem Wert des originalen SMS-EMOA „schlechter“ ist, ab der 9000. Generation sind diese identisch. Keines der beiden Selektionskriterien kann den Algorithmus auf der ZDT1-Funktion in den untersuchten Generationen entscheidend beeinflussen, so dass ein Kriterium bevorzugt benutzt werden könnte. Zu gleichem Ergebnis sind wir bereits bei der Betrachtung der Metrik-Werte des Algorithmus nach der 20.000 Generation gekommen (siehe Tabelle 6).
Die Abbildung 45 zeigt einen Sprung der Metrik-Werte in den Lösungen des Algorith-

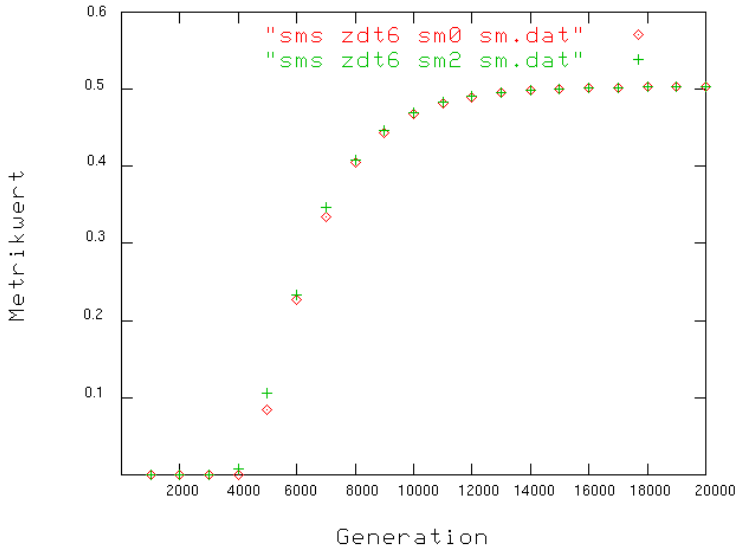

Abbildung 44: Der Verlauf der SMS-EMOA von der 1000. bis 20000. Generation auf der ZDT6-Funktion, gemittelt über fünf Läufe, bewertet mit der S-Metrik.

mus. Dies war ein Hinweis auf die von uns untersuchten Phänomene (siehe Abschnitt 7.2.4.3). Nach Betrachtung der Ergebnisse der 9000. Generation erkannten wir, dass der Lauf für das Selektionskriterium 2 nur Lösungen produzierte, dessen erste Variable gleich 0 war. Dieses Phänomen ist nicht spezifisch, es tritt auch bei anderen Selektionskriterien auf, d. h. bei dieser Auswertung trat das Phänomen bei dem Selektionskriterium 2 auf, was bei einer anderen Auswertung genauso bei den Lösungen des Selektionskriteriums 0 der Fall sein kann. Ähnliches ist auch bei der Auswertung der Ergebnisse für die Läufe auf der ZDT4 aufgetreten. Deshalb entsteht ein so großer Unterschied der Metrik-Werte. Hier wäre es interessant zu untersuchen, bei welchen Parameter-Einstellungen für den SMS-EMOA dieses Phänomen nicht auftritt und mit diesen Einstellungen diese Experimente für die Untersuchung des Verlaufes zu wiederholen. Auf Grund des Phänomens kann man hier nicht mit den Beobachtungen in dem oberen Abschnitt vergleichen.
Die Betrachtung des Verlaufes des Algorithmus auf der ZDT3-Funktion zeigte, dass bereits ab der 2000. Generation der SMS-EMOA mit dem Selektionskriterium 0 den mit dem Selektionskriterium 2 dominiert. Diese Erkenntnis ist unterschiedlich zu der die oben in der Tabelle 6 erlangt wurde, dort unterscheiden sich die Metrik-Werte nicht. Zu dieser Erkenntnis sind wir bereits bei der Betrachtung des Algorithmus nach der 20.000.-ten Generation gekommen(siehe Tabelle 6). Hier sind weitere Experimente nötig, um die unterschiedlichen Metrik-Werte zu vergleichen und das Verhalten des Algorithmus an der 7.000.-ten Generation zu erklären. Nach der 7.000.-ten Generation ist der Metrik-Wert des SMS-EMOA mit dem Selektionskriterium 2 viel schlechter als der des Algorithmus mit dem Selektionskriteriums 0. In den nächsten 3.000 Generationen „holt“ dieser wieder auf, die Metrik-Werte passen sich also wieder an.
Die Metrik-Werte der Algorithmen auf der ZDT4-Funktion zeigten, dass der SMS-EMOA mit dem Selektionskriterium 2 schneller einen höheren und bis zur 20000.-ten

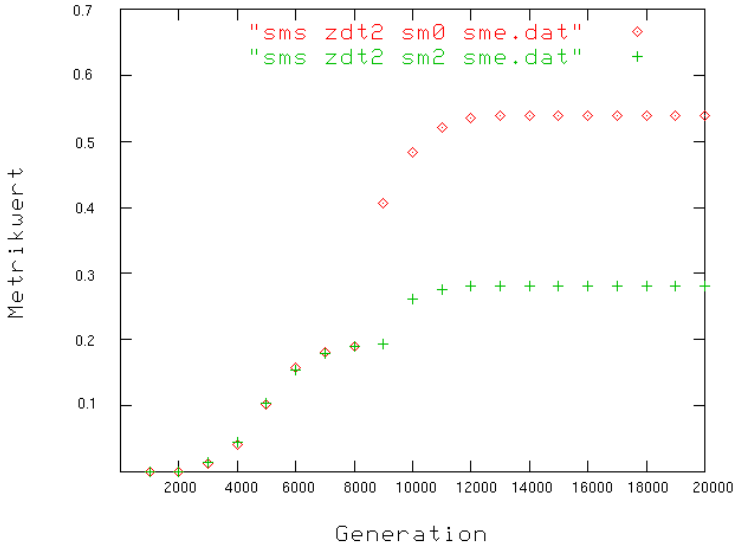

Abbildung 45: Der Verlauf der SMS-EMOA von der 1000. bis 20000. Generation auf der ZDT2-Funktion, gemittelt über fünf Läufe, bewertet mit der S-Metrik.

Generation „bessere“ Metrik-Wert erhält, als der SMS-EMOA mit dem Selektionskriterium 0. Die Darstellung der Verläufe scheint wieder von Phänomen beeinflusst zu sein, da dieser sehr unregelmäßig verläuft. Man beachte auch die Unterschiede der Metrik-Werte zu den Metrik-Werten in der Tabelle 6. Wir vermuten, dass sich die Auswertung des Algorithmus mit Einstellungen, die Phänomene dieser Art nicht hervorrufen, gleiche Ergebnisse liefert wie die Betrachtung der 20.000.-ten Generation. Die Frage wäre, ob sich die oben erkannte schnellere Erreichung eines hohen Metrik-Wertes für den Algorithmus mit Selektionskriterium 2 ergibt und somit den mit Selektionskriterium 0die Ergebnisse des Algorithmus ab der 8000.-ten Generation in der Nähe der pareto-optimalen Punkte der ZDT-Funktionen befinden. Da nicht nur pareto-optimale Punkte aus der Lösung einer Generation zur Berechnung der Euklid-Metrik-Werte benutzt wurden, kann eine allgemeine Aussage nicht getroffen werden. Trotzdem ist es auffällig, dass sich der Algorithmus mit den beiden Selektionskriterien fast gleich „schnell“ an die pareto-optimalen Punkte der ZDT-Funktionen annähert. Diese Vermutung sollte durch weitere Experimente untersucht werden.

Die Betrachtung des Verlaufes des SMS-EMOA mit den Selektionskriterien 0 und 2 lässt darauf schließen, dass keine wesentlichen Unterschiede in der Qualität der Ergebnisse existieren. Um genaue Unterschiede benennen zu können, müssen weitere Ergebnisse, die frei von Phänomenen und nicht-pareto-optimalen Punkten sind, durch Experimente gewonnen werden.

7.2.4.3 Robustheit gegenüber Diversitätsverlust

Schon in der Implementierungsphase des SMS_EMOA_pg ist uns folgendes Phänomen aufgefallen. Auf ZDT2 und ZDT4 versagt der Algorithmus bei bestimmten Kombi-

nationen von Populationsgröße (Parameter μ) und Random-Seed (Parameter `r`). Die Population verliert ihre Diversität. Alle Individuen setzten die erste Variable auf 0, im Extremfall führt das dazu, dass alle Fronten nur aus einem Element bestehen. Dies zeigt beispielsweise die Abbildung 46. Ein Lauf mit 100 Individuen und `r` = 3 ist erfolgreich - die Paretofront wird gut bedeckt. Nach einer sehr geringen Veränderung

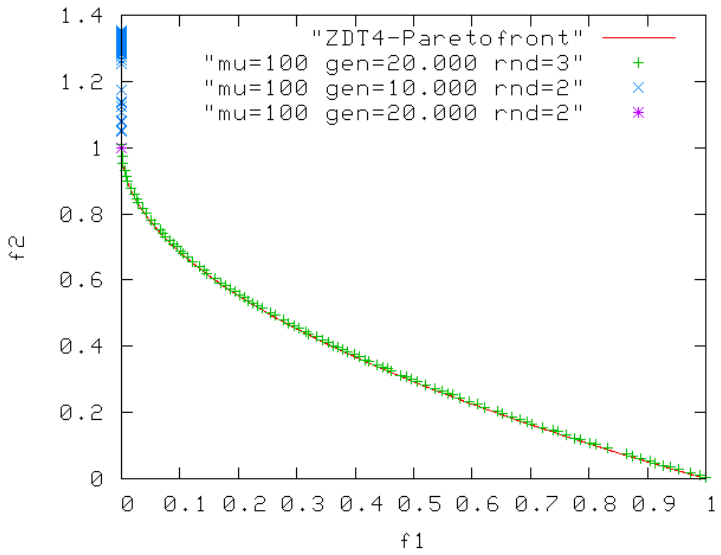

Abbildung 46: Nach einem Lauf mit `r`=3 wird die Paretofront gut bedeckt. Mit dem `r`=2 versagt der Algorithmus völlig. Bis zum Zeitpunkt von 10.000 Generationen setzten alle Individuen die erste Variable auf 0. Nach 20.000 Generationen bestehen alle Fronten nur aus einem Element.

(Random-Seed auf 2 gesetzt), versagt der Algorithmus völlig. Zum Zeitpunkt von 10.000 Generationen sammeln sich alle Individuen an der f_2-Achse. Die erste Variable (und dadurch der Wert von f_1) ist bei allen Individuen auf 0 gesetzt. Die Werte der f_2 unterscheiden sich nur geringfügig. Im Zielraum befinden sich Individuen dicht untereinander auf der f_2-Achse. In der Abbildung46 bilden sie einen unterbrochenen „Strich“ im f_2-Wertebereich von ca. 1.01 bis 1.38. Nach 20.000 Generationen befinden sich alle Individuen auf dem selben Punkt (0,1). Ähnliches kann man auf der Funktion `ZDT2` beobachten.

Wir vermuteten den Fehler im Code vom `SMS_EMOA_pg`. Nach der erfolglosen Fehlersuche stellte sich heraus, dass es an den `ZDT`-Funktionen liegt. Das Phänomen tritt bei anderen Implementierungen von verschiedenen Algorithmen auch auf, z. B. auch beim `MueRhoLES`. Dieses Phänomen wurde bei mehreren Algorithmen (NSGA-II, SPEA2, MOPSO), die innerhalb der Vorgänger Projektgruppe `KEA` implementiert wurden, ebenfalls beobachtet. Außerdem konnte dieses Phänomen mit der PISA-Algorithmenumgebung der ETH Zürich [20] nachvollzogen werden.
Die Ursache liegt vielleicht u. a. daran, dass `ZDT2` (wie `ZDT4`) aus zwei Funktionen besteht, wobei die erste Funktion f_1 gleich der ersten Variable ist, also $f_1 = x_1$. Die zweite Funktion hat dagegen 30 Variablen (im Fall `ZDT4` sind es zehn) und ist wesent-

lich komplizierter. Das könnte eine Erklärung sein, wieso f_1 so schnell minimiert wird.

Wir haben dieses Phänomen etwas näher untersucht. Zuerst haben wir überprüft, ob es auch bei `ZDT1` und `ZDT6` auftritt. Dazu haben wir folgende Design-Aufrufe gestartet:
```
--a SMS_EMOA_pg --t testfkt.ZDT1 -g s 10000 -mu i 20 110
-sm s 0 -r i 1 5
```
und
```
--a SMS_EMOA_pg --t testfkt.ZDT6 -g s 10000 -mu i 20 110
-sm s 0 -r i 1 5
```
Da die Phänomene schon nach wenigen Tausend Generationen auftreten, haben wir die Anzahl der Generationen auf 10000 festgesetzt. Wir haben die Anzahl der Individuen $(20-110)$ und Random-Seed $(1-5)$ variiert. Bei der Selektionsmethode handelt es sich um die von uns entwickelte Selektionsmethode mit den Eigenschaften, dass die Extrema beibehalten werden und das ein Individuum bei Unentscheidbarkeit zufällig selektiert wird (`sm=`0). Auf diesen zwei Testfunktionen sind früher keine derartige Phänomene aufgetreten und in diesem Experiment auch nicht.

Dann haben wir versucht zu überprüfen, ob die randomisierte Version der Selektionsmethode gegenüber der deterministischen Version der Selektionsmethode Verbesserung gegen Diversitätsverlust (Strich-Phänomen) bringen. Dazu haben wir folgende Design-Aufrufe gestartet:
```
--a SMS_EMOA_pg --t testfkt. ZDT2 -g 10000 -mu i 20 110
-sm s 5 3 -r i 1 5
--a SMS_EMOA_pg --t testfkt. ZDT3 -g 10000 -mu i 20 110
-sm s 5 3 -r i 1 5
```
und
```
--a SMS_EMOA_pg --t testfkt. ZDT4 -g 10000 -mu i 20 110
-sm s 5 3 -r i 1 5
```

Wie in dem Experiment davor, haben wir die Generationen auf 10000 festgesetzt. Wieder haben wir die Anzahl der Individuen $(20-110)$ bzw. Random-Seed $(1-5)$ variiert. Zusätzlich haben wir die Selektionsmethoden variiert (`sm=`5 ist die randomisierte und `sm=`3 die deterministische Version unserer Selektionsmethode). Zur Erinnerung: Falls bei der randomisierten Selektionsmethode mehrere Individuen bezüglich unserer Kriterien (gleiche Anzahl dominierender Punkte bzw. gleichen Beitrag zum `S-Metrik`-Wert), den gleichen Wert haben, dann wird aus diesen eines zufällig gewählt. Bei der deterministischen Selektionsmethode wird einfach das letzte Individuum gewählt. Wir hofften, dass die randomisierte Version bessere Ergebnisse erzielen würde.
Wie man aus der Tabelle 8 schließen kann, steigt bei `ZDT2` mit der randomisierten Version der Selektionsmethode die Anzahl der Fehlläufe. Bei `ZDT4` ist es genau umgekehrt. Außerdem treten die Phänomene anders verteilt auf, ohne erkennbares Muster. Allgemein ist hier also keine Aussage möglich.
Obwohl man uns von dem Phänomen auch auf der `ZDT3` berichtet hat, traten bei uns keine vergleichbaren auf. Hier tritt aber folgendes auf: bei bestimmten Einstellungen werden nicht sofort (also in den ersten 10000 Generationen) alle fünf Teilfronten gefunden. Lässt man den Algorithmus z. B. weitere 10000 Generationen laufen, werden in der Regel die fehlenden Fronten allerdings auch noch gefunden. Es gibt aber be-

Tabelle 8: Anzahl der Läufe mit dem Punkt- oder Strich-Phänomen für die deterministische und randomisierte Variante der Selektionsmethode auf den Funktionen ZDT2 und ZDT4.

Test-funktion	sm=3 deterministisch ohne Extrempunkte	sm=5 randomisiert ohne Extrempunkte
ZDT2	172 von 455 (37, 80%)	180 von 455 (39, 56%)
ZDT4	216 von 455 (47, 47%)	195 von 455 (42, 86%)

stimmte Einstellungen (z. B. `sm`$= 0$, `r`$= 1$ und `mu`$= 46$), bei denen auch nach 48000 Generationen nicht alle Teilfronten gefunden werden. Es werden lediglich vier der fünf Fronten gefunden. Aus Zeitmangel konnten wir dieses Phänomen leider nicht näher untersuchen.
Wir haben außerdem überprüft, ob das Behalten der Extrempunkte bei den Selektionsmethoden eine Verbesserung des Strich-Phänomens bringt.
(Design - Aufrufe: `--a SMS_EMOA_pg --t testfkt.ZDT4` bzw. `ZDT2` bzw. `ZDT3 -g 10000 -mu i 20 110 -sm s 4`(extremDeltaS+det) `-r i 1 5`).
Tabelle 9 zeigt alle drei Selektionsmethoden im Vergleich. Scheinbar bringt keine der Maßnahmen einen klaren Vorteil. Die Anzahl der Fehlläufe verändert sich nur minimal und das Phänomen tritt wieder ohne erkennbares Muster auf. Der Erfolg eines Laufs auf den Funktionen ZDT2 und ZDT4 hängt scheinbar nur vom Zufall ab.

Tabelle 9: Anzahl der Läufe mit dem Punkt- oder Strich-Phänomen für die verschiedenen Selektionsmethoden (deterministisch/randomisiert bzw. mit/ohne Extrempunkte behalten) auf den Funktionen ZDT2 und ZDT4.

Test-funk-tion	sm=3 deterministisch ohne Extrempunkte	sm=4 deterministisch Extrempunkte behalten	sm=5 randomisiert ohne Extrempunkte
ZDT2	172 von 455 (37, 80%)	166 von 455 (36, 48%)	180 von 455 (39, 56%)
ZDT4	216 von 455 (47, 47%)	217 von 455 (47, 69%)	195 von 455 (42, 86%)

7.2.5 Fazit

Die Projektgruppe hat mit der Entwicklung der `selectDomPoints`-Methode ihr Ziel erreicht, eine Metrik zu finden, die als Selektionskriterium innerhalb eines EMOA fungieren kann und mit $O(n^2k)$ eine geringere Berechnungskomplexität als die S-Metrik hat.
Darüber hinaus blieb die Qualität des SMS-EMOA sogar vollständig erhalten. Die in den vorherigen Abschnitten präsentierten Experimente zeigen, dass die Qualität der Ergebnisse des modifizierten SMS-EMOA von der des Originals sowohl nach oben als auch nach unten minimal abweicht. Die vergleichenden Analysen der SMS-EMOA Varianten zeigen keine wesentlichen Unterschiede im Verlauf der Algorithmen.
Der Projektgruppe ist es damit gelungen, den SMS-EMOA im praktischen Laufzeitverhalten zu verbessern. Die Analysen der Anzahl der Elemente in der letzten Front und der Häufigkeit der verwendeten Selektionsvarianten zeigen sehr viele einfache

Selektionsfälle, wodurch nahe gelegt wird, dass die Laufzeit normalerweise weit vom theoretischen worst case entfernt ist.
Mit den beiden ergänzten gängigen Verfahren „Randomisierung bei Unentscheidbarkeit“ und „Schützen der Extrempunkte“ konnten leider keine Verbesserungen der Robustheit der SMS-EMOA Varianten auf den ZDT-Funktionen festgestellt werden.
Die Arbeitsgruppe hat mit dem extremen Diversitätsverlust auf den ZDT-Funktionen ein interessantes Phänomen entdeckt, das bisher nicht bekannt war. Scheinbar unterliegen alle Algorithmen diesem Phänomen des zufälligen Versagens, was die ZDT-Funktionen als Test- und Vergleichs-Objekte in Frage stellt.

7.3 Robustheit

7.3.1 Einleitung

Die Robustheitsgruppe hat es sich zur Aufgabe gemacht, die Robustheit der in der Projektgruppe verwendeten Algorithmen zu untersuchen. Der Begriff „Robustheit“ hat verschiedene Bedeutungen. Wir einigten uns auf zwei Betrachtungsweisen, nämlich die Robustheit in Bezug auf Parameter-Einstellungen und das Verhalten verschiedener Algorithmen bei festgegebener Anzahl von Funktionsauswertungen.

Die meisten Optimier-Heuristiken besitzen exogene Parameter, welche vom Anwender spezifiziert werden müssen. Diese Parameter beeinflussen Algorithmen-interne Vorgänge und Verhaltensweisen, wie z. B. die Anzahl der Individuen (bei populationsbasierten Algorithmen), die Anzahl der Generationen, bzw. Iterationen, Mutationsraten (evolutions-basierte Algorithmen) und vieles mehr. Parameter-Einstellungen, die für alle Optimierungs-Probleme zu besten Ergebnissen führen, gibt es im Allgemeinen nicht. Da optimale Problem-spezifische Einstellungen wünschenswert, aber oft nicht bekannt sind, stellt sich die Frage, wie gut ein Algorithmus mit Standard-Einstellungen funktioniert und welchen Einfluss geringfügige Änderungen haben.

Eine weitere Frage ist, wie gut ein Algorithmus läuft, d. h. wie gut er optimiert, wenn eine feste Anzahl von Funktionsauswertungen vorgegeben wird. Insbesondere in der Praxis ergibt sich oft das Problem, dass die Berechnung der einzelnen Funktionswerte einer zu optimierenden Funktion sehr aufwändig ist. Da je nach Ressourcen-Aufwand ein anderer Algorithmus der beste sein könnte, ist der Anwender daran interessiert, für die Zeitbeschränkung bei seinem Optimierungs-Problem den jeweils besten Algorithmus zu finden.

In Kapitel 7.3.2 wird untersucht, wie sich der Particle-Swarm-Algorithmus `MopsoOne` bei Veränderung seiner exogenen Parameter verhält. Unter 7.3.3 wird im speziellen beim `MopsoOne` das Verhältnis zwischen der Partikel-Anzahl und der Anzahl der Generationen betrachtet. In Kapitel 7.3.4 werden verschiedene Algorithmen miteinander verglichen, unter der Vorgabe, dass allen die selbe Anzahl von Funktionsauswertungen vorgegeben wird. Die Ergebnisse der Algorithmen werden bei allen Robustheits-Untersuchungen anhand von Metriken beurteilt.

7.3.2 Parameter-Robustheit

7.3.2.1 Motivation und Fragestellung

Für den Particle-Swarm-Algorithmus `MopsoOne` sollte bei mehreren Testfunktionen untersucht werden, welche Parameter von entscheidender Bedeutung für das Verhalten des Algorithmus sind, wie diese Parameter für das jeweilige Problem zu wählen sind, und wie sehr sich diese Einstellungen bei verschiedenen Optimierungs-Problemen unterscheiden. Dabei wurden zunächst einzelne Parameter variiert, um ein Gefühl für das Algorithmen-Verhalten zu bekommen, später wurden dann statistische Analysen durchgeführt, um sichere Ergebnisse zu erhalten. Für diese Untersuchungen wurden die `Deb`-Testfunktion, die `DoubleParabola`, sowie die `Schaffer`-Testfunktion

(siehe Kapitel 5.4.1) benutzt.

Die Gruppe führte zunächst eine Literatur-Recherche bezüglich „empfehlenswerter“ Parameterwerte für den Particle-Swarm-Algorithmus `MopsoOne` durch. Hinsichtlich dieser Fragestellung wurde die Gruppe fündig in einer Veröffentlichung von Coello Coello ([11]). In dieser Arbeit wurden die Einstellungen des `MopsoOne` für die `Deb`-Testfunktion angegeben; darauf stützten sich dann die weiteren Untersuchungen. Leider beschreibt Coello Coello nicht, auf welchem Wege er diese Einstellungen ermittelt hat, wie genau diese eingehalten werden müssen (also wie robust sie sind), und auch nicht, wie sehr diese Einstellungen vom zu optimierenden Problem abhängen. Somit nahm sich diese Arbeitsgruppe der PG447 dieser Frage an.

7.3.2.2 Erste Parameter-Einstellungen

Die im Artikel von Coello Coello ([11]) für den `MopsoOne` und die `Deb`-Testfunktion angegebenen Parametereinstellungen waren im Einzelnen:

- 100 Generationen
- 400 Partikel
- inertia (die Trägheit der Partikel) von 0.4
- c1 (Gewichtung des Gedächtnisses des jeweiligen Partikels) gleich 1
- c2 (Gewichtung des globalen Gedächtnisses aller Partikel) gleich 1
- 30 Hyperkuben
- maximale Repository-Größe von 200 Partikeln
- maximale Geschwindigkeit der Partikel von 100

Die resultierende approximierte Pareto-Front mit den Einstellungen von Coello Coello zeigt Abbildung 47 links. Eine erste Variation zur „Gedächtnis-Gewichtung“ (`c1` und `c2`) ergab eine deutlich schlechtere Annäherung (siehe Abbildung 47 rechts) und zeigt, dass diese Parameter einen deutlichen Einfluss auf das Ergebnis haben.
Nach dieser ersten visuellen Auswertung wurde der `MetricsPlotter` (5.3.3) benutzt; ein von unserer Projektgruppe entwickeltes und implementiertes Visualisierungs-Tool, welches die Ausgaben (die Funktionswerte der gefundenen Lösungen) eines Algorithmenlaufs durch eine Metrik bewertet und diese Ausgabewerte in einem *Gnuplot*-Balkendiagramm graphisch darstellt.
In Abbildung 48 sieht man nun eine Ausgabe eines einzelnen Algorithmen-Laufes; es wurde der `MopsoOne` mit der `Deb`-Testfunktion gewählt; die Parametereinstellungen entsprachen wieder Coello Coellos Vorgaben, allerdings wurden `c1` und `c2` jeweils in $0,1$-er Schritten zwischen 1 und 2 variiert. Die Metrikenwerte wurden mit der `SMetrik` errechnet; dabei sind höhere Werte als „besser“ zu werten, da die Fläche zwischen den Ausgabewerten im Funktionsraum und dem darüber liegenden Referenzpunkt bei besserer Approximation größer wird.

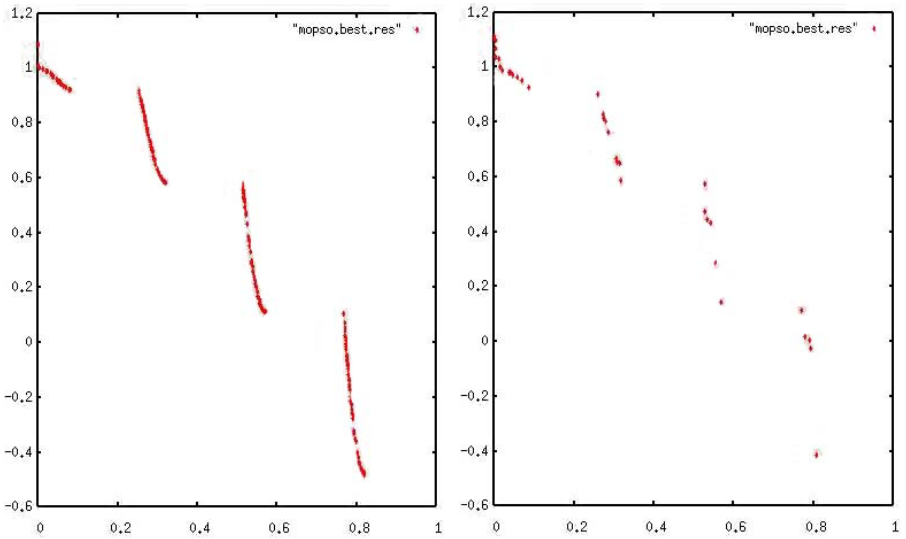

Abbildung 47: Die linke Abbildung zeigt die Lösungen des `MopsoOne` auf der `Deb`-Funktion mit den Parameter-Einstellungen nach Coello Coello; in der rechten Abbildung sind dagegen `c1` und `c2` von jeweils 1 auf jeweils 2 erhöht; die Abdeckung der Front ist nun wesentlich schlechter

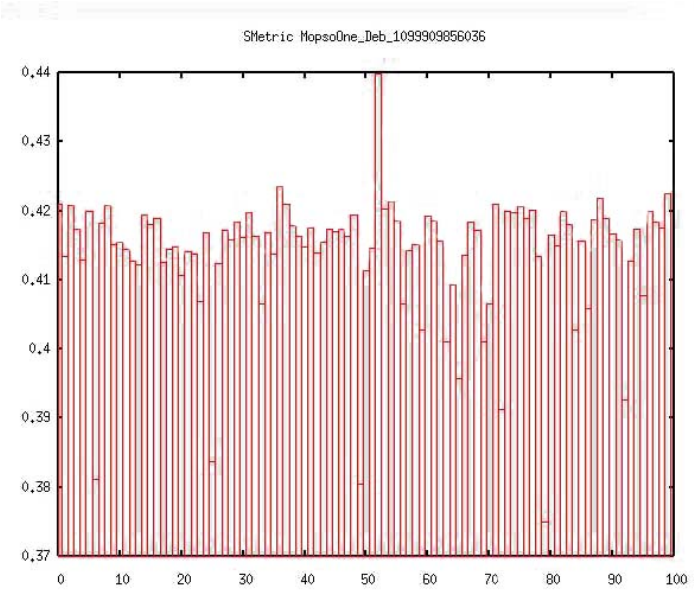

Abbildung 48: S-Metrik-Werte des `MopsoOne` auf der `Deb`-Funktion mit den Parameter-Einstellungen nach Coello Coello, aber `c1` und `c2` in $0,1$-er-Schritten zwischen 1 und 2 variiert; man sieht deutliche Unterschiede in den Metrikenwerten

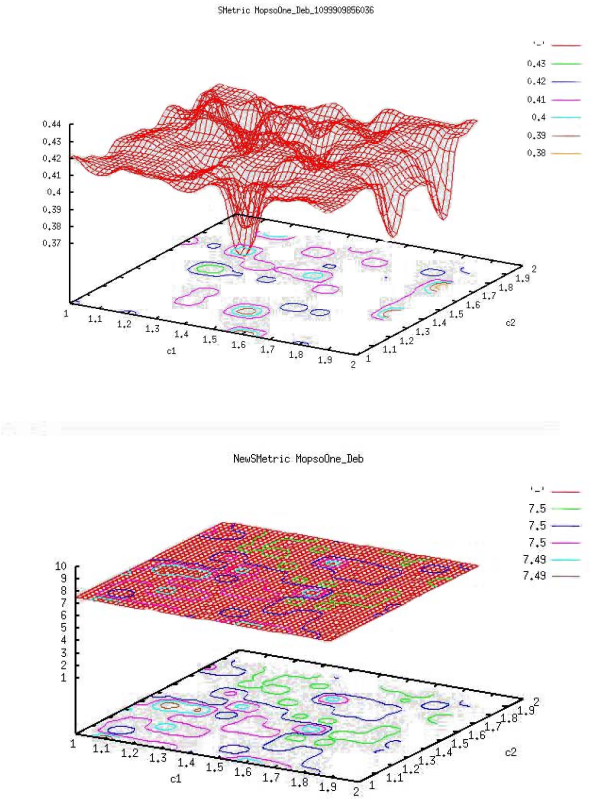

Abbildung 49: Oben: S-Metrik-Werte eines einzelnen Laufes des `MopsoOne` auf der `Deb`-Funktion mit den Parameter-Einstellungen nach Coello Coello, `c1` und `c2` zwischen 1 und 2 variiert als Surface-Plot. Hier zeigen sich leichte Unterschiede. Unten: Die selben Einstellungen, jedoch Metrik-Werte über 100 Algorithmen-Läufe gemittelt; es ergeben sich kaum noch sichtbare Unterschiede

Da mit dieser isolierten Betrachtung einzelner Parameter keine nennenswerten Aussagen über das Algorithmen-Verhalten getroffen werden können, wurde in der Robustheitsgruppe das Tool `SurfacePlotter` (5.3.5) entwickelt, mit dem, analog zum `MetricsPlotter`, die Metrikenwerte von verschiedenen Algorithmenläufen dargestellt werden können. Allerdings wird beim `SurfacePlotter` bezüglich **zweier** variierter Parameter der Metrikenwert ausgegeben, es entsteht eine Fläche (deren Höhe der Metrikenwert bestimmt) in Abhängigkeit der beiden Parameter. Einen solchen SurfacePlot bezüglich `c1` und `c2` zu den oben angegebenen Parameter-Werten zeigt Abbildung 49 links. Man kann als „Beulen“ in der Fläche die Unterschiede der Algorithmen-Läufe sehen.

Das Problem dieser Darstellung ist allerdings, dass hier die variierten Einstellungen von `c1` und `c2` auf jeweils einem einzigen Algorithmen-Lauf betrachtet werden. Da der `MopsoOne` allerdings mit Zufallswerten für die Bewegung der einzelnen Partikel

(Partikel = Punkt im Suchraum) arbeitet, hängen die Ergebnisse der einzelnen Algorithmenläufe auch in gewissem Maße von den Zufallswerten, also vom Random-Seed, dem Startwert der Zufallszahlenerzeugung, ab.

Um diesen Einfluss der Zufallszahlen auf die Algorithmenläufe zu eliminieren, wurden bei allen weiteren Untersuchungen der Robustheitsgruppe jeweils mehrere Algorithmenläufe (mindestens fünf) mit zwar exakt denselben Parametereinstellungen durchgeführt, allerdings mit unterschiedlichem Random-Seed, also intern anderen Zufallszahlen. Die daraus gewonnenen Metrikenwerte der n Läufe mit unterschiedlichem Random-Seed wurden nun arithmetisch gemittelt, also gemäß folgender Berechnung:

$$gemitteltesMetrik = \frac{\sum allerMetrikenwerte}{n}$$

Durch diese Vorgehensweise wurden die Abweichungen durch die Zufallszahlen „geglättet“; es ergeben sich ausgeglichenere Kurven ohne die Ausreißer einzelner Algorithmenläufe. Für die oben bereits genannten Einstellungen ergab sich durch die Mittelung 100 verschiedener Algorithmenläufe die Abbildung 49 rechts. Zu beachten ist hier, dass ein anderer Referenzpunkt benutzt wurde und der Maßstab so gewählt wurde, dass die gesamte z-Achse von 0 bis zu den Metrikenwerten sichtbar ist (vorher zwischen $0,37$ bis $0,44$, also eine Differenz von lediglich $0,07$; später zwischen 0 und 10, also Differenz von 10). Es ergibt sich eine nahezu ebene Fläche, was darauf hindeutet, dass die Variation der `c1`- und `c2`-Werte hier keinen bedeutenden Unterschied macht.

7.3.2.3 Gedächtnisgewichtung

Wie bereits oben erwähnt, steuern die Parameter `c1` und `c2` des `MopsoOne` den Einfluss von bisher gefundenen „guten“ Lösungen auf die Bewegung der Partikel; sie gewichten das Partikel-Gedächtnis. `c1` steuert den Einfluss der bisher vom jeweiligen Partikel gefundenen besten Lösung; `c2` steuert den Einfluss der vom gesamten Schwarm besten gefundenen Lösung. Nun wurde im Speziellen näher untersucht, wie diese beiden Parameter eingestellt werden sollten, damit der `MopsoOne` möglichst gut optimiert. Später wird dann beschrieben, wie die Robustheitsgruppe allgemeinere Aussagen über alle wichtigen Parameter des Algorithmus gefunden hat; hier werden zunächst nur Versuche bezüglich `c1` und `c2` beschrieben.

Die folgenden Versuche wurden allesamt mit dem `SurfacePlotter` vorgenommen. Dies war eine einfache und übersichtliche Methode, allerdings ist diese Art der Vorgehensweise im streng wissenschaftlichen Sinn nicht sehr aussagekräftig. Hierdurch konnte allerdings ein erster Eindruck, ein „Gefühl“ für den Einfluss der Parameter gewonnen werden. Bei dieser Methode der Surface-Begutachtung wird nur der Einfluss von `c1` und `c2` gemessen; alle anderen Parameter werden auf jeweils einem Wert festgehalten. Dadurch werden allerdings Wechselwirkungen mit den anderen Parametern nicht aufgedeckt; vor allem der `inertia`-Wert spielt in gewissen Grenzen mit `c1` und `c2` zusammen. Allerdings lässt sich im Folgenden z. B. erkennen, dass der Wert von `c1` wesentlich unwichtiger für die Funktionalität des Algorithmus ist, als `c2`.

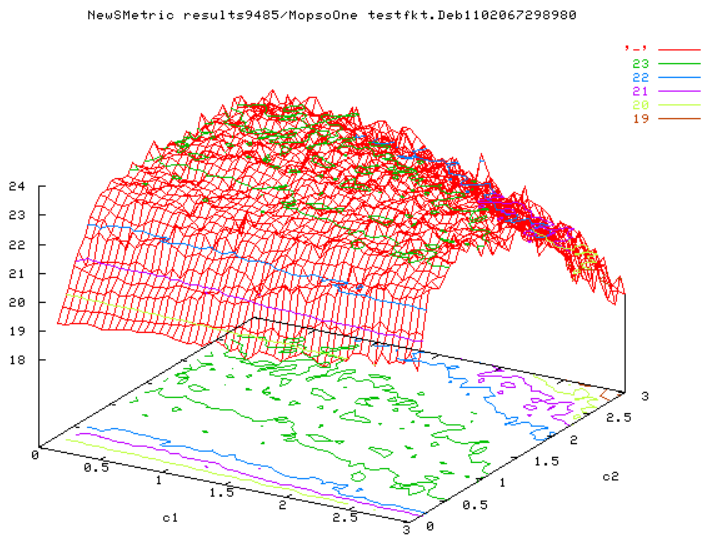

Abbildung 50: Surface-Plot der S-Metrik-Werte mit Variation von `c1` und `c2` zwischen jeweils 1 und 3 (`Deb`-Testfunktion). Man erkennt eine Abhängigkeit der Metrik-Werte von den `c2`-Werten

Wie bereits Abbildung 49 rechts gezeigt hat, variieren die Metrik-Werte bei den Einstellungen aus der Literatur fast gar nicht mehr; zumindest nicht so stark, um einen signifikanten Unterschied oder sogar einen Trend erkennen zu können. Um dies zu ändern, wurden die Anzahl der Generationen, sowie die Anzahl der Partikel stark reduziert, so dass sich nach Ende des Algorithmenlaufs der Algorithmus noch in der Anfangsphase befand und noch nicht sehr nah bei der gesuchten Pareto-Front war. So waren Unterschiede in den Algorithmenläufen stärker erkennbar.

Im Weiteren wurden folgende Parameterbelegungen verwendet:

- 50 Generationen
- 5 Partikel
- inertia (die „Trägheit“ der Partikel) von 0.9
- c1 zwischen 0,1 und 3 in 0,05-er Schritten
- c2 zwischen 0,1 und 3 in 0,05-er Schritten
- 30 Hyperkuben
- maximale Repository-Größe von 200 Partikeln
- maximale Geschwindigkeit der Partikel von 100
- Mittelung der Metrikenwerte über 5 Läufe mit unterschiedlichem Random-Seed

Diese Parameterbelegung für die `Deb`-Testfunktion hat, in Kombination mit der S-Metrik, das Ergebnis aus Abbildung 50 erzeugt. Es zeigt sich, dass die entstehende Fläche vor allem in `c2`-Richtung gewölbt ist. Das Maximum bezüglich `c2` befindet sich etwa im Bereich zwischen 1 und 2. Vor allem für kleinere `c2`-Werte ergeben sich wenig Änderungen der Fläche in Bezug auf den Parameter `c1`; erst ab `c2`-Werten von größer 2 lässt sich auch ein stärkerer Einfluss von `c1` erkennen.

Um nun Aussagen im Hinblick auf die Robustheit dieser Einstellungen von `c1` und `c2` treffen zu können, wurden weitere Optimierungs-Probleme aus dem Testfunktionen-Fundus der Projektgruppe ausgetestet.

Parameter c1 und c2 bei der DoubleParabola Die `DoubleParabola` (siehe 5.4.1.1) ist die einfachste Testfunktion, die von der Projektgruppe implementiert wurde; sie stellte keinen der implementierten und ausgetesteten Algorithmen vor irgendein Problem. Aufgrund ihrer Einfachkeit wurden die Pareto-optimalen Punkte stets schnell gefunden.

Um nun die als gut zu empfehlenden Einstellungen für `c1` und `c2` bei der `DoubleParabola` herauszufinden, wurden folgende Einstellungen des `MopsoOne` in Anlehnung an Coello Coello ([11]) verwendet:

- 100 Generationen
- 100 Partikel
- inertia (die „Trägheit“ der Partikel) von 0.9
- c1 zwischen 0,2 und 3,5 in 0,2-er Schritten
- c2 zwischen 0,2 und 3,5 in 0,2-er Schritten
- 30 Hyperkuben
- maximale Repository-Größe von 200 Partikeln
- maximale Geschwinddigkeit der Partikel von 100
- Mittelung der Metrikenwerte über 5 Läufe mit unterschiedlichem Random-Seed

Das Ergebnis dieser Algorithmenläufe zeigt Abbildung 51 als SurfacePlot. Vollkommen überraschend zeigt sich, dass alle Einstellungen für `c1` und `c2` von jeweils $0,2$ bis jeweils etwa 2 gleiche Metrikenwerte erzeugen; erst ab Werten von jeweils etwas über 2 zeigte sich ein Abfall des Metrikenwertes.

Eine Erklärung hierfür lässt sich in der Formulierung und Implementierung der Testfunktion `DoubleParabola` finden. Die `DoubleParabola` ist ein sehr einfaches mehrkriterielles Optimierungs-Problem; zudem werden die Startpositionen der Partikel des `MopsoOne` stets aus der Nähe der Pareto-optimalen Lösungen der Testfunktion ausgewählt. Somit halten sich die Partikel schon zu Beginn des jeweiligen Algorithmenlaufes sehr nahe der Pareto-optimalen Punkte auf; der Einfluss der bisher besten

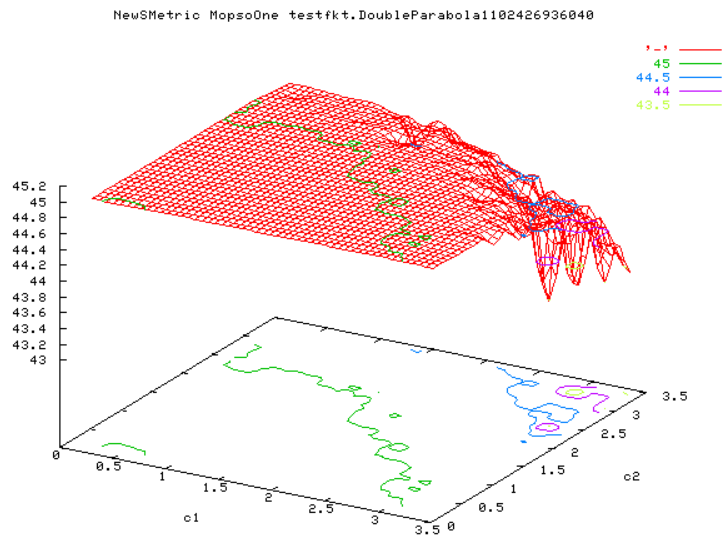

Abbildung 51: Surface-Plot mit Variation von `c1` und `c2` auf der `DoubleParabola`; erst jenseits $(2; 2)$ fallen die S-Metrik-Werte ab

gefundenen Lösungen der Partikel (ob nun lokal=`c1` oder global=`c2`) ist somit unbedeutend. Erst wenn die Werte für `c1` und `c2` zu groß werden, wird durch deren Werte der Geschwindigkeitsvektor jedes Partikels derartig hoch eingestellt, dass die Partikel an den Pareto-optimalen Punkten „vorbeischießen“ und somit mit schlechteren Lösungen auch einen schlechten Metriken-Wert erzeugen.

Parameter c1 und c2 bei der Schaffer-Funktion Die `Schaffer`-Testfunktion (siehe 5.4.1.5) ist eine weitere Funktion aus der Testfunktions-Sammlung der Projektgruppe. Da auch diese für unsere Algorithmen verhältnismäßig „einfach“ zu optimieren ist, wurden bei den Versuchen zu dieser Funktion gezielt nur sehr wenige Generationen, sowie auch sehr wenige Partikel bei der Einstellung des `MopsoOne` gewählt; ansonsten stimmen die Parameter-Werte mit denen von oben überein:

- 10 Generationen
- 10 Partikel
- inertia von 0.9
- c1 zwischen 0 und 3,5 in 0,2-er Schritten
- c2 zwischen 0 und 3,5 in 0,2-er Schritten
- 30 Hyperkuben
- maximale Repository-Größe von 200 Partikeln
- maximale Geschwindigkeit der Partikel von 100

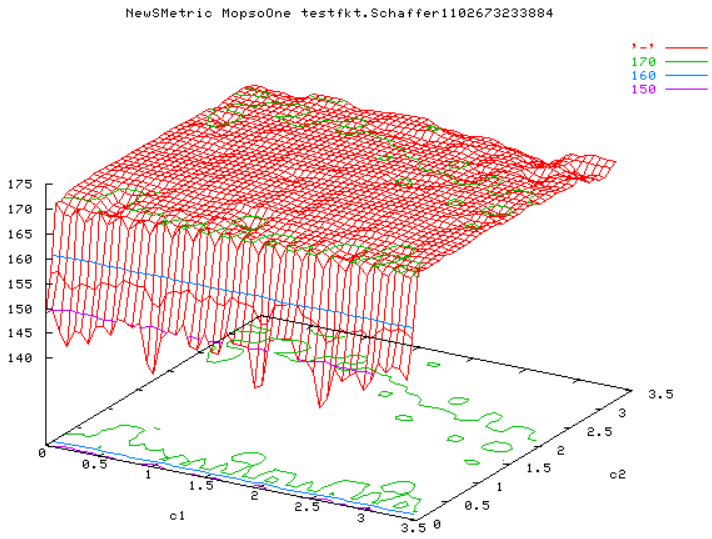

Abbildung 52: `Surface`-Plot mit Variation von `c1` und `c2` auf der `Schaffer`-Funktion; ab `c2`=0.1 bleiben die S-Metrik-Werte gleich gut

- Mittelung der Metrikenwerte über 10 Läufe mit unterschiedlichem Random-Seed

Die Abbildung 52 zeigt nun einen SurfacePlot des `MopsoOne` auf der `Schaffer`-Funktion mit verschiedenen `c1`- und `c2`-Einstellungen. Hier ist wieder zu beobachten, dass ab `c1`=3 und `c2`=3 sich die Metriken-Werte leicht verschlechtern; zudem liegen die Werte für `c2`=0 auch schlechter als bei anderen Einstellungen.

Dieses Phänomen lässt sich dadurch erklären, dass der Wert von `c2` einen wesentlich größeren Einfluss auf den Verlauf des Algorithmus hat, als dies bei `c1` der Fall ist. So beeinflusst das „lokale Gedächtnis“ welches ja von `c1` gewichtet wird, das Verhalten des Schwarms wesentlich weniger, als dies `c2` tut. Ist `c1`=0, so gibt es ja immer noch den `c2`-Wert, der dem Schwarm eine „Orientierung“ bietet. Somit lässt sich als Empfehlung erahnen, dass die Werte von `c1` und `c2` jeweils im Bereich $0<c1;c2<3$ liegen sollten.

Festzuhalten bleibt aber auch noch die Erkenntnis, dass diese Einstellungen doch merkbar abweichen von den Einstellungen für die `Deb`-Funktion (siehe 7.3.2.3), bei der ja vor allem für `c2`-Werte um etwa $1,5$ der Algorithmus gute Ergebnisse liefert und bei größeren Werten wieder schlechter wird.

Vorläufige Erkenntnisse Die oben beschriebenen Experimente haben bislang nur gezeigt, dass der Einfluss der Parameter `c1` und `c2` für die gute Approximation an Pareto-optimale Lösungen für den Algorithmus `MopsoOne` von Bedeutung ist. Allerdings konnte bislang nicht geklärt werden, ob überhaupt, und wenn ja, wie stark diese Einstellungen von den anderen Parametern abhängen, bzw. mit diesen wechselwirken.

7.3.2.4 Regressionsanalyse zum MopsoOne auf der Deb-Funktion

Durch die obigen Versuche wurde nun gezeigt, dass vor allem die Parameter für das „Flugverhalten“ der Partikel (`c1`, `c2` und `inertia`) einen großen Einfluss auf das Algorithmus-Verhalten haben. Ebenso ist der Parameter `quantityParticle`, welcher die Anzahl der Partikel im Schwarm festlegt, bedeutend, da er ja über die gleichzeitig zu überprüfenden Punkte im Suchraum entscheidet.

Als notwendige Bedingung für eine geplante weitere Analyse dieser einzelnen vier Parameter des `MopsoOne` wurde nun eine Regressionsanalyse durchgeführt.
Dazu wurden die Parameter folgendermaßen variiert: `inertia`-Werte $0,1$; $0,3$; $0,5$; $0,7$; $0,9$ und $0,99$; `quantityParticle` (die Anzahl der Partikel) von 10 bis 25 in Fünferschritten; Anzahl der Funktionsauswertungen fest auf 5000 (somit passte sich die Anzahl der Generationen an die Partikel-Anzahl so an, dass immer 5000 Funktionsauswertungen erreicht wurden), `c1` von $0,5$ bis $1,5$ in $0,3$-er Schritten und `c2` von $0,5$ bis $1,5$ in $0,3$-er Schritten. Die Parameter `maxVelocity` (maximale Partikel-Geschwindigkeit), `quantityHypercube` (Anzahl der Hyperkuben) und `maxRepository` (Größe des Archives) wurden auf 100, 30, bzw. 200 fest gesetzt. Ferner wurden die Ergebnisse über 5 Algorithmen-Läufe gemittelt. Die Metrikenwerte, auf die diese Analyse zurück geht, wurden mit der S-Metrik nach Fleischer gewonnen.

Abbildung 53 zeigt die Regressionsanalyse der vier Parameter; die Grafik wurde mit dem Statistik-Tool *R* erzeugt, der Metrikenwert wird dort mit „Y“ bezeichnet. Dabei wurde das interne, lineare Modell `Y~(inertia * c1 * c2 * quantityParticle)` für die statistischen Analysen verwendet. Der „Normal Q-Q-Plot“ zeigt, dass die Variationen des Metrikenwertes annähernd normalverteilt bezüglich der variierten Parameter sind. Somit war die Grundlage für die weitere Untersuchung der „guten“ Einstellungen für diese Parameterwerte geschaffen.

7.3.2.5 Lineares Modell des MopsoOne auf der Deb-Funktion

Aus dem Datensatz der Regressionsanalyse sollte nun für jeden einzelnen der vier Parameter `inertia`, `c1`, `c2` und `quantityParticle` bestimmt werden, welche die zu empfehlenden Einstellungen für diesen Parameter bezüglich der `Deb`-Testfunktion sind. So wurde mittels *R* zunächst eine Analyse dieses linearen Modells durchgeführt, um die für den Einfluss auf den Metrikenwert signifikanten Parameter-Einstellungen heraus zu finden. Zu dem Datensatz, der bereits für die Regressionsanalyse verwendet wurde, gab *R* Abbildung 54 aus.

Man sieht nun, dass vor allem die Parameter `inertia` und `c2`, aber auch die Anzahl der Partikel wesentlichen Einfluss auf das Verhalten des Algorithmus haben. Zudem scheint das Verhältnis von `inertia` zu `c2` von Bedeutung zu sein. Dagegen ist der Parameter `c1` von geringerer Bedeutung für die Leistung des Algorithmus, wie ja auch schon in 7.3.2.3 beschrieben wurde. Die „Erinnerung“ (genauer: der Einfluss dieser Erinnerung auf die zukünftige Bewegung) der Partikel an die global beste Lösung ist wichtiger als die Erinnerung an die jeweils besten Lösungen der einzelnen Partikel.

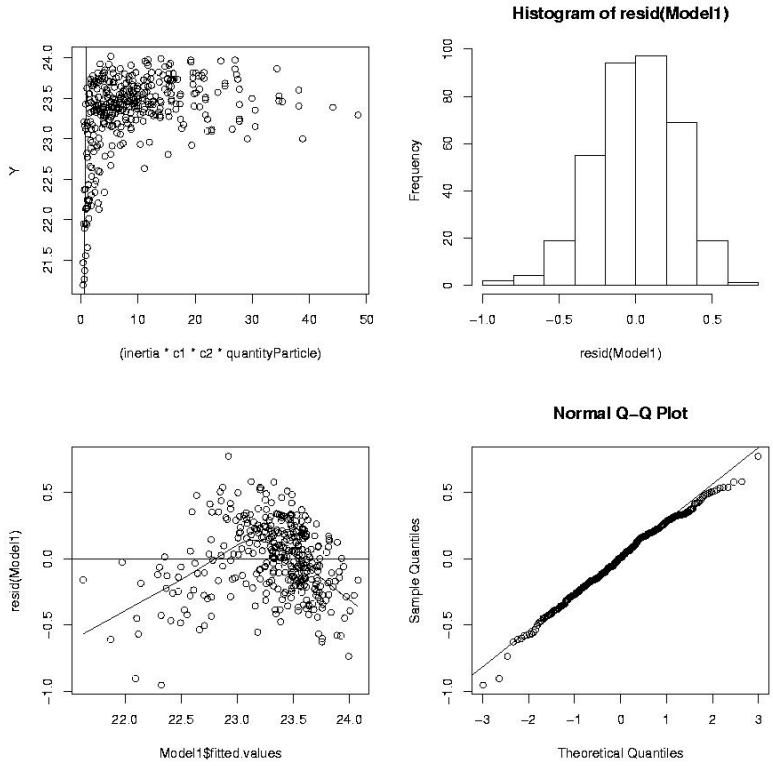

Abbildung 53: Regressionsanalyse von 4 Parametern des `MopsoOne` auf der `Deb`-Testfunktion; eine Normalverteilung der Metrikenwerte wird deutlich

7.3.2.6 Factorial Design zum MopsoOne

Um nun vernünftige Ergebnisse in Bezug auf die Parameter des `MopsoOne`, auf deren Wechselwirkung und deren Robustheit zu erhalten, wurde ein 2^4-Factorial-Design erstellt. Dabei konzentrierte sich die Arbeitsgruppe auf die Parameter `inertia`, `c1`, `c2`, sowie `quantityParticle`, die Anzahl der Partikel; diese wurden ja bereits in der Regressionsanalyse untersucht. Zudem sind mit den Parametern `c1`, `c2` und `inertia` die Parameter vertreten, die für die Bewegung der einzelnen Partikel, und somit auch des gesamten Schwarms, verantwortlich sind. Somit kann gerade die Besonderheit der Particle-Swarm Algorithmen, die Bewegung der Partikel, untersucht werden.

Der Parameter `g`, der die Anzahl der zu durchlaufenden Generationen angibt, wurde im Factorial Design nicht vernachlässigt; er wurde lediglich an die Partikel-Anzahl gekoppelt. Die Anzahl der Funktionsauswertungen, d. h. die Anzahl der Aufrufe einer Testfunktion, errechnet sich nämlich aus dem Produkt von Partikel-Anzahl und Generationenzahl. Um nun eine konstante Anzahl von Funktionsauswertungen zu erhalten, wurde bei steigenden Partikelzahlen die Generationen-Anzahl entsprechend gesenkt, so dass stets eine konstante Anzahl von Funktionsauswertungen sichergestellt wurde. Dieses Vorgehen war notwendig, da ja ein Algorithmenlauf mit mehr Funktionsaus-

```
Call:
lm(formula = Y ~ (inertia * c1 * c2 * quantityParticle), data = f1)

Residuals:
      Min        1Q    Median        3Q       Max
-0.954296 -0.174431  0.007343  0.197585  0.773732

Coefficients:
                                Estimate Std. Error t value Pr(>|t|)
(Intercept)                     16.94701    1.17238  14.455  < 2e-16 ***
inertia                          7.39064    1.77075   4.174 3.80e-05 ***
c1                               2.77102    1.16368   2.381  0.01780 *
c2                               4.73900    1.08562   4.365 1.68e-05 ***
quantityParticle                 0.15878    0.05975   2.658  0.00824 **
inertia:c1                      -2.93434    1.75761  -1.670  0.09593 .
inertia:c2                      -5.73624    1.63971  -3.498  0.00053 ***
inertia:quantityParticle        -0.17689    0.09024  -1.960  0.05078 .
c1:c2                           -1.89912    1.07757  -1.762  0.07889 .
c1:quantityParticle             -0.09367    0.05930  -1.580  0.11514
c2:quantityParticle             -0.10325    0.05606  -1.842  0.06637 .
inertia:c1:c2                    1.98711    1.62755   1.221  0.22295
inertia:c1:quantityParticle      0.09891    0.08957   1.104  0.27024
inertia:c2:quantityParticle      0.13757    0.08467   1.625  0.10513
c1:c2:quantityParticle           0.06793    0.05565   1.221  0.22302
inertia:c1:c2:quantityParticle -0.07443    0.08405  -0.886  0.37646
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.2771 on 344 degrees of freedom
Multiple R-Squared: 0.6872,     Adjusted R-squared: 0.6735
F-statistic: 50.37 on 15 and 344 DF,  p-value: < 2.2e-16
```

Abbildung 54: Analyse des linearen Modells der `MopsoOne`-Parameter

wertungen als ein Anderer auch weiter optimieren kann als der Andere.

Für das Factorial Design wurden für die vier Parameter folgende Wertepaare ausgewählt: `c1` und `c2` jeweils gleich $0,5$ und $1,5$; `inertia` gleich $0,5$ und $0,9$; `quantityParticle` (die Anzahl der Partikel) gleich 10 und 25. Somit ergab sich die Versuchstabelle 10. Als zu untersuchende Testfunktion wurde die `Deb`-Funktion gewählt, da die Ausgangswerte für die Parameter aus [11] für genau diese Funktion angegeben wurden. Dabei wurde der Wert der Funktionsauswertungen auf 500 festgehalten; so wurde der Algorithmus bei 10 Partikeln 50 Generationen lang laufen gelassen, bei 25 Partikeln wurden ihm nur 20 Generationen zugestanden. Für die Ergebnisse wurden die Metrikenwerte von 20 verschiedenen Algorithmenläufen gemittelt; für die Bewertung wurde die S-Metrik nach Fleischer benutzt.

Aus den gewonnenen Daten wurde mittels der Statistik-Software *R* ein Interactionplot erstellt, welcher in Abbildung 55 zu sehen ist. Man bemerkt, dass der Parameter `inertia` scheinbar den größten Anteil an der Algorithmus-Güte hat. Als problematisch, aber gleichzeitig auch aufschlussreich, zeigt sich in diesem Interactionplot das

Tabelle 10: Wertetabelle zum Factorial Design beim `MopsoOne`

inertia	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
c1	0.5	0.5	0.5	0.5	1.5	1.5	1.5	1.5
c2	0.5	0.5	1.5	1.5	0.5	0.5	1.5	1.5
quantityParticle	10	25	10	25	10	25	10	25
inertia	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9
c1	0.5	0.5	0.5	0.5	1.5	1.5	1.5	1.5
c2	0.5	0.5	1.5	1.5	0.5	0.5	1.5	1.5
quantityParticle	10	25	10	25	10	25	10	25

Verhältnis zwischen dem `inertia`-Wert und dem `c2`-Wert. Hier scheint also eine Wechselwirkung zu bestehen.

7.3.2.7 Wie sollten die Parameter nun eingestellt werden?

Aus dem Datensatz, der schon für die Regressionsanalyse und für die Analyse des linearen Modells der Parameter benutzt wurde, wurden nun Boxplots erstellt, aus welchen man die Verteilung der Metrikenwerte zu den Parametereinstellungen ablesen kann. Diese Boxplots sind in den Abbildungen 56 und 57 zu sehen. Mit diesen Boxplots ist es nun möglich, zu sagen, welche Einstellungen für die Parameter zu empfehlen sind.

Abbildung 56 links zeigt den Einfluss des Wertes des `inertia`-Parameters auf den Metrikenwert. Dabei gehen alle Algorithmenläufe der oben beschriebenen Parameter-Einstellungen in das Ergebnis mit ein. So sind beim `inertia`-Wert von $0,7$ die meisten Algorithmenläufe im Bereich hoher Metrikenwerte; dies gilt für **alle** anderen Parameter-Einstellungen (also egal, wie `c1`, `c2` und `quantityParticle` gewählt werden) aus dem oben angegebenen Bereich.

Somit kann nun eindeutig gesagt werden, dass beim `MopsoOne` und der `Deb`-Funktion der Parameter `inertia` am besten im Bereich um $0,7$ liegen sollte, wenn sich die anderen drei Parameter in den oben erwähnten Bereichen befinden ($0,5 \leq c1 \leq 1,5$; $0,5 \leq c2 \leq 1,5$; $10 \leq quantityParticle \leq 25$).

In Abbildung 56 rechts sieht man nun die Verteilung der Metrikenwerte für den Parameter `c1`. Die Erklärungen zum Boxplot von `inertia` gelten genauso hier. Allerdings ist das Ergebnis ein vollkommen anderes: Man kann bei weitem nicht so einfach eine empfehlenswerte Einstellung zu `c1` finden. Man sieht mit einem Blick, dass die Einstellung dieses Parameters relativ egal ist; lediglich bei $1,1$ scheinen sich die Metrikenwerte etwas stärker zu häufen als bei den anderen Werten. Diese Beobachtung deckt sich mit den aus Kapitel 7.3.2.3 gewonnenen Beobachtungen.

Die Abbildung 57 links zeigt den Einfluss des Parameters `c2`; man erkennt hier einen guten Bereich um $1,1$. Allerdings zeigt auch der Wert $1,4$ ein gutes Verhalten; bei weiterführenden Untersuchungen wäre es zu empfehlen, hier auch noch die etwas größe-

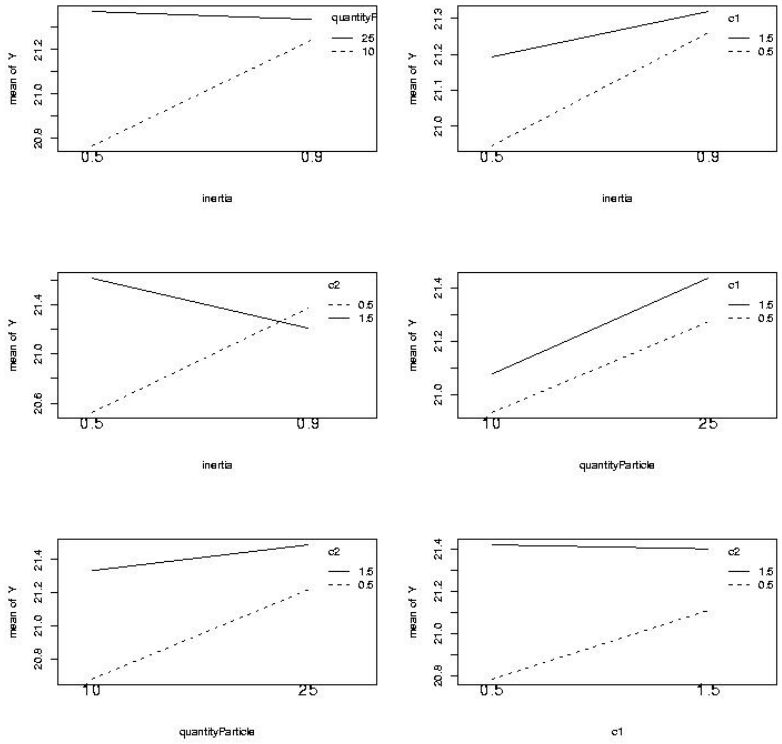

Abbildung 55: Interactionplot von vier Parametern des `MopsoOne` auf der `Deb`-Testfunktion; man erkennt Wechselwirkungen zwischen `inertia` und `c2`

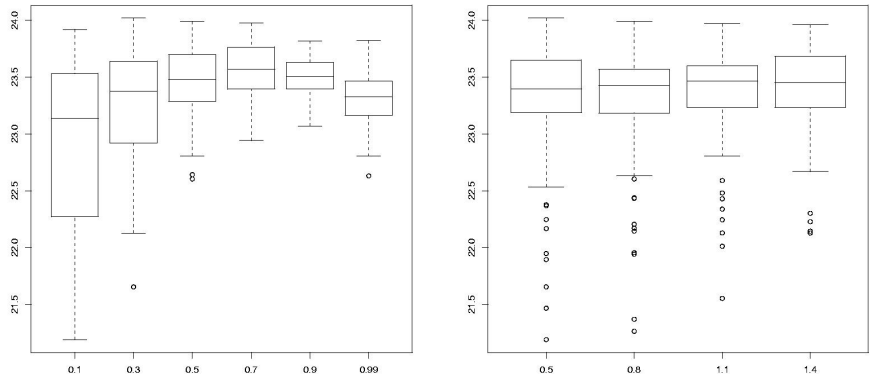

Abbildung 56: Links: Boxplot zum Parameter `inertia` beim `MopsoOne` auf der `Deb`-Funktion; bei $0,7$ ergeben sich die besten Werte. Rechts: Boxplot zum Parameter `c1` beim `MopsoOne` auf der `Deb`-Funktion; es sind kaum Unterschiede in den Metrik-Werten zu erkennen

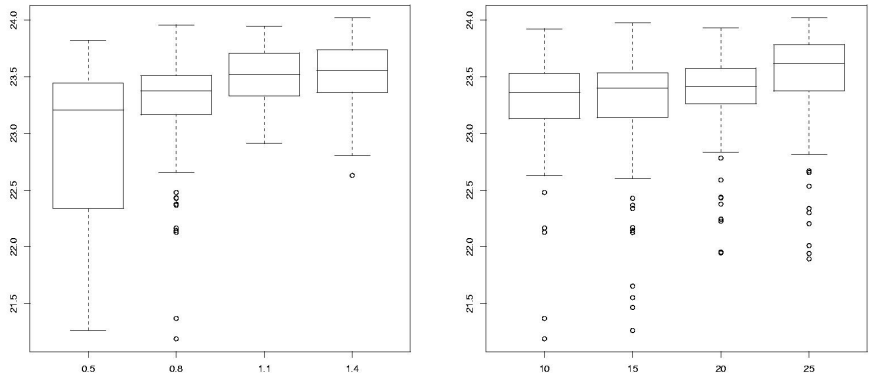

Abbildung 57: Links: Boxplot zum Parameter `c2` beim `MopsoOne` auf der `Deb`-Funktion; Werte um $1,1$ scheinen die bestenErgebnisse zu erzeugen. Rechts: Boxplot zum Parameter `quantityParticle` beim `MopsoOne` auf der `Deb`-Funktion; größere Werte liefern bessere Metriken-Ergebnisse

ren Bereiche für `c2` zu untersuchen.

In Abbildung 57 rechts sieht man nun entsprechend die Verteilung der Metrikenwerte für den Parameter `quantityParticle`. Man erkennt hier deutlich, dass die Werte für größere Partikel-Anzahlen besser werden. Dies lässt sich leicht dadurch erklären, dass ein größerer Schwarm ja schneller ein größeres Gebiet absuchen kann. So sollten für weitere Untersuchungen auch noch größere Partikel-Anzahlen betrachtet werden. Allerdings ist aufgrund der feststehenden Anzahl von Funktionsauswertungen und der dadurch bedingten Koppelung der Anzahl der Generationen mit der Anzahl der Partikel zu erwarten, dass die Metrikenwerte mit mehr Partikeln zwar steigen, bei einem bestimmten Wert aber wieder abfallen. Dann wäre zwar ein großer Schwarm vorhanden, wegen der Koppelung der Schwarmgröße mit der Generationenanzahl wäre aber die Anzahl der Generationen zu niedrig, um noch nahe genug an die Pareto-optimalen Punkte zu gelangen. Es müsste also eine Art Maximum der Metrikenwerte bezüglich der Anzahl der Partikel geben.

Ferner wurde ein weiterer Interactionplot mit *R* erstellt, da ja das Verhältnis vom `inertia`-Wert zur Anzahl der Partikel noch dargestellt werden musste. Diese Grafik ist in Abbildung 58 zu sehen. Man erkennt, dass die größtmögliche Partikel-Zahl (25), sowie der `inertia`-Wert von $0,7$ zu den besten Ergebnissen führen. Allerdings wird auch sichtbar, dass bei kleineren `inertia`-Werten zum Teil auch eine kleinere Partikel-Anzahl zu empfehlen ist. Die Ursache für dieses Verhalten ist bislang nicht bekannt und wäre ein interessantes Feld für weiter gehende Untersuchungen.

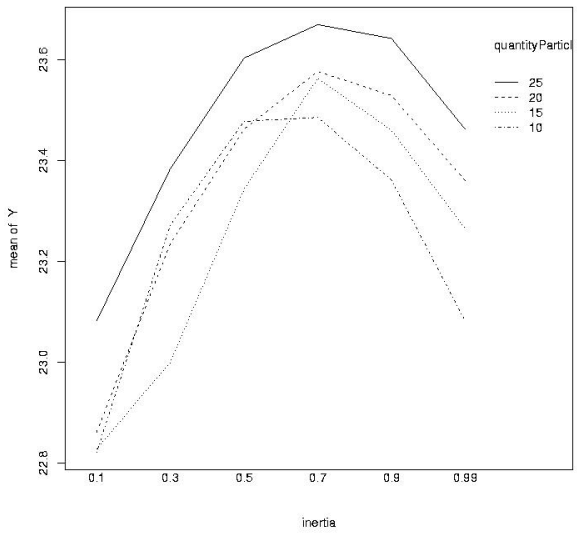

Abbildung 58: Interaction-Plot für `inertia` und `quantityParticle` beim `MopsoOne` auf der `Deb`-Funktion; es ist zu bemerken, dass bei niedrigen `inertia`-Werten die Verläufe der niedrigen Werte für `quantityParticle` einander schneiden

7.3.2.8 Fazit und Ausblick

Es wurde in den obigen Untersuchungen gezeigt, wie die Parameter des `MopsoOne` dessen Verhalten beeinflussen, und wie stark einzelne Parameter daran beteiligt sind. Einige Parameter wurden als weniger einflussreich charakterisiert; bei anderen Parametern konnte gezeigt werden, dass sie sehr wohl einen deutlichen Einfluss auf den Erfolg des Algorithmus haben.

Die hier vorgestellte Analyse der Parameter-Einstellungen war zwar aufwändig, aber doch lohnenswert. Es hat sich gezeigt, dass es Sinn macht, sich bei der Optimierung nicht mit dem erstbesten Ergebnis eines Algorithmenlaufs zufrieden zu geben, sondern dass Parameter-Tuning die Ergebnisse des Algorithmus teilweise stark verbessern kann. Zwar liefert der `MopsoOne` mit „Standard-Einstellungen“ schon gute Ergebnisse, aber bei wichtigen Projekten lohnt der Aufwand des Parameter-Tunings, da die Ergebnisse noch weiter verbessert werden können.

So bleibt nun fest zu halten, dass die in [11] vorgeschlagenen Parameter für den `MopsoOne` grundsätzlich schon den richtigen Weg weisen. Problemspezifisch können bessere Einstellungen gefunden werden, allerdings ist dies zum Teil umständlicher, als wenn man z. B. den Algorithmus einfach länger laufen lassen würde, zumal ja für die Suche der guten Parameter-Einstellungen der Algorithmus viele Male aufgerufen werden muss, um verlässliche Daten zu erhalten. So ist es bei rechenaufwändigen Problemen gegebenenfalls ratsamer, den Algorithmus nur länger rechnen zu lassen, anstatt ein Design mit vielen Algorithmus-Aufrufen zu starten.

Eine aufregende Idee für weitere Untersuchungen wäre es nun, z. B. eine einfache Evolutionsstrategie zu benutzen, um die besten Parametereinstellungen für den `MopsoOne` zu finden. So wäre aus dem `MopsoOne` mit seinen statischen Parameterwerten, die exogen eingestellt werden, ein Algorithmus mit selbstanpassenden Parametern geworden. Der Preis hierfür wäre allerdings eine wesentlich längere Laufzeit.

Es bleiben also noch einige Fragen offen, sowie weitere Untersuchungen; vor allem von Praxis-Gebieten. So hat eine Untergruppe unserer Projektgruppe, welche sich mit dem Problem der Temperierbohrungen bei Spritzgusswerkzeugen beschäftigt hat (siehe 5.4.3), herausgefunden, dass die Parameter des `MopsoOne` in ganz anderen Bereichen liegen sollten, als dies bei der `Deb-` oder der `Schaffer`-Funktion der Fall ist.

7.3.3 MopsoOne: Verhältnis zwischen Partikelanzahl und Generationen

7.3.3.1 Motivation

Als eine unserer Teilaufgaben wollten wir untersuchen, wie sich das Verhältnis zwischen der Anzahl der Partikel und der Anzahl der Generationen auf die Güte der Ergebnisse beim Particle Swarm Algorithmus `MopsoOne` auswirkt. Es ist bekannt, dass der `MopsoOne` schlechte Ergebnisse liefert, wenn sich die Partikel zu ähnlich sind und er deswegen eine nicht zu kleine Anzahl von Partikeln benötigt. Nach anfänglichen Überlegungen sind uns weitere Fragen eingefallen, die in diesem Zusammenhang untersucht werden könnten. Wie groß muss die Population sein, um vernünftige Ergebnisse zu liefern und welche Anzahl ist bei welcher Testfunktion optimal? Gibt es vielleicht sogar ein optimales Verhältnis zwischen der Anzahl der Partikel und der Anzahl der Generationen, so dass bei einer größeren Anzahl an Funktionsauswertungen auch eine größere Anzahl an Partikel bessere Ergebnisse liefert? Wirkt sich die Anzahl der Partikel auf die Stabilität des Algorithmus aus, so dass bei einer Vergrößerung der Population die Ergebnisse im Durchschnitt vielleicht nicht besser werden, aber eine größere Anzahl von Läufen einen zufriedenstellenden Metrikwert erreicht? Ist die optimale Anzahl der Partikel abhängig von der jeweiligen Testfunktion oder hängt sie sogar mit anderen Parametereinstellungen des Algorithmus zusammen? Das waren die Fragen, die uns zu dem Thema zunächst einfielen und die wir mit Experimenten untersuchen wollten.

7.3.3.2 Planung

Unsere Experimente wollten wir zuerst mit den `ZDT1`, `ZDT2` , `ZDT3` und `ZDT6` Testfunktionen durchführen, wobei wir uns später nach dem Parameter-Tuning aus Zeitmangel auf die `ZDT1` beschränkt haben. Als Güte für die Algorithmen-Läufe haben wir die S-Metrik nach Fleischer mit dem festen Referenzpunkt $(3; 3)$ auf die Ergebnisse angewandt, was uns eine gute Einschätzung und Vergleichbarkeit der Ergebnisse garantieren sollte. Dabei bedeutet der Metrikwert 0, dass kein Punkt der gefundenen Pareto-Front unterhalb des Referenzpunktes liegt. Um zu sehen, wie gut der Algorithmus sich der wahren Pareto-Front genähert hat, haben wir die `SmetrikFleischer` auf die wahre Pareto-Front angewandt. Die Tabelle 11 zeigt die Metrikwerte der wahren Pareto-Fronten der ZDT-Funktionen.

Tabelle 11: Maximaler Metrikwert bei Referenzpunkt $(3; 3)$

Testfunktion	optimaler Metrikwert
ZDT1	$8,665$
ZDT2	$8,328$
ZDT3	$10,591$
ZDT6	$7,764$

7.3.3.3 Parametersuche bei den ZDT-Funktionen

Um mit den Untersuchungen zu beginnen, wollten wir zuerst gute Parameter und eine angemessene Anzahl von Funktionsauswertungen für diese Testfunktionen finden,

bei denen der Algorithmus recht gute Werte liefert, bei denen aber noch Unterschiede in der Güte der Ergebnisse bei kleinen Änderungen der Einstellungen zu erkennen sind. Da wir die Parameter für die `Deb`-Funktion bereits im Kapitel 7.3.2 ausführlich untersucht und gute Einstellungen gefunden haben, haben wir uns nun mit der Suche nach guten Einstellungen des `MopsoOne` bezüglich der ZDT-Testfunktionen beschäftigt. Als erstes wollten wir die Anzahl der Funktionsauswertungen bestimmen, bei denen der Algorithmus vernünftige Ergebnisse liefert. Dazu haben wir für jede ZDT-Testfunktion zwei Designaufrufe mit folgenden Parametern gestartet:

Erster Designaufruf :
```
--t testfkt.ZDTi --a MopsoOne -g s 2000
-quantityParticle s 100 -maxRepository s 200
-quantityHypercube s 300 -inertia s 0.8 -c1 s 1.6
-c2 s 1.6 -r i 1 10 -live s 1000
```

Zweiter Designaufruf :
```
--t testfkt.ZDTi --a MopsoOne -g s 2000
-quantityParticle s 100 -maxRepository s 200
-quantityHypercube s 300 -inertia s 0.6 -c1 s 1.2
-c2 s 1.2 -r i 1 10 -live s 1000
```

Bei den zwei unterschiedlichen Versuchen haben wir die Parameter `inertia`, `c1` und `c2` leicht verändert, was aber einen sehr großen Einfluss auf die Güte der Ergebnisse hatte. Während der Algorithmus mit der ersten Parametereinstellung auf allen Testfunktionen keine Verbesserungen erzielte, wurden beim zweiten Designaufruf die Testfunktionen `ZDT1` und `ZDT3` ziemlich gut approximiert, jedoch wurde selbst mit 200000 Funktionsauswertungen bei der `ZDT2` lediglich ein Wert von $4,6$ (Maximum bei $8,3$) und bei der `ZDT6` ein Wert von $3,8$ (Maximum bei $7,7$) erreicht. Hiermit wurde uns klar, dass der `MopsoOne`-Algorithmus bei den ZDT-Funktionen viel empfindlicher auf Parameter-Änderung reagiert, als auf der `Deb`-Testfunktion. Um nun möglichst gute Einstellungen der wichtigsten Parameter `inertia`, `c1` , `c2` und `quantityParticle` für die eigentlichen Experimente zu finden, haben wir für alle ZDT-Funktionen das folgende Design erstellt. Dabei haben wir die wichtigsten Parameter `inertia`, `c1`, `c2` und `quantityParticle` mit drei Einstellungen variiert und den Parameter `quantityHypercube` mit den Werten 30 und 300 belegt. Da dieses Design bereits 182 Algorithmusaufrufe benötigt, und wir die Experimente bei 100000 Funktionsauswertungen ansetzen wollten, haben wir jede Parametereinstellung jeweils nur vier Mal mit verschiedenen Random-Seeds ausprobiert.

MopsoOne:

- `--a MopsoOne`
- `--t testfkt.ZDT`i [für i= 1, 2, 3, 6]
- `-fevals s 100000` (Anzahl der Funktionsauswertungen)
- `-g s 1` (spielt keine Rolle, da die Generationen durch `fevals` und `quantityParticle` im Algorithmus auf den richtigen Wert gesetzt werden)

- `-quantityParticle s 20 100 500`
- `-inertia s 0.3 0.5 0.7`
- `-c1 s 0.6 1.2 1.8`
- `-c2 s 0.6 1.2 1.8`
- `-quantityHypercube s 30 300`
- `-maxVelocity s 5.0`
- `-maxRepository s 200`
- `-r i 1 4`

Um die Einflüsse der Parameter zu erkennen, haben wir die Läufe mit dem Statistik-Programm *R* ausgewertet. Dazu haben wir mit dem `MeanMetricsPlotter` die Algorithmenausgaben in eine für *R* lesbare Datei umgewandelt. Aus dieser Datei haben wir zu allen variierten Parametern Boxplots und für die wichtigsten Parameter `inertia`, `c1`, `c2` und `quantityParticle` Interactionplots zwischen diesen erstellt. Die Abbildungen 59 und 60 zeigen diese für die `ZDT1`-Funktion.
Wie man den Boxplots entnehmen kann, scheinen die optimalen Werte für `inertia` bei $0,3$ bis $0,5$ für `c1` bei $0,6$ bis $1,2$ und für `c2` eindeutig bei $1,2$ zu liegen. Bei der Anzahl der Partikel sieht man, dass bei diesen Läufen eine große Population kleine Varianz bei den Metrikwerten garantiert, allerdings wurden die besten Werte mit kleineren Populationsgrößen erzielt. Bei der Betrachtung des Boxplots für den Parameter `quantityHypercube` ist zu erkennen, dass ein kleiner Wert einen minimal besseren Metrikwert erzielt hat. Bei der Betrachtung der Interactionplots fällt sofort auf, dass zwischen den Parametern `inertia` und `c2` die stärksten Interaktionen vorliegen. Mit den Werten $0,5$ für `inertia` und $1,2$ für `c2` wurde durchschnittlich über alle Läufe mit diesen Einstellungen ein Metrikwert von $6,5$ erzielt, während mit den Einstellungen von $0,7$ für `ineria` und $1,8$ für `c2` ein Metrikwert von 1 erzielt wurde. Die zweitwichtigste Interaktion findet man zwischen `c1` und `c2`. Bei hohen Einstellungen von $1,8$ bei beiden Parametern erzielt der Algorithmus im Durchschnitt einen Metrikwert von $1,5$, während bei dem Wert von $1,2$ für `c2` die Variation des Parameter `c1` sehr geringe Unterschiede der Güte bewirkt.
Bei der Betrachtung der Box- und Interactionplots der anderen ZDT-Funktionen sind diese Beobachtungen mit Ausnahme der `ZDT6` ebenfalls zu erkennen, wobei hier die durchschnittlichen Metrikwerte bei allen Kombinationen unter $1,5$ liegen.

Um die Parameter weiter zu untersuchen, haben wir einen weiteren Designaufruf für die ZDT-Funktionen gestartet, allerdings haben wir hierbei die Parameter `inertia` und `c2` mit jeweils fünf Parametereinstellungen in $0,1$-Schritten um die im letzten Versuch als beste vermuteten Werte variiert und die Parameter `c1` und `quantityParticle` weniger berücksichtigt. Außerdem haben wir die Anzahl der Funktionsauswertungen auf 50000 reduziert und die Anzahl der Random-Seeds auf zehn erhöht. Der Designaufruf sah folgendermaßen aus:

MopsoOne:

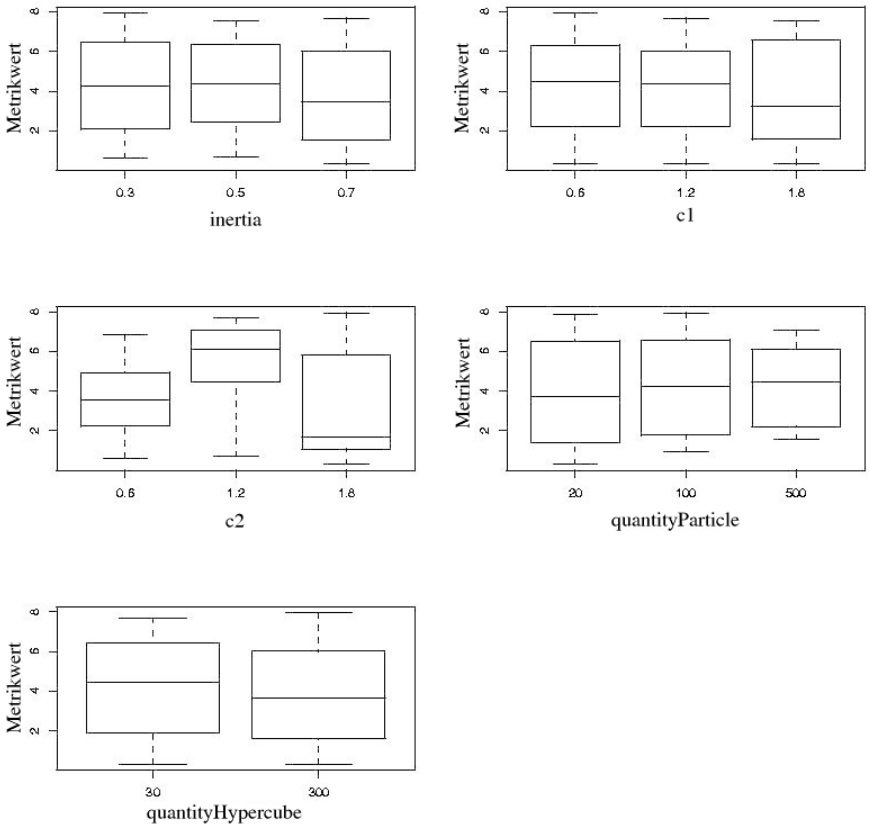

Abbildung 59: Boxplots für die Parameter `inertia`, `c1`, `c2`, `quantityParticle`, `quantityHypercube`. Es fällt auf, dass mit dem Wert von $1,2$ für `c2` die besten Ergebnisse erzielt wurden.

- `--a MopsoOne`
- `--t testfkt.ZDT`i [für i = 1,2,3,6]
- `-fevals s 50000`
- `-g s 1` (spielt keine Rolle, da die Generationen durch `fevals` und `quantityParticle` im Algorithmus auf den richtigen Wert gesetzt werden)
- `-quantityParticle s 20 100`
- `-inertia s 0.2 0.3 0.4 0.5 0.6`
- `-c1 s 0.8 1.0 1.2`
- `-c2 s 1.0 1.2 1.4 1.6 1.8`
- `-quantityHypercube s 150`

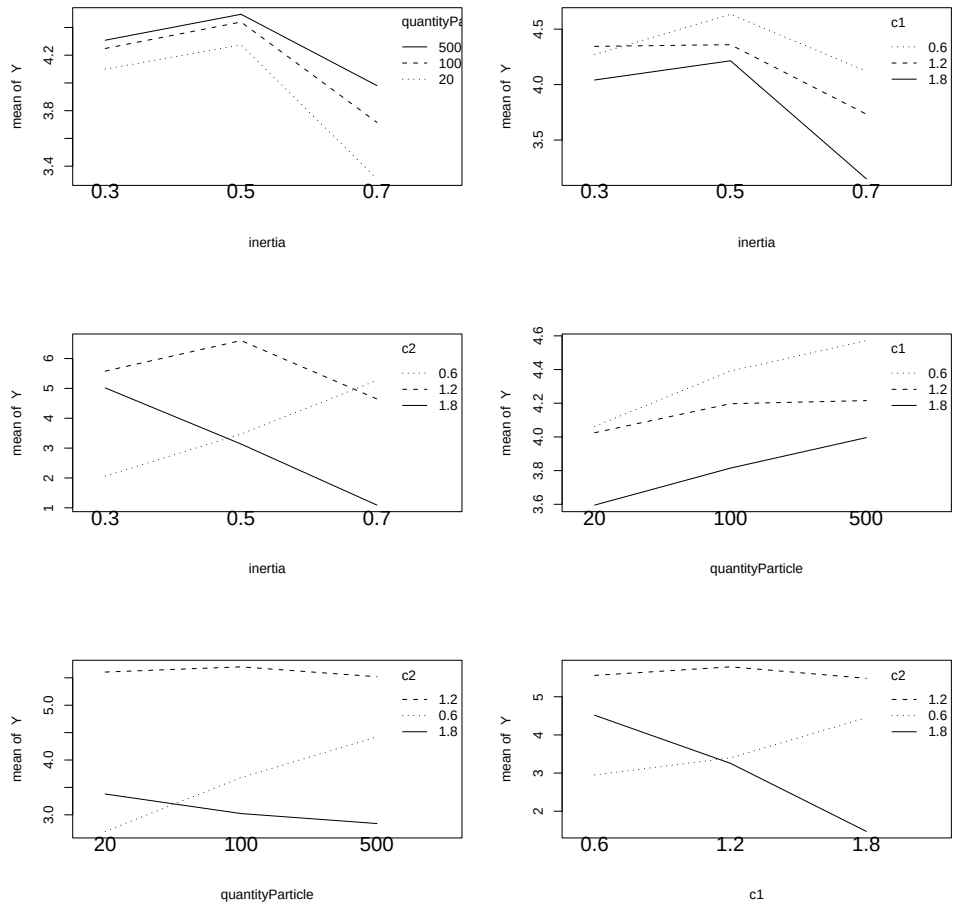

Abbildung 60: Interactionplot für die Parameter `inertia`, `c1`, `c2`, `quantityParticle`. Man erkennt die größten Wechselwirkungen zwischen `inertia` und `c2`; zwischen `inertia` und `quantityParticle` keine Interaktion zu erkennen

- `-maxVelocity s 5.0`
- `-maxRepository s 200`
- `-r i 1 10`

Wie im letzten Experiment haben wir diesmal wieder Boxplots und Interactionplots für die Analyse benutzt (siehe Abbildung 61 und 62).
Wie man auf den Boxplots (Abb. 61) erkennen kann, liegen die optimalen Werte für die `ZDT1`-Funktion bei diesem Experiment für `inertia` bei $0,3$ und $0,4$ und für `c2` bei $1,4$ und $1,6$. Der Einfluss der Parameter `c1` und `quantityParticle` scheint wieder sehr gering zu sein, der dazugehörige Boxplot bestätigt jedoch die Beobachtung aus dem vorherigen Experiment, dass eine größere Population die Stabilität des Algorithmus verbessert. Der Interactionplot zwischen `inertia` und `c2` (Abb. 62) legt

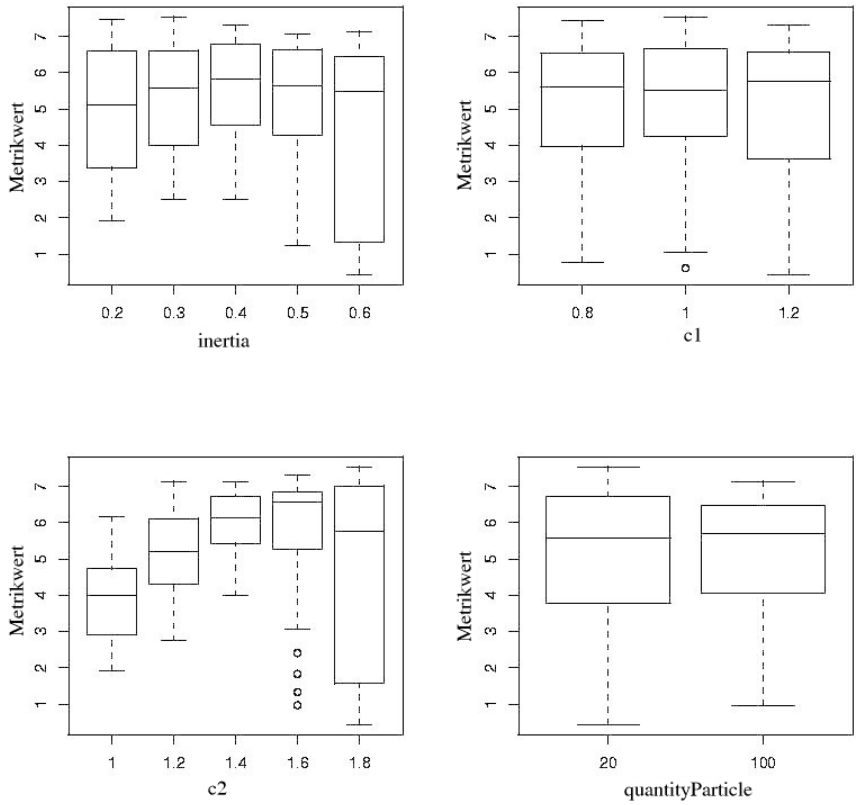

Abbildung 61: Boxplots für die Parameter `inertia`, `c1`, `c2`, `quantityParticle`. Der Wert $1,4$ für `c2` garantiert die kleinste Varianz; die besten Ergebnisse wurden aber mit einem höheren Wert erzielt.

wieder nahe, dass die richtige Einstellung dieser Parameter gute Ergebnisse garantieren sollten. Es ist sogar fast ein linearer Zusammenhang zwischen diesen Parametern zu erkennen. Wählt man nämlich die Parameter so, dass die Summe ungefähr zwei beträgt, liegt der durchschnittliche Metrikwert bei Variation aller anderer Parameter in den untersuchten Bereichen mindestens bei sechs. Der Verlauf der Kurven bei `c2`-Werten von $1,2$ und $1,0$ lässt vermuten, dass der optimale Wert für `inertia` höher als $0,6$ liegt. Siehe dazu Abb. 63.

7.3.3.4 Optimale Populationsgröße

Nachdem wir einigermaßen gute Einstellungen für `inertia` und `c`-Parameter gefunden haben, wollten wir untersuchen, wie die optimale Populationsgröße bei unterschiedlichen, guten Parametereinstellungen aussehen. Gibt es eine optimale Anzahl von Partikeln, die bei jeder Einstellung der anderen Parameter optimal ist, oder hängt die optimale Populationsgröße bei der selben Funktion von anderen Parametern ab? Um das zu untersuchen, haben wir vier unterschiedliche Parametereinstellungen

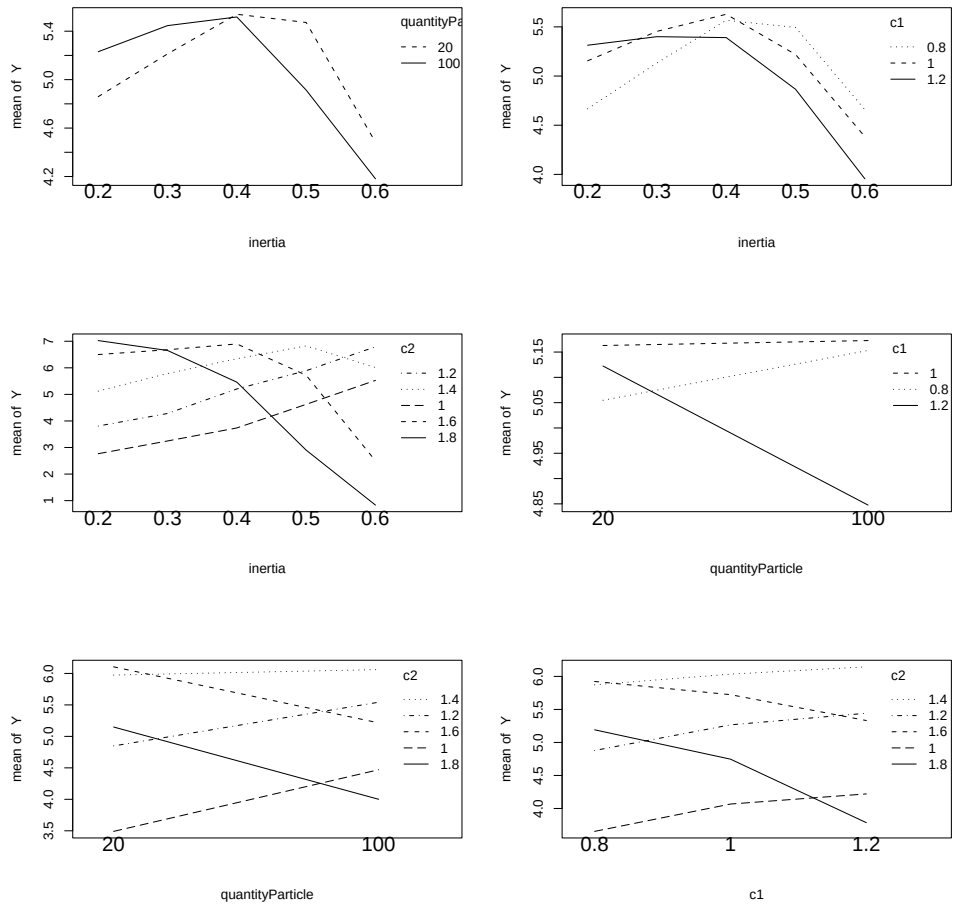

Abbildung 62: Interactionplot für die Parameter `inertia`, `c1`, `c2`, `quantityParticle`. Die größten Interaktionen sind zwischen `inertia` und `c2` zu erkennen.

gewählt und darauf durch Variation der Populationsgröße bei gleichbleibender Anzahl von Funktionsauswertungen die optimalen Populationsgrößen gesucht.

Erstes Experiment Um uns erstmals einen groben Überblick zu verschaffen, haben wir die Anzahl der Partikel sehr gestreut. Die Designaufrufe waren :

MopsoOne: Designparameter, die bei allen vier Versuchen gleich waren:

- `--t testfkt.ZDT1`
- `-fevals s 50000`
- `-g s 1` (spielt keine Rolle, da die Generationen durch `fevals` und `quantityParticle` im Algorithmus auf den richtigen Wert gesetzt werden)

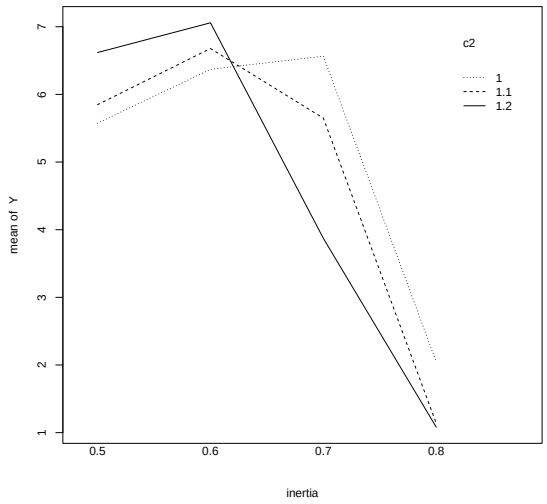

Abbildung 63: Interactionplot für die Parameter `inertia` und `c2`. Der beste inertia-Wert für `c2=1` liegt bei $0,7$.

- `-quantityParticle s 2 5 10 50 100 200 500 750 1000 5000 10000 25000`
- `-c1 s 1.0`
- `-quantityHypercube s 150`
- `-maxVelocity s 5.0`
- `-maxRepository s 200`
- `-r i 1 30`

Unterschiedliche Parameter :

- 1. Design: `inertia` = $0,2$; `c2` = $1,8$
- 2. Design: `inertia` = $0,4$; `c2` = $1,6$
- 3. Design: `inertia` = $0,6$; `c2` = $1,2$
- 4. Design: `inertia` = $0,7$; `c2` = $0,9$

Die Abbildungen 64 und 65 zeigen die gemittelten Metrikwerte zu verschiedenen Einstellungen der Populationsgröße, wobei die x-Achse in logarithmischer Skalierung dargestellt ist. Die Bilder deuten an, dass bei der ersten Einstellung mit `inertia`=$0,2$ und `c2`=$1,8$ die optimale Anzahl der Partikel zwischen 10 und 50 liegen müsste, bei den nächsten Einstellungen immer größer sein müsste, was auf einen Zusammenhang mit den Werten von `inertia` und `c2` deutet.

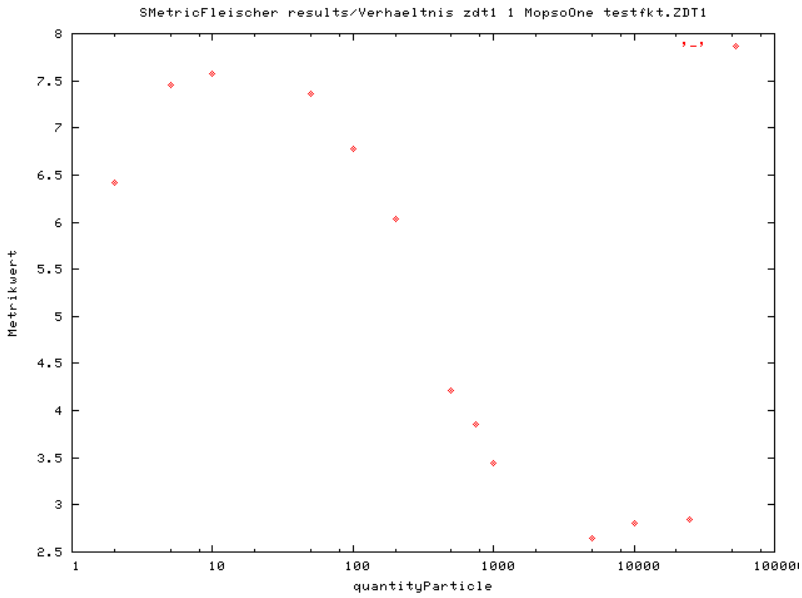

Abbildung 64: Metrikwerte zu Populationsgröße bei `inertia`=0.2 und `c2`=1.8. Der optimale Wert liegt zwischen zehn und 50 Partikeln.

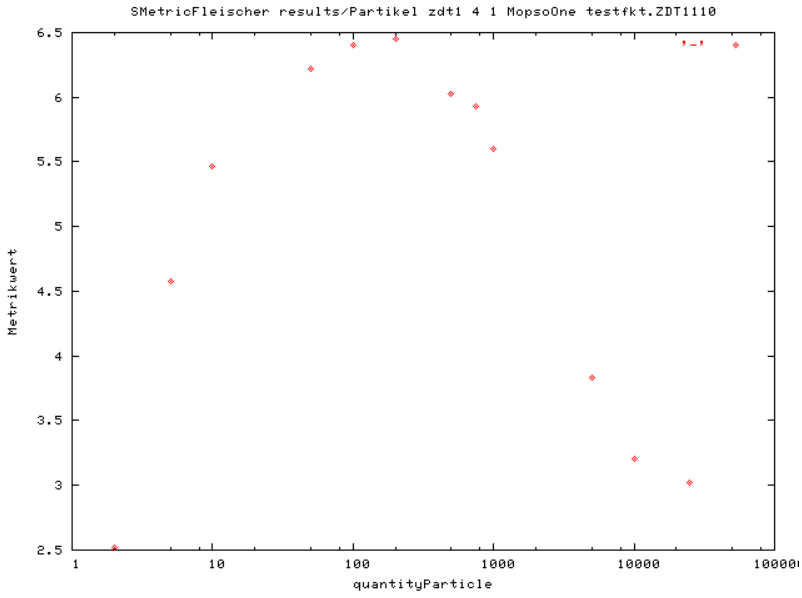

Abbildung 65: Metrikwerte zu Populationsgröße bei `inertia`=0.6 und `c2`=1.2. Der optimale Wert liegt zwischen 100 und 500 Partikeln.

Zweites Experiment Um diese Vermutung zu verifizieren, haben wir das erste und vierte Experiment mit 100 verschiedenen Random-Seeds und variierter Partikelanzahl in den Bereichen $[2; 157]$ und $[10; 400]$ gestartet, wobei wir jeweils in fünfer bzw. zehner- Schritten vorgegangen sind.

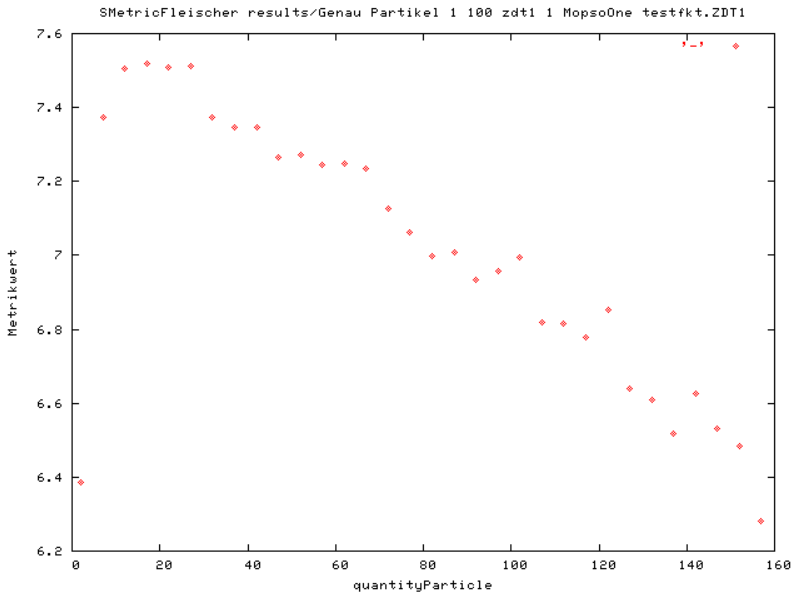

Abbildung 66: Metrikwerte zu Populationsgröße bei `inertia`=$0,2$ und `c2`=$1,8$. Die optimale Populationsgröße liegt zwischen 12 und 27.

Beobachtungen: Die Bilder 66 und 67 zeigen, dass es keine optimale Populationsgröße für eine bestimmte Funktion gibt, sondern dass diese mit anderen Parametereinstellungen interagieren. Bei dem ersten Experiment mit `inertia`=$0,2$ und `c2`=$1,8$ ist die optimale Anzahl der Partikel zwischen 10 und 40, bei der zweiten Parametereinstellung ist kein eindeutiges Ergebnis zu erkennen, obwohl wir hier mit einer großen Anzahl von 100 Random-Seed die Versuche durchgeführt haben. Bei den besten Einstellungen, die zwischen 50 und 300 liegen, ist keine Gesetzmäßigkeit zu erkennen (die durchschnittlichen Metrikwerte unterscheiden sich nur um Kleinigkeiten), allerdings sind die optimalen Einstellungen für die Anzahl der Partikel deutlich höher als bei der ersten Einstellung der Werte für `inertia` und `c2`. Ob die optimale Populationsgröße eher vom `inertia`-Wert oder `c2` abhängt, haben wir nicht mehr untersuchen können.

7.3.3.5 Einfluss der Populationsgröße auf die Robustheit

Aus den Algorithmen-Ausgaben dieser Experimente haben wir zusätzlich Boxplots erstellt, um die Auswirkung der Populationsgröße auf die Robustheit zu untersuchen. Die Abbildungen 68 und 69 widerlegen jedoch unsere Vermutung, dass eine große Anzahl von Partikeln wenige Ausreißer nach unten garantieren. Bei beiden Einstellungen haben die besten Einstellungen der Populationsgröße auch die kleinste Varianz der Metrikwerte.

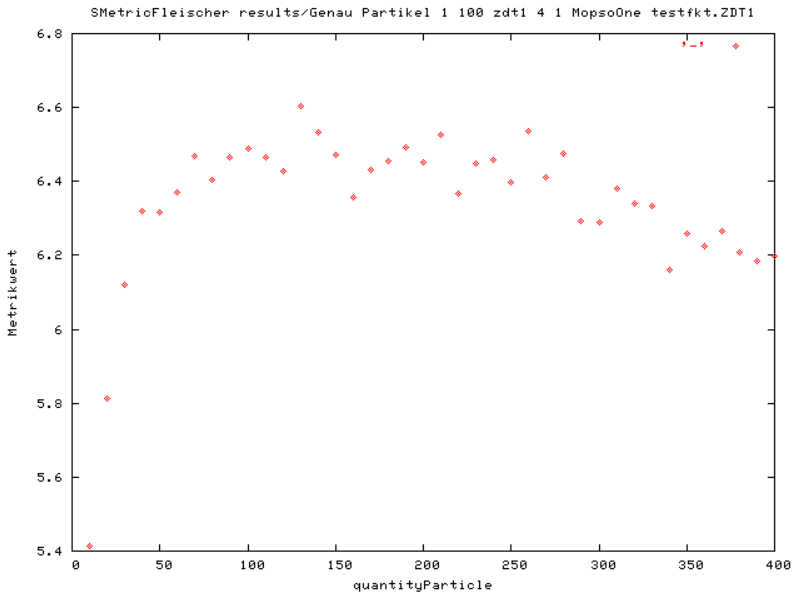

Abbildung 67: Metrikwerte zu Populationsgröße bei `inertia`=0.7 und `c2`=0.9. Der optimale Wert für die Populationsgröße ist 130. Die Ergebnisse unterscheiden sich im Bereich zwischen 80 und 270 jedoch nur gering.

7.3.3.6 Verhältnis zwischen der Populationsgröße und Anzahl der Funktionsauswertungen

Zum Abschluss wollten wir untersuchen, ob es ein optimales Verhältnis zwischen der Populationsgröße und der Anzahl der Funktionsauswertungen gibt. Dazu haben wir für dieselben Parametereinstellungen die Populationsgröße variiert und die Archive während der Laufzeit gespeichert, so dass wir zu verschiedenen Populationsgrößen die gemittelten Metrikwerte im Verlauf von 200000 Generationen darstellen konnten. Unser Design sah folgendermaßen aus:

MopsoOne:

- `--a MopsoOne`
- `--t testfkt.ZDT1`
- `-fevals s 200000`
- `-g s 1`
- `-quantityParticle s 4 8 20 60 120 200`
- `-inertia s 0.4`
- `-c1 s 1.0`
- `-c2 s 1.6`
- `-quantityHypercube s 150`

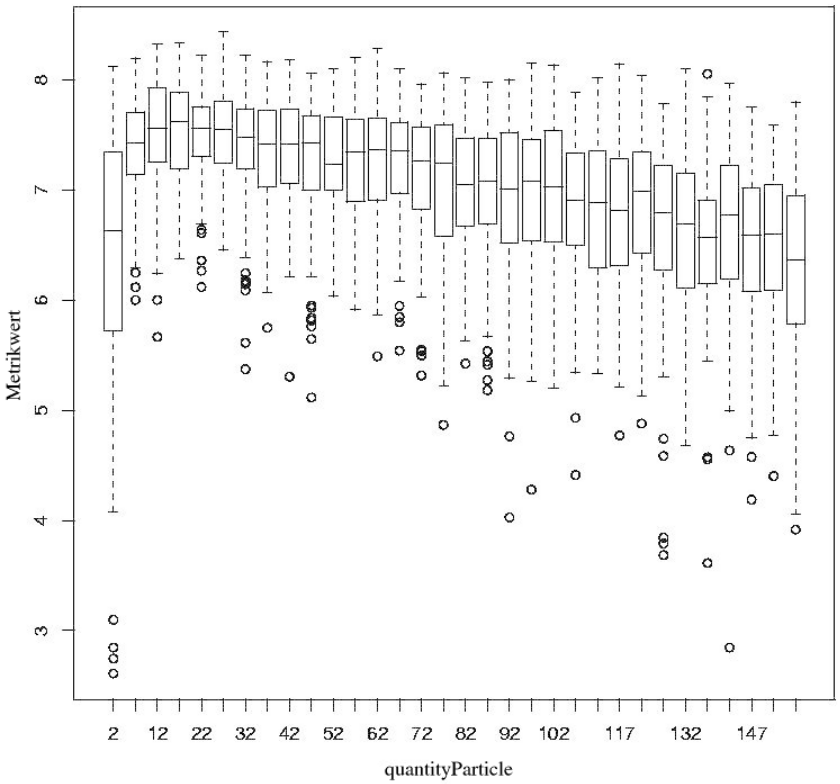

Abbildung 68: Boxplots für Populationsgröße bei `inertia`=0,2 und `c2`=1,8. Die kleinste Varianz ist bei besten Werten für Populationsgröße zu erkennen.

- `-maxVelocity s 5.0`
- `-maxRepository s 200`
- `-r i 50`

Die Abbildung 70 zeigt die gemittelten Verläufe des Algorithmus bei verschiedenen Populationsgrößen. Wie man sieht, ist der Lauf mit der kleinsten Population am Anfang deutlich besser, was daran liegt, dass er mehr Generationen hatten, um sich kontinuierlich zu verbessern und schon recht gute Ergebnisse zu liefern. Allerdings ist der Lauf mit vier Partikeln nach 200.000 Funktionsauswertungen auch der schlechteste und verbessert sich nach 40.000 Funktionsauswertungen kaum noch. Der Lauf mit 20 Partikeln ist nach ungefähr 30.000 Funktionsauswertungen der beste, wird aber noch vom Sechziger-Lauf abgefangen. Der Lauf mit der größten Anzahl von Partikeln ist erst nach etwa 120.000 Funktionsauswertungen der beste und scheint sich auch zum Ende hin am meisten zu verbessern. Insgesamt bestätigt dieses Experiment unsere Vermutung, dass bei steigender Anzahl der zur Verfügung stehenden Funktionsauswertungen eine größere Population zu wählen ist.

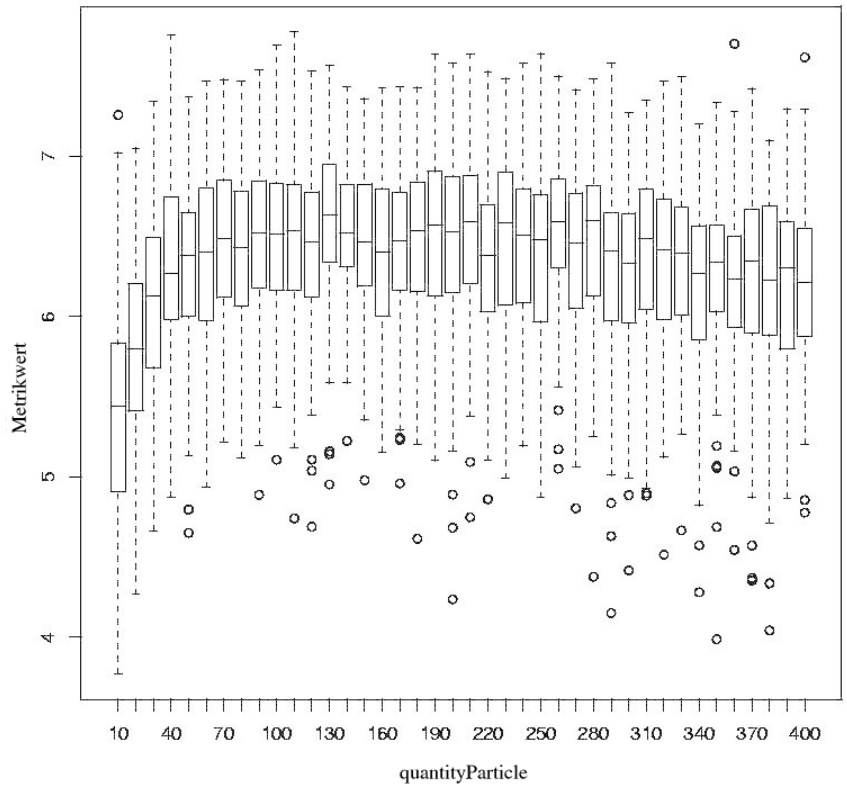

Abbildung 69: Boxplots für Populationsgröße bei `inertia`=0, 7 und `c2`=0, 9. Zwischen 80 und 400 sind keine großen Unterschiede bei der Varianz zu erkennen.

7.3.3.7 Fazit und Ausblick

In den obigen Experimenten konnten wir einen Einblick geben, wie stark sich die Parameter des `MopsoOne` beeinflussen, und wie schwierig es deswegen ist, die „optimalen“ Parameter zu finden. Außerdem wollten wir insbesondere die Populationsgröße untersuchen und einige Zusammenhänge herausstellen. Wir konnten zeigen, dass es nicht die optimale Populationsgröße für eine Testfunktion gibt, sondern diese auch von anderen Parametern abhängt. Die Vermutung, dass eine große Anzahl von Partikeln eine bessere Robustheit garantieren würde, konnte von uns nicht bestätigt, sogar an einem Beispiel widerlegt werden. Was das Verhältnis zwischen Population und Funktionsauswertungen angeht, konnten wir zumindest an der `ZDT1`-Funktion zeigen, dass bei einer größeren Zahl an Funktionsauswertungen prinzipiell eine höhere Zahl von Partikeln zu wählen ist. Ob sich unsere Beobachtungen auch an anderen Testfunktionen bestätigen, konnten wir aus Zeitgründen nicht untersuchen.

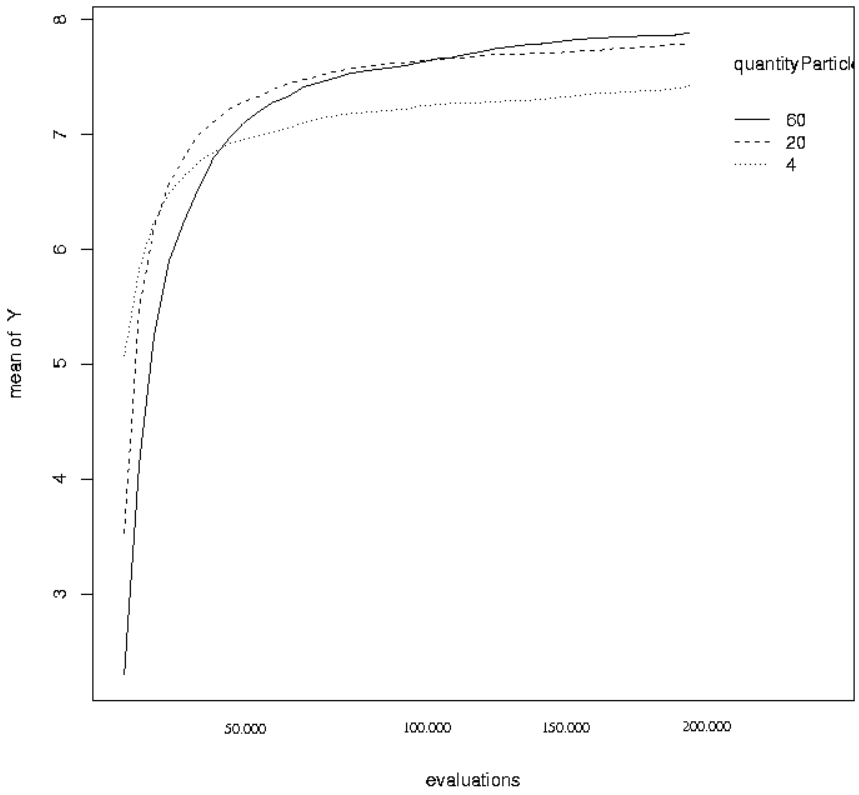

Abbildung 70: Verlauf bei verschiedenen Populationsgrößen auf der ZDT1-Testfunktion. Der Lauf mit der größten Population erreicht am Ende den besten Metrikwert.

7.3.4 Anzahl der Funktionsauswertungen

7.3.4.1 Motivation

Nach anfänglichen Überlegungen, in welchen Bereichen wir forschen wollten, fiel auf, dass der Begriff „Zeit“ eine wesentliche Rolle bei der Bewertung der Algorithmen spielt. Es kann sein, dass man schon nach wenigen „Zeiteinheiten“ gute Ergebnisse erzielt, oder aber man sehr lange warten muss, bis brauchbare Ergebnisse vorliegen. Diese Betrachtungsweise ist aber mit weiteren Fragestellungen verbunden. Wie definiert man eine „Zeiteinheit“? Sicherlich zählt hier auch der Begriff „Raum“ oder „Umgebung“ mit hinein, da Programme auf verschiedenen Rechnern schneller oder langsamer laufen. Auf welchem Problem arbeite ich? Welche Zeitgrenze wird zum Limit gesetzt in der man ein Ergebnis zu erzielen hat? Ein Beispiel wäre hier aus dem Bereich der Echtzeitsysteme zu nennen, wo man überwiegend schnell Ergebnisse benötigt. Es wäre fatal, wenn eine Steuerungssoftware in einem Flugzeug sehr lange für die Berechnung einer Notfallsequenz benötigen würde. Andererseits werden die Zeitgrenzen sicherlich größer, wenn man ein Bauprojekt zu planen hat. Aufbauend auf diesen Fragestellungen wollten wir eine Vorstellung entwickeln, wie man mit der „Zeit“ bei evolutionären Algorithmen umzugehen hat.

Schnell wurde aber klar, dass diese Frage im Rahmen der PG-Veranstaltung zu abstrakt ist und man sich zunächst auf konkretere Fragen einlassen sollte. Daher kamen wir zu einer spezielleren Frage, mit der wir untersuchten, wie sich verschiedene Funktionsaufrufe auf die verschiedenen Algorithmen auswirken. Welches Verhalten zeigen die Algorithmen, wenn man z. B. 100 oder 1000 Funktionsaufrufe festsetzt? Die zunächst erste Erwartung war, dass sie sich mit dem Anstieg der Funktionsaufrufe stetig verbessern. Im Verlauf der Untersuchungen kann man davon aber nicht immer ausgehen. Eine Strategie verschlechterte sich während des Verlaufs. Es besteht auch die Möglichkeit, dass ein Algorithmus am Anfang bessere Ergebnisse als ein anderer liefert, aber dann im Verlauf von ihm überholt wird.

7.3.4.2 Planung

Um nun Aussagen über die beschriebene Problematik machen zu können, wurden mehrere Experimente durchgeführt. Diese wurden folgendermaßen geplant:

Algorithmen: Zur Verfügung haben wir neun verschiedene Algorithmen, aus denen vier ausgewählt wurden. Es sollten zwei vermutlich schnellere und zwei langsamere ausgewählt werden. Hier ist „schnell“ und „langsam“, in Bezug darauf zu sehen, wie lange es vermutlich dauert bis brauchbare Ergebnisse vorliegen. Die schnelleren sind der `MopsoOne`, der `MueRhoLES` und die langsameren der `OnePlusOnePG` und der `BasisGA`. Im Verlauf der Untersuchungen stellte sich heraus, dass wenn wir „fair“ vergleichen und die Ergebnisse anschaulich darstellen wollen, man bis zu 40000 Funktionsauswertungen bei der `Deb` und der `ZDT1` Testfunktion benötigt. Erst bei dieser Anzahl kann man bei der `Deb` und der `ZDT1` das Verhalten sichtbar machen. Bei der `Schaffer` hingegen, kam man schon bei einer kleineren Anzahl zu Ergebnissen.
Es war nun im Vorfeld schon bekannt, dass der `BasisGA` eine uneffiziente

Laufzeit bzgl. einer Funktionsauwertung hat und es wurde klar, dass es zu aufwändig war, brauchbare Ergebnisse in einer absehbaren Zeit zu bekommen. Daher wurde er nicht mit in die Bewertung einbezogen.

Testfunktionen: Da wir für unsere Experimente nun brauchbare Testfunktionen benötigten, wurde unser Paket auf elf Testfunktionen erweitert, mit Ausnahme der zwei praktischen Probleme. Hieraus wurden die `Deb`, die `Schaffer`, und die `ZDT1` Funktion genommen.

Metrik: Die folgenden Ergebnisse sind mit der `SMetrikFleischer` berechnet worden.

Design- und Analysetools: Für die Experimente wurde das `Design`-Tool und für die Analyse der `MetricsPlotter`, der `MeanMetricsPlotter` und der `SurfacePlotter` verwendet.

Der nächste Schritt bestand darin, festzulegen, wie die Designaufrufe aussehen sollten. Möchte man fair vergleichen, so benötigt man zunächst ein einheitliches Vergleichskriterium, welches wir mit unserer Metrik haben. Darüber hinaus muss man für die jeweilige Testfunktion gute Parametereinstellungen der jeweiligen Algorithmen finden. Diese Problematik wurde bereits oben erläutert. Hat man diese nun gefunden, kann man die Funktionsauswertungen, also die Anzahl der Generationen oder die Anzahl der Partikel beim `MopsoOne`, variieren. Wie die Funktionsauswertungen verändert werden können, ist von Algorithmus zu Algorithmus unterschiedlich und hängt von dessen Einstellmöglichkeiten ab.

Als erstes wurde die `Deb` Funktion verwendet, von der wir schon Parametereinstellungen für den Mopso aus dem Artikel von Coello Coello [11] besaßen. Um aber nachvollziehen zu können, ob diese auch wirklich gut sind und um unseren Erfahrungsspeicher aufzufüllen, wurde zunächst folgender Designaufruf gestartet (diese folgenden Aufrufe sind mit der Design-Klasse direkt ausführbar):

MopsoOne:

- –a MopsoOne
- –t testfkt.Deb
- -inertia s 0.4
- -c1 i 1.0 1.5 step 0.1
- -c2 i 1.0 1.5 step 0.1
- -quantityHypercube i 20 60 step 10
- -g i 100 400 step 100
- -quantityParticle i 40 400 step 20
- -maxVelocity s 100

- -maxRepository i 100 900 step 100
- -randomSeed i 1 20

Für den `MueRhoLES` hatten wir keine Anhaltspunkte. Daher mußte auch hier ein umfangreiches Designs gestartet werden.
MueRhoLES:

- –a MueRhoLES
- –t testfkt.Deb
- -archive i 200 900 step 100
- -lambda i 100 250 step 50
- -mue i 15 25 step 5
- -tau1 s 0.1
- -tau2 s 0.1
- -g i 100 200 step 100
- -rho i 2 3 step 1
- -kappa i 3 9 step 3
- -r i 1 20

Aufgrund der wenigen Parameter beim `OnePlusOnePG` war das Design hier nicht so groß.
OnePlusOnePG:

- –a OnePlusOnePG
- –t testfkt.Deb
- -archive i 100 900 step 100
- -deviation i 1.0 5.0 step 1.0
- -g i 1000 40000 step 5000
- -r i 1 20

7.3.4.3 Ergebnisse

Die Daten, die durch die Designs produziert wurden, sind nun mit den verschiedenen Plottern analysiert worden. Dabei konnte man nur eine sehr grobe Vorstellung erhalten, wie die Parameter einzustellen sind, weil viele Läufe dicht beieinander liegende Metrikwerte erzeugten. Als nächstes wurden daher weitere Designs aufgerufen, bei denen einzelne Einstellungen spezieller behandelt wurden. Die Ergebnisse wurden dann hauptsächlich mit dem `MeanMetricsPlotter` analysiert. Im Folgenden wird gezeigt, welche Parameter bei den verschiedenen Algorithmen interessant waren und welche Designaufrufe dadurch besonders zu erwähnen sind.

MopsoOne: Der `MopsoOne` hat mit seinen neun Parametern die meisten der ausgewählten Algorithmen, und ist daher schwierig einzustellen. Einer der Hauptfaktoren ist sicherlich die Anzahl der Partikel. Anfängliche Untersuchungen ergaben bei der Debfunktion, dass sich eine Zahl von 40 Partikeln als vorteilhaft erwiest (siehe Abbildung 71). Wie genau das Verhältnis zwischen Generation und Partikel sein sollte, wurde in einer gesonderten Experimentierreihe untersucht.
Designaufrufe waren hier z. B.:
(Auswirkung der Partikel)

```
--t testfkt.Deb --a MopsoOne -quantityHypercube s 40
-c1 s 1.0 -c2 s 1.0
-g s 400 -quantityParticle i 20 80 step 10
-maxRepository s 200
-maxVelocity s 100 -Inertia s 0.4
-randomSeed i 1 30
```

(Auswirkung der Generationen)

```
--t testfkt.Deb --a MopsoOne -g i 100 1000 step 10
-quantityParticle s 40 -scalingInertia s 1
-maxInertia s 0.9 -minInertia s 0.4 -c1 s 1 -c2 s 1
-quantityHypercube s 40 -maxRepository s 600
-maxVelocity s 100
-randomSeed i 1 30
```

Durch diese und noch weiteren Experimentierreihen kam man zu den folgenden, bis dahin festgestellten guten Parameter:

```
-t testfkt.Deb -g 300 -quantityParticle 40
-scalingInertia 1 -maxInertia 0.9 -minInertia
0.4 -c1 1 -c2 1 -quantityHypercube 40
-maxRepository 600 -maxVelocity 100
```

MueRhoLES: Aus der Literatur war ersichtlich, dass der `MuRhoLES` mit einem Selektionsdruck von 1/7 und 1/10 (`mue/lambda`) gute Ergebnisse im Allgemeinen erzielt. Dies waren aber Angaben im einkriteriellen Fall. Diese Besonderheit wurde auch im mehrkriteriellen Fall getestet. Es gab aber keine besonderen

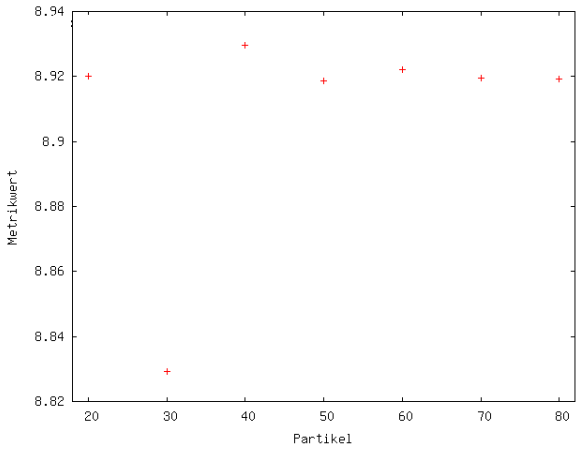

Abbildung 71: Auswirkung von verschiedenen Partikelanzahlen

Erscheinungen bei der `Deb` Funktion. Bei der `Schaffer` Funktion sieht man aber in den Abbildung 72, dass bei einem Selektionsdruck von 1/7 sich die Optimierung mehr auf die erste Front bezieht. Bei einem Druck von 1/10 bezieht sie sich auf die zweite. Über mehrere Generationen hinweg ergeben sich aber besser Gütewerte, wählt man einen Druck von 1/7.

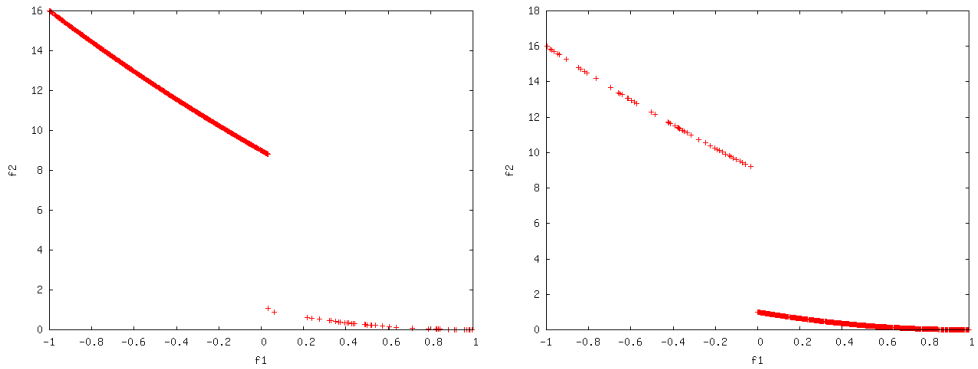

Abbildung 72: links Selektionsdruck 1/7 rechts Selektionsdruck 1/10

Designaufrufe waren hier z.B.:
(Auswirkung des Selektionsdrucks)

```
--t testfkt.Schaffer -g s 100
-mue i 10 15 step 5 -tau1 s 0.1
-tau2 s 0.1 -archive s 600 -lambda s 100
-rho s 3 -r i 1 30
-kappa s 9
```

(Variation von `kappa`)

```
--t testfkt.Deb --a MueRhoLES -g i 10 400 step 10
-mue s 15 -rho s 3 -lambda s 100
-kappa i 1 9 step 1
-tau1 s 0.1 -tau2 s 0.1
```

```
-archive s 600 -randomSeed i 1 30
```

Bekannt war auch, dass der Parameter kappa, der das Maximale Alter eines Individuums angibt, oft auf drei gesetzt wird. Unsere Experimente zeigten, dass neun ein vorteilhafter Wert ist. Dies wurde bei der `Deb` Funktion festgestellt. Die guten Parameter ergaben sich zu:

```
-t testfkt.Schaffer -g 10 -mue 15
-rho 3 -lambda 100 -kappa 9 -tau1 0.1
-tau2 0.1
-t testfkt.Deb -g 200 -tau 0.1
-tau2 0.1 -lamdba 200 -rho 3
-mue 15 -kappa 9
```

OnePlusOnePG: Der interessante Parameter, der hier einzustellen war, ist `deviation`, die Standardabweichung. Es wurden Experimente von 1 bis 3 durchgeführt, wodurch sich ein Wert von eins als gut herausstellte. Man kann davon ausgehen, dass der `OnePlusOnePg` bei einer erhöhten Anzahl von Generationen zu besseren Ergebnissen kommt, was sich auch bestätigte (siehe unten).

Design:

```
--t testfkt.Deb --a OnePlusOnePG
-deviation i 1.0 3.0
-g i 1000 40000 step 1000
-r i 1 30
```

Gute Einstellungen :

```
-t testfkt.Deb -g 40000 -deviation 1.0
-t testfkt.Schaffer -g 3000 -deviation 1.0
```

Der nächste Schritt bestand nun darin die Anzahl der Funktionsauswertungen fest zu legen, die von 1000 bis 40000 fest gesetzt wurden, variiert durch den Generationsparameter. Dann wurde immer nach 1000 Evaluationen ausgewertet und der jeweilige Algorithmus neu gestartet. Es gab also keinen Live-Output und zu jedem Experiment wurden 30 verschiedene Läufe bzgl. der Zufallszahlen durchgeführt. Über die dann erzeugten 30 Metrikwerte wurde ein gemittelter Wert gebildet. Bei der Abbildung 73 sieht man schließlich, wie sich die Algorithmen verhalten. Hier wurde die Testfunktion `Deb` benutzt. Der gemeinsamer Referenzpunkt lag bei $(1,994489; 11,965702)$. Bei wenigen Funktionsauswertungen zeigt der `MopsoOne` ein deutlich besseres Ergebnis als die anderen zwei, wobei `MueRhoLES` und `OnePlusOnePG` dicht beieinander liegen. Im Verlauf der Iterationen behauptet der `MopsoOne` seine Spitzenposition, während der `MueRhoLES` im Mittelfeld bleibt. Nach einer kurzen Verschlechterungsphase verbessert er sich wieder und gelangt nach 20000 Funktionsauswertungen auf den vorher erreichten Gütewert. Er ist auch der einzige, der sich während des Verlaufs verschlechtert. Seine Kurve steigt nicht kontinuierlich. Dies liegt darin, dass bei einer hohen Generationsanzahl das Archiv, welches die besten nicht dominierten Lösungen enthält, komplett gefüllt ist. Treffen nun weitere Lösungen ein, werden die ersten entfernt, wobei man hier vermutlich eine Verbesserung erzielen würde, wenn die Lösungen zufällig entfernt werden würden. Der `OnePlusOnePG` verbessert sich ste-

tig, wobei der MopsoOne nach 14000 nicht über einen Gütewert von $23,99$ hinaus kommt, wobei man am Ende bei 40000 Funktionsauswertungen sieht, dass er bis zu diesem Zeitpunkt der „Beste“ bleibt. Erwähnenswert ist noch, dass der MueRhoLES zum Ende hin nicht weiter optimiert und vom OnePlusOnePG überholt wird (siehe Abb. 74).

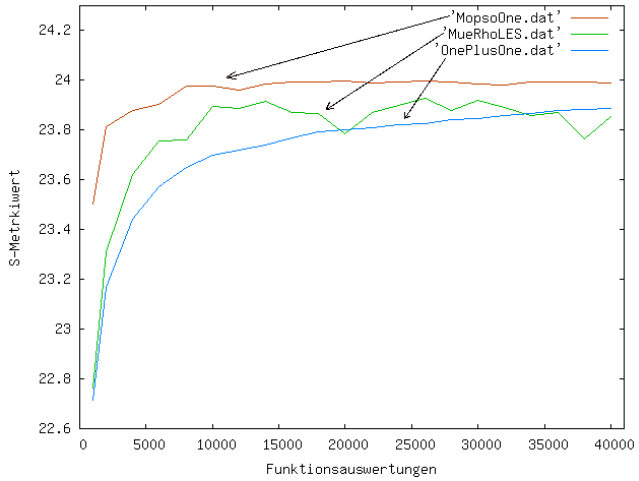

Abbildung 73: Verlauf der Funktionsauswertungen bei Deb

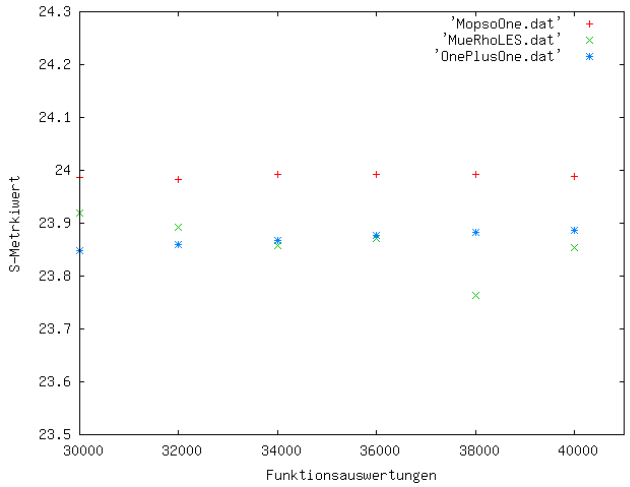

Abbildung 74: Endverlauf der Funktionsauswertungen bei Deb

Als nächstes galt es nun die Frage zu klären, ob sich dieses Bild auch bei anderen Testfunktionen ergibt, wenn die Paramter gleich bleiben. Bleibt der MopsoOne der Beste oder wird er von den anderen überholt oder produziert er sogar von Anfang an schlechtere Ergebnisse? Bei den daraufhin folgenden Experimenten wurden die Parametereinstellungen auf der ZDT1 für den MopsoOne und den MueRhoLES nicht verändert. Bei der Schaffer wurde dann noch ein weiteres Paramter-Tuning durchgeführt. Möchte man ein absolut präzises Bild bekommen, ab wann welcher Algorithmus am besten abschneidet, sind sicherlich weitere genauere Experimente bzgl. guten

Parameter notwendig. Denn wie bereits gesagt, sind die optimalen Paramter bei verschiedenen Testfunktionen unterschiedlich. Präzise statistische Versuchsauswertungen wären dazu notwendig, wie wir sie beim MopsoOne durchgeführt haben (siehe Kapitel 7.3.2 und 7.3.3). Abbildung 75 zeigt nun die Ergebnisse auf der ZDT1. Der Referenzpunkt lag hier bei $(1,999979; 7,341294)$. Was vor den Experimenten als Vermutung geäußert wurde, dass nämlich ein Algorithmus den anderen während der Optimierung überholen könnte, tritt hier nun ein. Der OnePlusOnePG kam hier nicht mit in die Bewertung, weil er zu schlechte Ergebnisse lieferte und selbst bei einer sehr hohen Anzahl von Iterationen nicht an die Pareto-Front heran kam. Am Start gewinnt der MopsoOne nun deutlich gegenüber dem MueRhoLES, welcher sich auch erst nach 7000 Funktionsauswertungen verbessert. Ab 19000 schneiden sich nun die beiden Läufe und der MueRhoLES übernimmt die Spitze. Ab hier wird sein Verhalten aber auch undurchsichtiger. Mal optimiert er weiter; mal nicht, wobei der MopsoOne sich auch verbessert, dies aber nicht so „sprunghaft“.

Die Schafferfunktion ist eine der einfacheren mehrkriteriellen Testfunktionen. Sie hat,

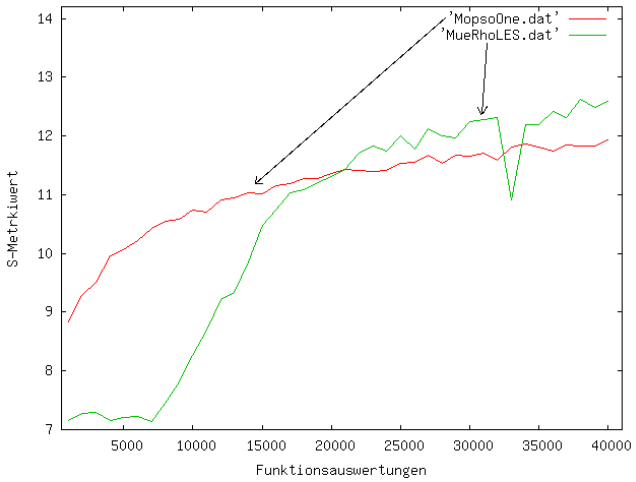

Abbildung 75: Verlauf der Funktionsauswertungen bei ZDT1

wie auch die Debfunktion, eine geteilte Pareto-Front. Wurden die Experimente mit den selben Einstellungen wie für die Deb benutzt, kam es zu Verschlechterung beim MopsoOne und beim MueRhoLES. Es macht nun auch keinen Sinn, die beiden Algorithmen mit 40000 Funktionsauswertungen auf einer solchen Funktion laufen zu lassen, da sie schon recht früh sehr gute Ergebnisse zeigen. Bei 2000 erkennt man beim MueRhoLES schon eine hohe Abdeckung der Pareto-Front (siehe Abb. 76).
Werden große Evaluationen benutzt, decken die Algorithmen die Teilfronten nicht gleichmäßig ab (siehe Abb. 77). Hier sieht man, dass die untere Teilfront beim MueRhoLES stärker abgedeckt wird, wobei beim MopsoOne die obere Front sogar „aufbricht“ und nicht mehr gleichmäßig abgedeckt wird. Dies erklärt auch, warum beide bei einer größeren Anzahl von Funktionsauswertungen, schlechtere Ergebnisse zeigen (siehe Abbildung 80). Es war nun auch zu vermuten, dass der OnePlusOnePG gute Ergebnisse zeigt, was sich auch bestätigte. Die Abbildung 78 zeigen die Fronten einmal bei 500 und 4000 Funktionsauswertungen.

Wer erreicht nun aber hier ab wann den besten Gütewert? Aufgrund der vorherigen

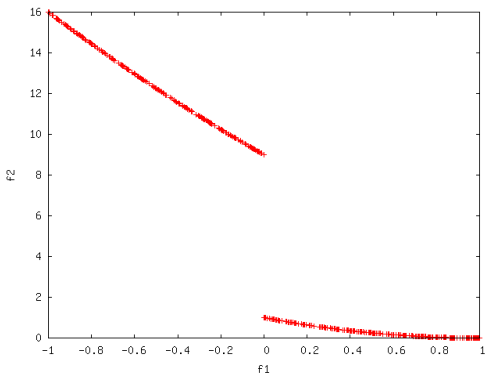

Abbildung 76: Pareto-Front produziert vom `MueRhoLES` bei 2000 Funktionsauswertungen bei der Schafferfunktion

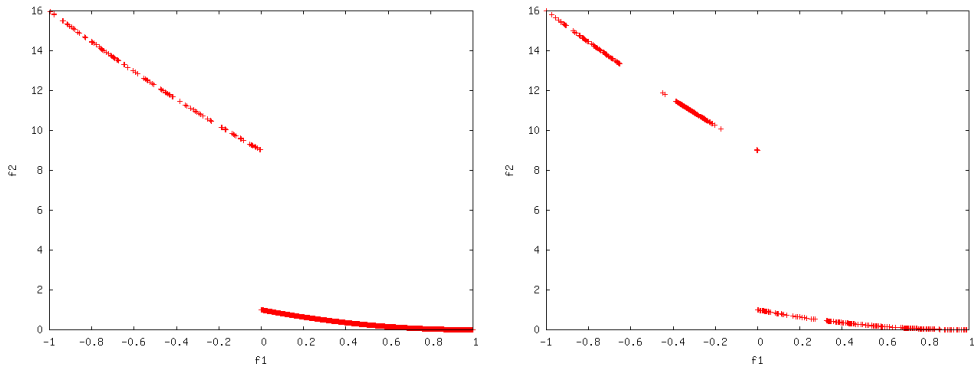

Abbildung 77: Die linke Abbildung zeigt die Pareto-Front vom `MueRhoLES` bei 20000 und die rechte zeigt den `MopsoOne` bei 40000 Evaluationen

Experimente, die zeigten, dass man schon recht früh zu guten Ergebnissen kam, legte man ein Intervall von 200 bis 3000 Evaluationen fest. Abbildung 79 und 80 zeigen den Verlauf. Der Referenzpunkt lag bei $(2,086976; 17,938794)$. Der `MopsoOne` gewinnt am Anfang und lässt beide wiederum hinter sich. Danach pendeln sich alle zwischen $40,9$ und $41,1$ ein, wobei der `MueRhoLES` auch hier sehr viele Ausreißer zeigt. Der `OnePlusOnePG` verbessert sich im Laufe der Optimierung monoton und liefert nach 3000 Auswertungen sogar den besten Metrikwert.
Als Fazit kann man sagen, dass bei einfachen Funktionen der `OnePlusOnePG` auch eine geeignete Wahl ist. Bedenken muss man, dass die anderen beiden eine komplexere Strategie benutzen und vermutlich eine längere Zeitspanne für ihre internen Berechnungen benötigen. Bei komplexeren Testfunktionen, wie `ZDT1` versagt er aber vollkommen.

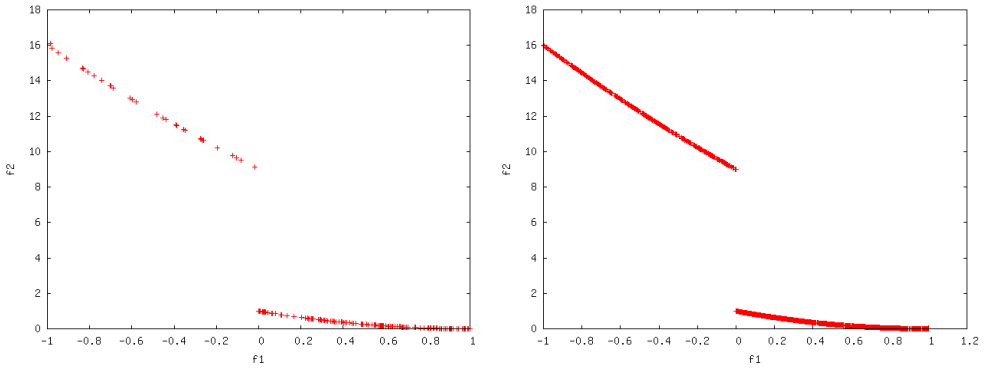

Abbildung 78: Pareto-Front produziert vom `OnePlusOnePG` bei 500 und 4000 Evaluationen

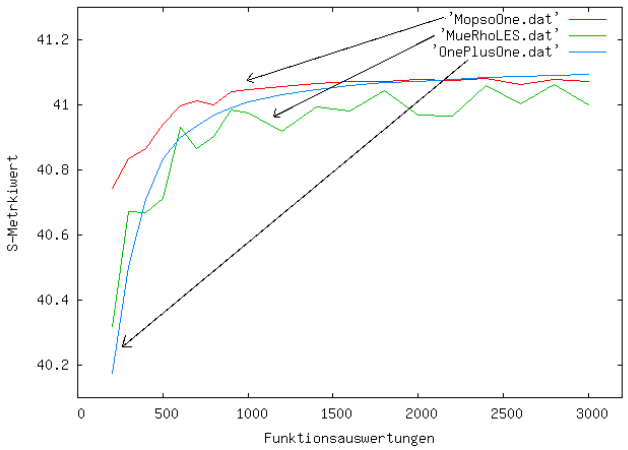

Abbildung 79: Verlauf der Funktionsauswertungen bei Schaffer

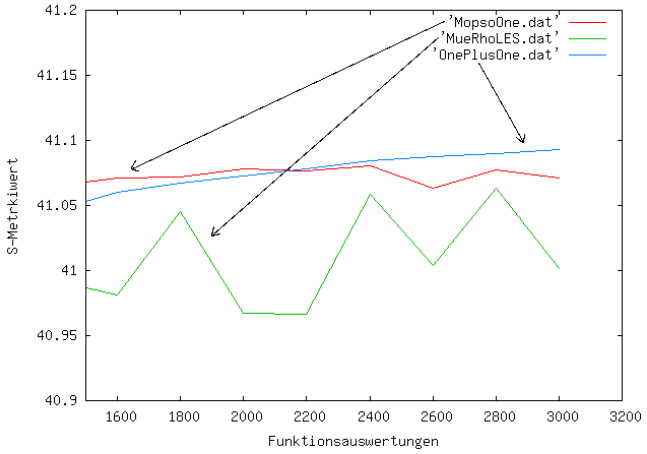

Abbildung 80: Endverlauf der Funktionsauswertungen bei Schaffer

7.3.4.4 Fazit

Ist nun die Frage, wie sich verschiedene Funktionsaufrufe auf die verschiedenen Algorithmen auswirken, eindeutig geklärt? So einfach die Frage klingt, um so schwerer ist es eine genauere Antwort darauf zu finden. Auf den benutzten Testfunktionen kann nun gesagt werden: Ab hier ist der `MopsoOne` der `MueRhoLES` oder der `OnePlusOnePG` besser. Aber gilt dieses für jedes Problem? Mit Sicherheit nicht. Man muss immer die Rahmenbedingungen betrachten, unter denen die Experimente durchgeführt wurden. Was man sagen kann ist, dass sich die Algorithmen im Verlauf ihrer Iterationen verbessern, wenn sie optimal eingestellt sind und die Ergebnisse richtig interpretiert werden. Laut unseren Ergebnissen, verbessern sie sich aber ab einer gewissen Grenze nicht mehr. Werden undurchdachte Funktionsauswertungen benutzt, kann es zu falschen Aussagen kommen und ein Algorithmus als „schlecht“ abgestempelt werden. Für die Praxis gilt, dass wenn das Problem, auf dem optimiert wird, leicht ist, man mit dem `OnePlusOnePG` schnell zu guten Ergebnissen kommt. Wird das Problem schwieriger (`Deb`, `ZDT1`) so sollte man zu anderen Algorithmen mit komplexeren Strategien wechseln.

7.3.5 Fazit der Robustheitsanalysen und Ausblick

Im zweiten Semester der PG-Veranstaltung hat die Robustheitsgruppe versucht, drei grundlegende Fragen bzgl. der Robustheit von mehrkriteriellen evolutionären Algorithmen zu beantworten. Dabei sind tiefere Einsichten zur Verhaltensweise der Optimiermethoden gewonnen worden. Bei unterschiedlichen Eingaben, also verschiedenen Parametereinstellungen, kam es teilweise zu massiven Änderungen der Ergebnisse. Wurden andere Testfunktionen benutzt, so war es auch notwendig, andere Einstellungen zu wählen, um sich der Pareto-Front zu nähern. Wie die Parameter einzustellen sind, ist also ebenso ein Optimierungsproblem, wie die eigentliche ursprüngliche Suche nach dem Maximum bzw. Minimum selbst. Die Gruppe hat mit statistischen Analysemethoden gearbeitet und festgestellt, wie aufwendig, aber auch sehr interessant diese Suche ist. Möchte man nicht auf vorgegebene Einstellungen zurückgreifen, kann man beliebig viel Zeit investieren um „gute“ Einstellungen zu finden. Es stellt sich die Frage, ob sich dieser Aufwand auch lohnt. Für theoretische Untersuchungen ist dies sicherlich erforderlich, in der Praxis kommt es auf die Ausgangssituation an. Haben wir ein schwieriges Problem, bei dem keine Informationen über die Struktur vorliegen, gibt man sich sicher mit einer guten Lösung zufrieden und sucht nicht weiter nach der absoluten. Ist diese Lösung auch robust, d. h. sind bei kleinen Änderungen der Einstellungen keine Instabilitäten in den Lösungen vorhanden, ist dies sicherlich speziell im Ingenieurbereich sehr zufriedenstellend, da die Stabilität des Systems vielleicht damit auch größer ist. Hier gibt es sicherlich weiteren Forschungbedarf bzgl. der Frage, ob robuste Lösungen der Optimierstrategie auch stabile Lösung des zugrundeliegenden Systems hervorrufen.
Für die Theorie, wie auch für die Praxis, ist die Frage, ab wann welcher Algorithmus welche Lösung produziert, interessant. Kommt man in endlicher Zeit zu einer Lösung? Braucht man 1000 oder 50000 Iterationen, um das Optimum zu erreichen? Welcher Algorithmus produziert in einer angemessenen Zeit brauchbare Lösungen? Unsere Ergebnisse zeigen, dass diese Frage auch nicht so leicht zu beantworten ist.

Man kann nicht sagen, dass, wenn einer am Anfang der Suche bessere Ergebnisse als ein anderer produziert, dies auch im weiteren Verlauf der Suche so ist. Durchaus ist es möglich, dass eine am Anfang als schlecht eingestufte Strategie, sich nach mehreren Durchläufen als besser herausstellt. Um zu einer allgemeineren Aussage zu kommen, sind hier weitere Möglichkeiten offen, indem man die Algorithmen auf mehreren Testfunktionen benutzt, um sagen zu können: „Auf schwierigen Problemen lohnt es sich, ab dann mit diesem Algorithmus zu arbeiten“. Ist das Problem leicht, so reicht eine einfachere Strategie. Wir können sagen, dass der `MopsoOne` bei schweren Testfunktionen gute Ergebnisse liefert. Diese sind auch robust, geht man von einer stetigen Verbesserung der Lösung aus. Der `MueRhoLES` hingegen verbessert sich auch, doch geschieht dies ziemlich sprunghaft. Wird er „falsch“ eingestellt und werden undurchdachte Funktionsauswertungen benutzt, kann es zu falschen Aussagen bzgl. seiner Güte kommen. Bei einfachen Problemen reicht es aber sicherlich aus, mit dem `OnePlusOnePG` zu arbeiten. Die Aussagen über gute Paramtereinstellungen kamen überwiegend durch die Betrachtung des `MopsoOne`. Die erläuterten Ergebnisse bieten einen Anhaltspunkt, wie der Algorithmus einzustellen ist und welche Analysemöglichkeiten es gibt, um die gewonnenen Ergebnisse zu interpretieren. Es wäre weiterhin interessant, wie sich diese Problematik bei anderen Algorithmen verhält. Der `MueRhoLES` bietet hier mit seinen vielen Parametern großes Potential. Beim `OnePlusOneOG` würde man sicher schnell zu einer genaueren Aussage kommen. Abschließend kann man sagen, dass die Erforschung der Algorithmen im Rahmen der Gruppe sehr viel Spaß gemacht hat, da es auch oft zu unerwarteten Situationen kam, die aber letztlich das Verständnis der Funktionsweise bestärkten. Die Beschäftigung mit diesem Thema bietet weiterhin Spielraum und lässt noch einige Fragen offen.

7.4 Simulatoren

7.4.1 Temperierbohrung

7.4.1.1 Designs

Nachdem der Simulator für die Temperierbohrung in *NObELTJE* eingebunden wurde und damit benutzt werden konnte, wurden Design-Aufrufe erstellt. Diese Design-Aufrufe sollten dazu dienen, gute Parameter für das Temperierbohrungsproblem zu liefern. Folgende Designs wurden mit dem Evolver auf der Basis von Erfahrungswerten gestartet:

- Der `MueRhoLES` wurde mit den Parametern `g` 100, `mue` 10, `lambda` 70, `rho` 2, 5, 10, `tau 1` 0.1 und 0.2, `archive` 50 und 100, `kappa` 0, 1 und 5, sowie mit 3, 4 und 5 Bohrungen gestartet. Zudem wurden der Algorithmus statt mit `mue` 10 und `lambda` 70 auch mit der Kombinationen `mue` 25 und `lambda` 125 sowie `mue` 50 und `lambda` 350 gestartet.

- Die Läufe des `BasisPSO` wurden mit den Parametern `c1` von 0 bis 2 in Schrittweite 0.2, `c2` analog gestartet. `Inertia` wurde in Schrittweite 0.2 im Bereich 0.2 bis 1 laufen gelassen, `g` von 20 bis 100 in Schrittweite 20, die Zyklen-Anzahl (Parameter `cycles`) von 100 bis 500 in Schrittweite 100 und die Archivgröße wurde auf 1000 gesetzt. Dieses Design wurde nur für 5 Bohrungen durchgeführt.

- Der `MopsoOne` wurde mit den Parametern `g` von 100 bis 500 in 100er Schritten laufen gelassen, zudem noch `quantityParticle` von 50 bis 100 in Schrittweite 1, `c1` und `c2` von 1 bis 5 in Schrittweite 1, `scalingInertia` 1, `maxInertia` 0.9, sowie mit drei verschiedenen Randomseeds. Ebenso wurde dieses Design für 3, 4 und 5 Bohrungen laufen gelassen.

- Die Parameter für das Räuber-Beute Design waren `g` von 200 bis 10000 in Schrittweite 200, die Mutationsschrittweite von 0.5 bis 5 in 0.2er Schritten. Dieses Design lief für 3, 4 und 5 Bohrungen druch. Zudem wurde die Archivgröße so gewählt, dass der Wert der Gittergröße entsprach.

- Der recht einfache `OnePlusOnePG` Algorithmus wurde mit einem Design aufgerufen, bei dem die Parameter Standardabweichung der Mutation von 0 bis 5 in Schrittweite 0.2, die Archivgröße von 10 bis 1000 in Schrittweite 20 und die Generationsanzahl von 10 bis 1000 in Schrittweite 20 gewählt wurden. Dieses wurde für 2, 3, 4 und 5 Bohrungen gemacht.

Die Laufzeit der vorgenannten Aufrufe war teilweise sehr hoch. Zudem kam erschwerend hinzu, dass der Simulator nur unter Windows läuft, sodass diese Aufrufe nicht im Batch-System des Lehrstuhls 11 bearbeitet werden konnten.
Nach dem Durchlauf dieser Designs mussten die Resultate noch nachbearbeitet werden. Zunächst wurden mit Hilfe der Klasse `FilterAndDelete` die ungültigen Polygonzüge aus den Ergebnissen entfernt, sodass di Restriktionen erfüllt wurden. Anschließend wurde das 12-dimensionale Problem auf drei Dimensionen heruntergebrochen, indem nur die zweite, die vierte und die sechste Spalte des Resultatvektors (Inhalt siehe Tabelle 4) betrachtet wurde. Da die S-Metrik nach Fleischer die einzige im

Projekt implementierte Metrik ist, die mit 12 Dimensionen umgehen kann, diese aber eine zu lange Laufzeit hatte, musste die Komplexität an dieser Stelle reduziert werden. Alleine eines von den oben genannten Designs hatte unter Beibehaltung der Dimensionstiefe eine Laufzeit von ca. 35 Tagen.
Um nun zu guten Parametereinstellungen zu gelangen, wurden die bearbeiteten Resultate mit der S-Metrik nach Fleischer bewertet. Die Parameter, die auf der S-Metrik die besten Werte erreicht haben, wurden als gute Parameter für das Temperierbohrungsproblem bezeichnet und mit diesen auch die weiteren Testläufe gestartet.
Ziel war es zu sehen, wie gut die bereits implementierten Algorithmen reale Problemen lösen. Da, im Gegensatz zu den Testfunktionen, die genauen Paretofronten unbekannt sind, war dieses Problem von besonderem Interesse. Auf den Testfunktionen kann die Güte eines Algorithmus gemessen werden, indem Ergebnisse mit der wirklichen Paretofront verglichen werden. Bei dem Simulator hingegen werden nur die Algorithmen untereinander verglichen. Ein Vergleich könnte bewerten, wie schnell die Algorithmen einen bestimmten Metrikwert erreichen. Eine andere Möglichkeit ist, die maximal erreichten Metrikwerte zu vergleichen. Es existieren viele weitere Methoden, die Güte von Algorithmen zu vergleichen, die hier aber nicht weiter aufgeführt werden sollen.
Weiter wurden mit den gefundenen guten Parametereinstellungen neue Testläufe durchgeführt, die nach einer vordefinierten Anzahl von Bewertungen das Archiv in einer Datei zwischenspeicherten. Dadurch wurde der Verlauf der Algorithmen sichtbar und damit auch feststellbar, welcher Algorithmus sich wie schnell einem bestimmten Metrikwert annähert.
Für die evaluationsbedingte Ausgabe wurden die Designs mit den gefundenen guten Parametern gestartet. Dieses sind die Parameter, die in der Tabelle 12 aufgelistet wurden. Zudem waren die Experimente auf fünf Bohrungen beschränkt, da mehr Bohrungen eine höhere Komplexität des Problems darstellen, die damit schwerer zu approximieren sind. Die Metrikwerte waren besser, wenn nur mit drei Bohrungen gearbeitet wurde. Da die Algorithmen jedoch auch dann gute Werte liefern sollten, wenn die Komplexität höher ist, wurden für diese Untersuchung zunächst nur fünf Bohrungen betrachtet.

Tabelle 12: Gute Parametereinstellugen für den Evolver

Algorithmus	Parameter
OnePlusOne	standardDeviation 3, g 1000
BasisPSO	inertia 1, c2 1, g 90, c1 1.5
MueRhoLES	tau2, 0.2, g 100, lambda 175, tau1 0.2, rho 2, mue 25 kappa 0
MopsoOne	minInertia 0.4, c2 3, g 100, quantityParticle 80, c1 1, scalingInertia 1, maxInertia 0.9
BasisGA	mprob 0.03, rprob 0.9, psize 1000

Ein Beispiel, wie ein Design nach durchgeführter Metrikwert-Berechnung aussieht, ist unter Abbildung 81 zu sehen. Für diese Grafik wurde ein *Gnuplot*-Aufruf ausgeführt, der auf der y-Achse die erreichten Metrikwerte nach der S-Metrik von Fleischer zeigt und auf der x-Achse die verschiedenen Läufe, die durch verschiedene Parametereinstellungen gekennzeichnet waren. Diese Grafik ist durch den Aufruf des

`MeanMetricPlotters` entstanden. Da aufgrund von Platzmangel auf der x-Achse nicht alle Parameter verzeichnet sind, die zu einem Lauf gehören, schreibt der `MeanMetricPlotter` diese noch einmal in eine separate Textdatei. In dieser Textdatei steht dann ebenfalls der erreichte Metrikwert und damit auch der maximal erreichte Metrikwert.
Bei dieser Grafik kann man erkennen, dass der `OnePlusOnePG` Algorithmus im wesentlichen auf allen Parametern gleich gut / schlecht ist. Es gibt viele Schwankungen, so dass man gute Parametereinstellungen nicht ohne weiteres erkennen kann.

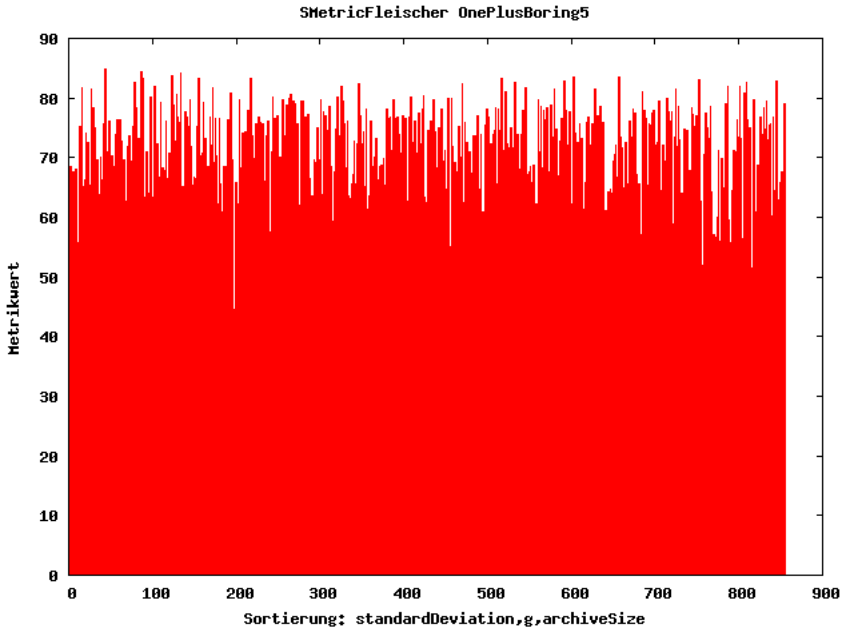

Abbildung 81: `MeanMetricPlotter`-Plot vom `OnePlusOnePG`. Aufgrund starker Schwankungen der Metrikwerte ist eine Analyse schwierig

7.4.1.2 Erste Evaluationsberechnung

Die Designs für diesen evaluationsbasierten Output waren wie folgt:

- Je nach Algorithmus die oben genannten guten Parametereinstellungen.
- Mindestens 10000 Evaluationen. Da jeder Algorithmus eine von verschiedenen Parametern abhängige Anzahl von Bewertungen durch den Simulator hat, kann man hier nicht generell den Parameter angeben, wodurch diese Anzahl an Evaluationen zustande kam. Für den `OnePlusOne` ergibt sich, dass z. B. die Anzahl der Evaluationen nur abhängig von der Anzahl der Generationen ist, während beim `BasisPSO` die Multiplikation aus den Parametern `cycles`, `populationSize` und der Anzahl der Generationen schließlich die Anzahl der Evaluationen ergibt. Generell wurde die Anzahl der Gesamtbewertungen auf 10000 festgesetzt, der Parameter `live` wurde auf 500 gesetzt. Falls ein Algorithmus eine Grundpopulation hat, so wurde diese auf 100 Individuen festgesetzt.

– Um einen Durchschnitt über eine bestimmte Anzahl an Läufen bilden zu können, wurde der `randomSeed` auf 500 verschiedene Werte festgesetzt.

Nachdem diese evaluationsbasierten Designs ausgewertet wurden, indem zuerst wieder die Resultate mittels `FilterAndDelete` auf drei Zielfunktionen heruntergebrochen und alle unzulässigen Polygonzüge entfernt wurden, konnten die Resultate mittels des `PercentageMetricPlotter` bewertet und visualisiert. Die Ergebnisse entsprachen nicht den Erwartungen, da diese alles andere als aussagekräftig waren. Die erstellten Grafiken zeigten über komplette Zeitachse jeweils nur einen Wert, der für den `OnePlusOne` bei 30% und für dem `MopsoOne` bei 90% lag. Alle Läufe brauchten die gleiche Anzahl an Bewertungen bzgl. einer Parametereinstellung, um einen Metrikwert zu erreichen, der mindestens 90% des besten erreichten Metrikwerts hatte. Auf der x-Achse sieht man dafür die Anzahl der benötigten Evaluationen. Die erstellten Bilder liefern Hinweise, dass der `OnePlusOne` scheinbar im Mittel über alle Läufe nur selten die 90% des besten erreichten Metrikwertes erreicht. Dieses liegt daran, dass der `OnePlusOnePG` sehr oft unzulässige Polygonzüge berechnet, die durch das `FilterAndDelete` gelöscht werden. Dies bedeutet, dass sehr viele Läufe kein einzigen zulässigen Polygonzug errechnet haben.
Hingegen erreicht der `MopsoOne` bei jedem Lauf den Metrikwert, der 90% des maximal erreichten Metrikwertes entspricht.
Nach Betrachtung der Grafiken von allen getesteten Algorithmen kam die Vermutung auf, dass die Algorithmen in der Anfangszeit (also unter 1000 Evaluationen) bereits recht gut werden und dann keine großen Veränderungen mehr auftreten. Die evaluationsbasierten Designs waren einfach wesentlich zu groß angelegt, um eine Aussage zu erhalten, welche Algorithmen auf dem Temperierbohrungsproblem schnell gut werden. Daraufhin wurden alle bereits getesteten Algorithmen mit weniger Gesamt-Evaluationen wiederholt gestartet.

7.4.1.3 Zweite Evaluationsberechnung

Da das erste gestartete Vorhaben den Verlauf der Algorithmen in einem zu späten Stadium betrachtete, wurden alle Designs noch einmal mit einer Evaluationsanzahl von 1000 gestartet, des weiteren nur mit 150 verschiedenen Random-Seeds (siehe Tabelle 13).
Dabei sind die gewählten Parametereinstellungen beim Aufruf `MopsoOne_1` die Parametereinstellungen, welche bei dem Anfangsdesign den größten Metrikwert erreicht haben, die Einstellungen der Parameter von `MopsoOne_2` sind diejenigen, die nach der Robustheitsuntersuchungen generell gut zu sein scheinen.
Die Resultate dieser Läufe (siehe Abbildung 82) sind wesentlich aussagekräftiger. Nachdem die Aufrufe aus der Tabelle 13 beendet waren, wurde für jeden Algorithmus insgesamt dreimal der `PercentageMetricPlotter` aufgerufen. Der erste Aufruf des Plotters diente dazu, die Referenzpunkte der einzelnen Algorithmen zu bestimmen. Damit die Algorithmen vergleichbar werden, braucht man für die S-Metrik nach Fleischer einen einheitlichen Referenzpunkt, sowie den besten erreichten Metrikwert bezüglich dieses Referenzpunktes.
Nachdem der `PercentageMetricPlotter` für alle Algorithmen einmal aufgerufen wurde, konnte man den Referenzpunkt für die S-Metrik festlegen. Für das Temperierbohrungsproblem mit diesen Läufen war der Referenzpunkt 2.1-5.0-50.0 geeignet.

Tabelle 13: Designaufrufe für die zweite Berechnung

Algorithmus	Design-Aufruf
OnePlusOne	-standardDeviation s 3 -g s 1000 -archiveSize s 1000 -r i 50 200 step 1
MopsoOne 1	-minInertia s 0.4 -c1 s 1 -c2 s 1 -g s 100 -quantityParticle s 10 -scalingInertia s 1 -maxInertia s 0.9
MopsoOne 2	-minInertia s 0.7 -c1 s 1 -c2 s 1 -g s 100 -quantityParticle s 10 -scalingInertia s 1 -maxInertia s 1
MueRhoLES	-archive s 100 -tau2 s 0.2 -tau1 s 0.2 -g s 15 -kappa s 0 -lambda s 70 -rho s 2 -mue s 10
BasisPSO	-archive s 100 -cycles s 10 -populationSize s 10 -c1 s 1.5 -c2 s 1 -g s 10 -inertia s 1
BasisGA	-mprob s 0.03 -rprob s 0.9 -g s 10 -psize s 100

Danach wurde für alle Algorithmen ein weiteres Mal der `PercentageMetricPlotter` aufgerufen, allerdings diesmal mit dem festgelegten Referenzpunkt für die Metrik. Durch diesen Aufruf wurde der maximalen Metrikwert aller Läufe und aller Algorithmen ermittelt. Der maximale Metrikwert wurde von dem `MopsoOne_1` erreicht und nahm einen Wert von knapp 258.3 an.
Anschließend wurde der `PercentageMetricPlotter` ein drittes Mal für alle Algorithmen aufgerufen, wobei der feste Referenzpunkt gesetzt wurde und der Parameter `-bestMetricValue` auf den vom `MopsoOne_1` erreichten Maximalwert gesetzt wurde.

Wie die Abbildung 82 zeigt, erreichen alle Algorithmen - eine Ausnahme stellt der `OnePlusOnePG` dar - innerhalb kürzester Evaluationsanzahl recht gute Ergebnisse. Die beiden `MopsoOne`-Einstellungen zeigten keine großen Unterschied. Auf der Grafik erkennt man, dass der `MopsoOne_1` leicht schlechtere Metrikwerte liefert als der `MopsoOne_2`. Beim `MopsoOne_1` hingegen wurden die Parameter genommen, die auf dem Temperierbohrungsproblem recht gut waren. Auch bei kleinerer Anzahl an Evaluationen erreichte der `MopsoOne_1` den besten Metrikwert, auch wenn er auf der Grafik „nur“ 99 % erreicht hat. Man kann sagen, dass sowohl die gefundenen guten Einstellungen, als auch die von der Robustheitsgruppe gefundenen guten Einstellungen auf dem Temperierbohrungsproblem gute Werte liefern.
Der `BasisGAPG` und der `BasisPSO` machen im Verlauf ihrer Entwicklung einen „Sprung“ bei ca. 100 Evaluationen. Dieses liegt daran, dass das Archiv nur alle 50 Evaluationen geprüft und aktualisiert wird. Damit werden Änderungen nur in 50er Schritten überhaupt angezeigt.
Der `OnePlusOnePG` schneidet im Vergleich zu den anderen Algorithmen sehr schlecht ab. Dieses liegt hauptsächlich daran, dass stets nur ein Individuum erzeugt und anschließend bewertet wird. Dabei gibt der Simulator nur die Grenzen des Werkstücks als gültigen Bereich und die resultierenden Polygonzüge, die der OnePlusOne erzeugt sind in über 70% der Fälle unzulässig.

Ein weiterer interessanter Algorithmus ist der `MueRhoLES`, der bei 100 Evaluationen einen „Sprung“ im Ergebnissgraphen aufweist und anscheinend danach konstant

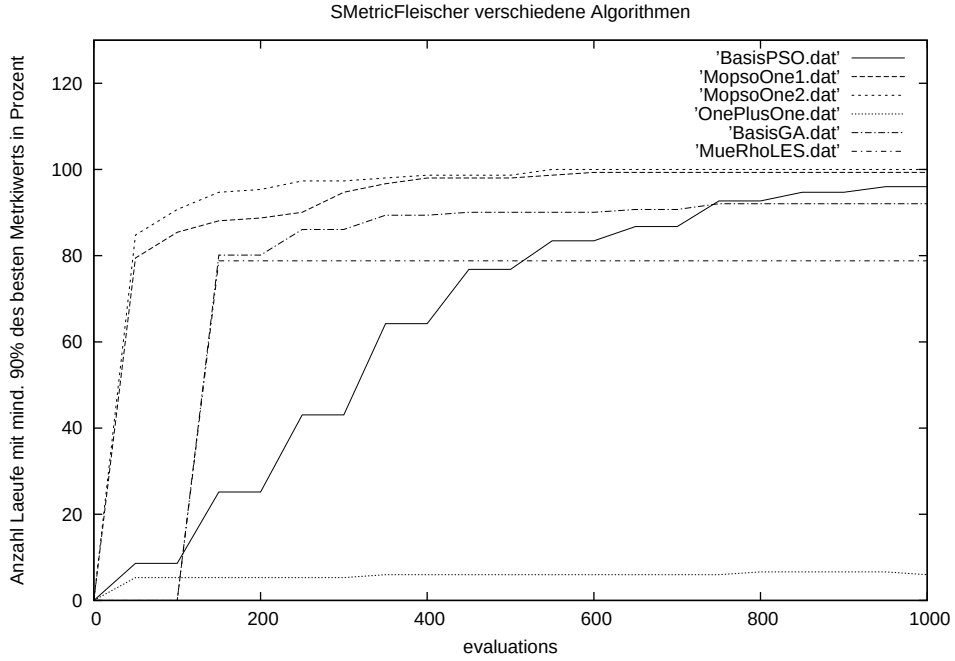

Abbildung 82: Verhalten der Algorithmen bei kleinerer Anzahl Evaluationen, veranschaulicht anhand des Mittelwerts über jeweils 150 Läufe

bleibt. Dieses Resultat war zunächst verwirrend, da bei den zuerst erzeugten evaluationsbasierten Berechnung dieser recht gut war. In der ersten Berechnungsphase erreichte der `MueRhoLES` bei allen Läufen mindestens 90% des besten Metrikwertes. Um dieses zu erklären wurden die gleichen Parametereinstellungen wie in der Tabelle 13 genommen, und auf 2000 Evaluationen erweitert. Die Abbildung 83 stellt dar, wie sich der MueRhoLES mit der Zeit entwickelt.
Der starke Sprung, der beim `MueRhoLES` zu beobachten ist, tritt exakt nach 100 simulierten Individuen auf. Bei dem benutzen Aufruf ist dies exakt die Größe der Startpopulation. Auffällig bei den Ergebnissen war weiterhin das erzeugte Datenvolumen. Das Archiv der paretooptimalen Lösungen wurde gegenüber anderen Algorithmen zwar langsamer gefüllt, allerdings konnte der `MueRhoLES` bis hin zu 9500 Evaluationen immer noch weitere paretooptimale Lösungen finden, während z. B. der `MopsoOne` schon bei 2000 Evaluationen keine neuen Lösungen mehr findet und die Anzahl der gefunden Lösungen im Archiv sogar wieder reduziert.

7.4.1.4 Anmerkungen

Die Anzahl der ungültigen Läufe, die Algorithmen bei der zweiten Evaluationsbewertung erzeugt haben, ist in der Tabelle 14 aufgelistet. Dies ist ein weiteres Kriterium, um die Güte eines Algorithmus zu beurteilen.

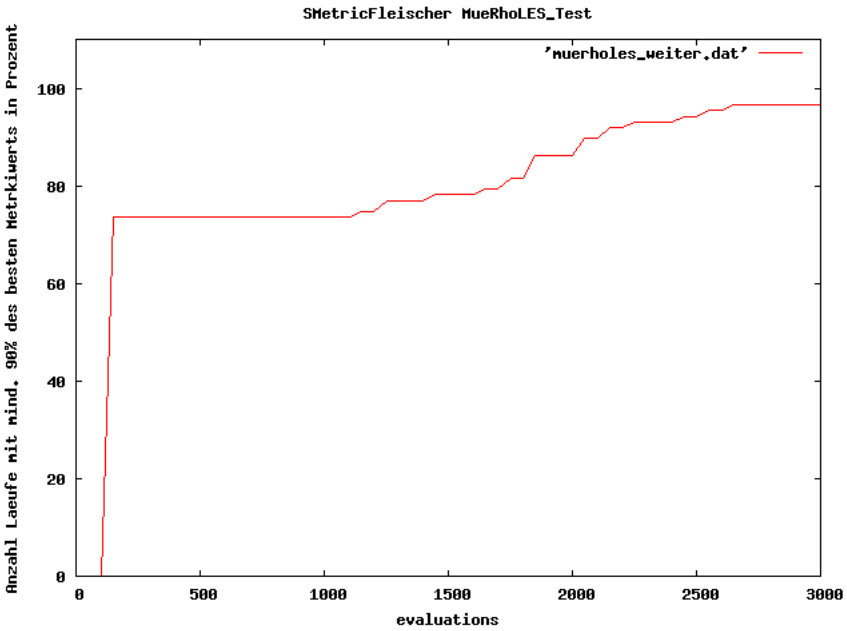

Abbildung 83: MueRhoLES-Lauf mit insgesamt 3000 Evaluationen

Tabelle 14: Anzahl ungültiger Läufe je Algorithmus

Algorithmus	Anzahl Evaluationen	Anzahl ungültige Läufe
OnePlusOne	100 bis 1000 Evaluationen	ca. 70%
BasisGA	50 Evaluationen	100%
	100 Evaluationen	100%
	150 bis 1000 Evaluationen	0%
BasisPSO	50 und 100 Evaluationen	ca. 30%
	150 und 200 Evaluationen	ca. 10%
	250 und 300 Evaluationen	ca. 2%
	350 bis 1000 Evaluationen	0%
Mopso 1	100 bis 1000 Evaluationen	0%
Mopso 2	100 bis 1000 Evaluationen	0%
MueRhoLES	50 und 100 Evaluationen	100%
	150 bis 1000 Evaluationen	0%

Die Anzahl der ungültigen Läufe spiegelt sich auch in der Abbildung 82 wieder. Dieses erklärt, warum z. B. der `OnePlusOnePG` immerzu schlechte Ergebnisse produziert, als auch die oben erwähnten „Sprünge“. Warum die Anzahl der ungültigen Läufe so unterschiedlich sind, hängt davon ab, wie die Selektion und Mutation des Algorithmus aussehen. Dieses Kriterium zeigt auch, wie gut ein Algorithmus mit der Zeit „lernt“ um sich zu verbessern.

7.4.2 Fahrstuhl

7.4.2.1 Gute Einstellungen für den Simulator

Bei dem verwendeten einfachen Fahrstuhlmodell (siehe 5.4.2.3) stellte sich sehr früh ein ernstes Problem bei der Wahl der richtigen Einstellung für den Simulator. Der S-Ring-Simulator arbeitet selbst mit Zufallszahlen und ermittelt einen Fitnesswert über einer festen Anzahl von Iterationen, die der Simulator für die Evaluation eines Individuums (genauer: einer Simulator-Eingabe) durchläuft. Mit steigender Anzahl dieser Iterationen wächst auch seine Genauigkeit. So variieren die Ergebnisse bei mehrfachen identischen Aufrufen mit weniger Iterationen wesentlich stärker als bei solchen Aufrufen mit deutlich mehr Iterationen.

Zu Beginn verwendeten wir einen voreingestellten Standardwert für die Anzahl der Iterationen von einer Million. Unsere ersten Tests zeigten, dass dabei die Laufzeit einer Simulation im Vergleich zu unseren Testfunktionen sehr lange dauerte. Tabelle 15 zeigt die Laufzeiten von 100 Simulationswiederholungen bei unterschiedlichen Iterationenanzahlen. Diese Daten wurden auf einem Testsystem gewonnen, das allen Läufen möglichst gleiche Voraussetzungen bot. Auch sind diese Angaben Mittelwerte aus fünf Wiederholungen.

Tabelle 15: Laufzeit von 100 Simulationen mit variierender Iterationenanzahl; der verwendete Wert ist hervorgehoben.

Iterationen	Laufzeit in Sekunden
10	≈ 5
100	≈ 5
1.000	≈ 5
10.000	≈ 6
100.000	≈ 17
1.000.000	≈ 128
10.000.000	≈ 1242

Durch die lange Laufzeit des Simulators wurde schnell deutlich, dass die Relevanz der Geschwindigkeit der implementierten Algorithmen, die bei Anwendung auf unsere Testfunktionen noch der bestimmende Faktor war, deutlich in den Hintergrund rückt. Umfangreichere Experiment-Designs waren mit dieser Laufzeit auf der uns zur Verfügung stehenden Hardware kaum realisierbar.

Aus dieser Not wurden intensive Laufzeitanalysen mit dem Simulator durchgeführt, bei denen die Anzahl der Iterationen pro Simulation variiert wurde. Die Tabellen 16 und 17 zeigen die Fitnesswerte für drei verschiedene Individuen, jeweils für die Ankunftswahrscheinlichkeiten $p \in \{0,3; 0,4\}$ und für sieben Iterationszahlen. Während in Tabelle 16 die Fitnesswerte nach einer Simulation aufgeführt sind, zeigt Tabelle 17 die gemittelten Fitnesswerte nach 100 Wiederholungen (siehe auch Kapitel 5.4.2.6).
Nach Abwägen der Geschwindigkeitsvorteile und dem Genauigkeitverlust ergab sich für uns eine sinnvolle Iterationenanzahl von 10.000. Der Informationsverlust gegenüber

einer Million Iterationen war dabei als vertretbar einzustufen, insbesondere vor dem Hintergrund der Ergebnis-Nachbearbeitung, auf die im Folgenden eingegangen wird.

Tabelle 16: Eine Simulation unterschiedlicher Individuen bei variierender Iterationenanzahl. Ab 10.000 Iterationen verbessern sich die Ergebnisse im Verhältnis zum Mehraufwand nur noch unwesentlich. Der verwendete Wert ist hervorgehoben.

	1. Individuum		2. Individuum		3. Individuum	
Iterationen	$p = 0.3$	$p = 0.4$	$p = 0.3$	$p = 0.4$	$p = 0.3$	$p = 0.4$
10	$= 2,80$	$= 3,10$	$= 2,80$	$= 3,10$	$= 2,80$	$= 3,10$
100	$= 5,17$	$= 5,44$	$= 2,65$	$= 3,46$	$= 2,60$	$= 3,76$
1.000	$\approx 5,92$	$\approx 5,94$	$\approx 2,97$	$\approx 3,30$	$\approx 2,87$	$\approx 3,52$
10.000	$\approx 5,99$	$\approx 5,99$	$\approx 3,17$	$\approx 3,42$	$\approx 3,02$	$\approx 3,47$
100.000	$\approx 6,00$	$\approx 6,00$	$\approx 3,11$	$\approx 3,44$	$\approx 2,99$	$\approx 3,46$
1.000.000	$\approx 6,00$	$\approx 6,00$	$\approx 3,11$	$\approx 3,44$	$\approx 3,02$	$\approx 3,46$
10.000.000	$\approx 6,00$	$\approx 6,00$	$\approx 3,10$	$\approx 3,44$	$\approx 3,02$	$\approx 3,45$

Tabelle 17: Mittelung von 100 Simulationen unterschiedlicher Individuen bei variierender Iterationenanzahl. Auch hier werden die Ergebnisse nach 10.000 Iterationen kaum besser. Der verwendete Wert ist hervorgehoben.

	1. Individuum		2. Individuum		3. Individuum	
Iterationen	$p = 0.3$	$p = 0.4$	$p = 0.3$	$p = 0.4$	$p = 0.3$	$p = 0.4$
10	$\approx 2,28$	$\approx 2,90$	$\approx 2,28$	$\approx 2,78$	$\approx 2,08$	$\approx 2,58$
100	$\approx 4,88$	$\approx 5,29$	$\approx 2,88$	$\approx 3,28$	$\approx 2,94$	$\approx 3,42$
1.000	$\approx 5,90$	$\approx 5,941$	$\approx 3,09$	$\approx 3,44$	$\approx 3,02$	$\approx 3,45$
10.000	$\approx 5,99$	$\approx 5,99$	$\approx 3,10$	$\approx 3,44$	$\approx 3,03$	$\approx 3,45$
100.000	$\approx 6,00$	$\approx 6,00$	$\approx 3,10$	$\approx 3,44$	$\approx 3,02$	$\approx 3,45$
1.000.000	$\approx 6,00$	$\approx 6,00$	$\approx 3,10$	$\approx 3,45$	$\approx 3,02$	$\approx 3,45$
10.000.000	$\approx 6,00$	$\approx 6,00$	$\approx 3,10$	$\approx 3,45$	$\approx 3,02$	$\approx 3,45$

Wie bereits erwähnt, arbeitet der Simulator selbst mit Zufallszahlen. Um diese zu initialisieren benötigt der Simulator einen Random-Seed, der ihm als Parameter übergeben wird. Um statistisch auswertbare Ergebninsse zu erhalten, werden Simulatorergebnisse verschiedener Random-Seeds gemittelt. Durch diese Arbeit vervielfacht sich die Laufzeit eines Algorithmus jedoch direkt. Auch wenn es sich lediglich um einen konstanten Faktor handelt, ist in der Praxis selbst eine Verdopplung der Laufzeit schmerzhaft.
Dies wird umgangen, indem während eines Algorithmen-Laufs nur mit einfachen Simulationen gerechnet wird (welche ein Verrauschen zur Folge haben) und lediglich die besten Lösungen des Gesamtlaufs durch Wiederholung und Mittelung nachbearbeitet werden. Dazu haben wir jedes Ergebnis für jedes Kriterium weitere 100 mal durch den Simulator evaluieren lassen und diese 100 Ergebnisse arithmetisch gemittelt. Diese

Mittelwerte bilden schließlich den endgültigen Fitnessvektor.

Ein alternativer Ansatz zur Beseitigung des Rauschens als die Wiederholung der Simulation mit gleicher Iterationenanzahl ist, die Nachbearbeitung mit nur einer einzelnen Simulation pro Kriterium, jedoch mit deutlich mehr Iterationen im Simulator durchzuführen.
Spezielle Versuche haben ergeben, dass der Unterschied zwischen den Ergebnissen einer Simulation mit 1.000.000 Iterationen und dem Mittel aus 100 Wiederholungen bei gleicher Iterationenanzahl relativ gering ausfällt. Dies bedeutet, dass eine Nachbearbeitung bei so vielen Iterationen nicht benötigt würde. Wie aus den Tabellen 16 und 17 ersichtlich ist, ist die Differenz zwischen einem Lauf mit 1.000.000 Iterationendem und dem Mittel aus 100 Wiederholungen mit 10.000 Iterationen deutlich größer. Aufgrund der Annahme, dass eine höhere Anzahl an Iterationen ein genaueres Ergebnis liefert, wäre es daher ohne eine Verschlechterung bei der Laufzeit möglicherweise vorteilhaft, statt 100 Wiederholungen mit 10.000 Iterationen lediglich eine Simulation mit 1.000.000 Iterationen durchzuführen.
Da diese Idee erst sehr spät entstand, wurden bei allen im Folgenden vorgestellten Läufen der Algorithmen die Nachbearbeitung mit der 100-fachen Wiederholung und Mittelung bei 10.000 Iterationen durchgeführt.

7.4.2.2 Idee und Vorgehen

Vor dem Hintergrund der Realzeitsimulation musste festgestellt werden, welche Algorithmen auf dem SRing-Modell die besten und welche innerhalb kürzester Zeit gute Ergebnisse liefern können.
Zu diesem Zweck wurden in einem ersten Schritt die im Projekt vorhandenen Algorithmen untersucht und somit der Kreis der Testkandidaten festgelegt. In diesen wurden nur der TabuSearch-Algorithmus (siehe 5.1.8) und der `TestAlgo` (siehe 5.1.1) nicht aufgenommen. TabuSearch entspricht nicht den geforderten Kriterien, da der S-Ring ein reellwertiges und kein kombinatorisches Problem darstellt. Der erste als Implementation zu Demonstrationszwecken entwickelte `TestAlgo` versprach als rein randomisierte Heuristik letztendlich keine viel versprechenden Ergebnisse. Außerdem wurde er an die im Laufe der Zeit weiterentwickelten Funktionen des *NObELTJE*-Frameworks nicht angepasst, so dass einige Entwicklungsarbeit notwendig gewesen wäre, um mit diesem Algorithmus effizient Versuche mit dem SRing-Modell durchführen zu können.

Nach der Auswahl der Kandidaten galt es, die Kriterien für die Versuche festzusetzen. Da als wichtiges Kriterium in unserem Experiment die „Zeit“ gelten sollte, musste als erstes eine Zeitdefinition gefunden werden, die einen Vergleich zwischen den sehr unterschiedlich arbeitenden Algorithmen ermöglicht. Da die Echtzeit unter anderem aufgrund der Notwendigkeit von Simulationen auf unterschiedlichen Rechnern ausschied, entschieden wir uns, als Zeitfaktor die Anzahl der Simulationsaufrufe zu nehmen. Dabei zählte die Simulation einer Lösung für alle Kriterien als ein Simulationsaufruf.

Zwar galt unser Hauptaugenmerk der Leistungsfähigkeit innerhalb kurzer Zeitspannen, jedoch interessiert es natürlich auch, wie die Algorithmen sich bei längeren Simulationen verbessern. Daher sollten die Algorithmen bis zu 50000 Evaluationen ausge-

wertet und die Zwischenergebnisse alle 100 Simulationsaufrufe festgehalten werden, um einen Vergleich im Langzeit-Verhalten durchführen zu können. Um ufgrund der Randomisierung der Algorithmen zufällig gute (oder schlechte) Läufe auszugleichen, wurde jedes Design mit etwa 100 unterschiedlichen Random-Seeds gestartet.

Das simulierte Gebäude hat sechs Stockwerke und zwei Aufzüge. Als zu optimierende Kriterien wurden die beiden Ankunftswahrscheinlichkeiten $0,3$ und $0,4$ gewählt.

Diese Werte haben einen rein beschreibenden Charakter der Eigenschaften des Gebäudes und des darin befindlichen Fahrstuhlsystems und wurden nicht durch wissenschaftliche Analysen gewählt. Eine Untersuchung über die Auswirkungen der Gebäudeparameter auf die Optimierungsverläufe der Algorithmen wäre ein weiterer, interessanter Aspekt.

Für die graphische Auswertung wurden der extra hierfür geschaffene `PercentageMetricPlotter` (siehe 5.3.6) und der `MeanMetricPlotter` (siehe 5.3.4), jeweils in Kombination mit der S-Metrik nach Fleischer (siehe 5.2.2.2), herangezogen.

Das Vorgehen bei der Analyse der einzelnen Algorithmen mit dem Simulator ähnelt sehr stark dem der Temperierbohrungs-Experimente (siehe 7.4.1.1). So wurden die ersten Versuche durchgeführt, um eine gute Konfiguration der Algorithmen zu gewährleisten. Aufgrund des engen Zeitrahmens konnte diese Phase leider nicht erschöpfend abgeschlossen werden, so dass nicht auszuschließen ist, dass durch weitere Optimierungen an den Parametern bessere Ergebnisse erzielt werden können. Die so gewonnenen und verwendeten Parametrisierungen können in Kapitel 7.4.2.6 nachgelesen werden.

Bei einigen Algorithmen stellte sich heraus, dass die im Allgemeinen als gut bewerteten Einstellungen auch auf dem S-Ring-Modell gute Ergebnisse liefern können (z. B. `BasisGAPG`, siehe 5.1.3). Insbesondere beim `MopsoOne` (siehe 5.1.6) gestaltete sich die Suche nach guten Parametern aufgrund der Vielfalt der Optionen als schwierig. Aus diesem Grund wurde dieser Algorithmus zweimal in die Bewertung aufgenommen: Einmal mit den optimierten Einstellungen die im Rahmen der ersten Experimente gefunden wurden („MopsoOne-1“), und einmal mit den Einstellungen nach Coello ([11], „MopsoOne-2“). Erstaunlicherweise führen diese Parametrisierungen zu gravierenden Unterschieden, die auf die Komplexität der Einstellmöglichkeiten des `MopsoOne` zurückzuführen sind.

Nachdem die ersten Ergebnisse vorlagen, wurden für den Percentage-Plotter der Referenzpunkt $(7,0; 7,0)$ und als `BestMetricValue` 20 festgelegt, um einen direkten Vergleich der Ergebnisse in einem Diagramm zu ermöglichen. Mit diesen hohen Werten wurde sichergestellt, dass keine Punkte einer Front ausgeschlossen und somit gute Ergebnisse beschnitten werden können.
Aufgrund der internen Funktionsweise des Simulators sind die Fitnesswerte immer kleiner als die Anzahl der simulierten Stockwerke. In diesem Fall ist der Fitnesswert jedes Kriteriums also kleiner als sechs. Aufgrund des gewählten Refernzpunktes be-

trägt der Metrikwert, der bei der verwendeten `SMetricFleischer` die „Fläche“ zwischen den Ergebnissen und dem Referenzpunkt entspricht, daher mindestens eins. Dies ist bei der Analyse mancher Ergebnisse von sehr hoher Relevanz (wie noch gezeigt wird).

7.4.2.3 Ergebnis

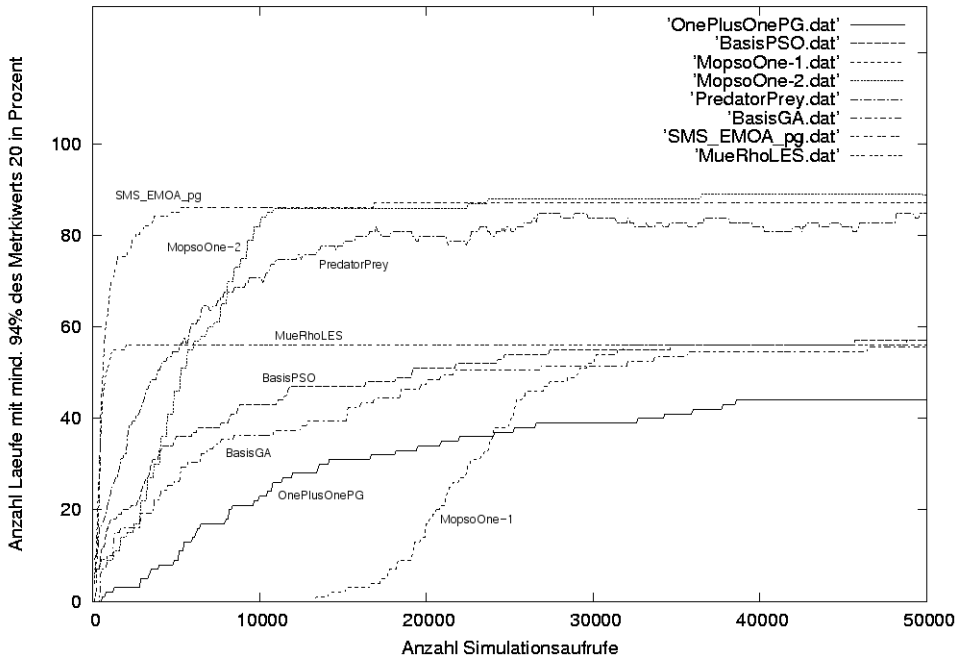

Abbildung 84: Auswertung des Experiments mit dem PercentageMetricPlotter nach *S-Metrik*. Übersicht mit den Ergebnissen bis 50000 Evaluationen.

Bei dem so erstellten Plot (Abbildung 84) kann man die getesteten Algorithmen bezüglich der Endergebnisse in zwei große Gruppen aufteilen: Der `SMS_EMOA_pg`, der `MopsoOne` (Parametrisierung aus der Literatur) und der `PredatorPrey` liefern relativ früh Ergebnisse, von denen zwischen 80 und 90 Prozent einen Metrikwert über $18,4$ liefern. Die nächste Gruppe (`BasisPSO`, `BasisGA`, `MopsoOne` mit selbst gewählten Einstellungen und `MueRhoLES`) pendelt sich um 55 Prozent ein, während der `OnePlusOnePG` deutlich darunter bleibt.

Betrachtet man nun die Zielsetzung, innerhalb kurzer Zeit gute Ergebnisse erzielen zu wollen, sollte man seinen Blick näher auf den Ursprung richten (siehe Abbildung 85). Auch hier gruppieren sich die Algorithmen im Großen und Ganzen nach zwei Kriterien: Während `SMS_EMOA_pg` und `MueRhoLES` ihre Ergebnisse in kurzer Zeit deutlich verbessern, benötigen die anderen eine längere „Anlaufphase“. Der `MopsoOne` mit selbst gewählten Einstellungen benötigt gar weit über 10.000 Simulationsaufrufe um überhaupt annähernd akzeptable Werte liefern zu können und ist daher in Abbildung 85 mit bis 3.000 Evaluationen auf der X-Achse noch nicht zu erkennen.

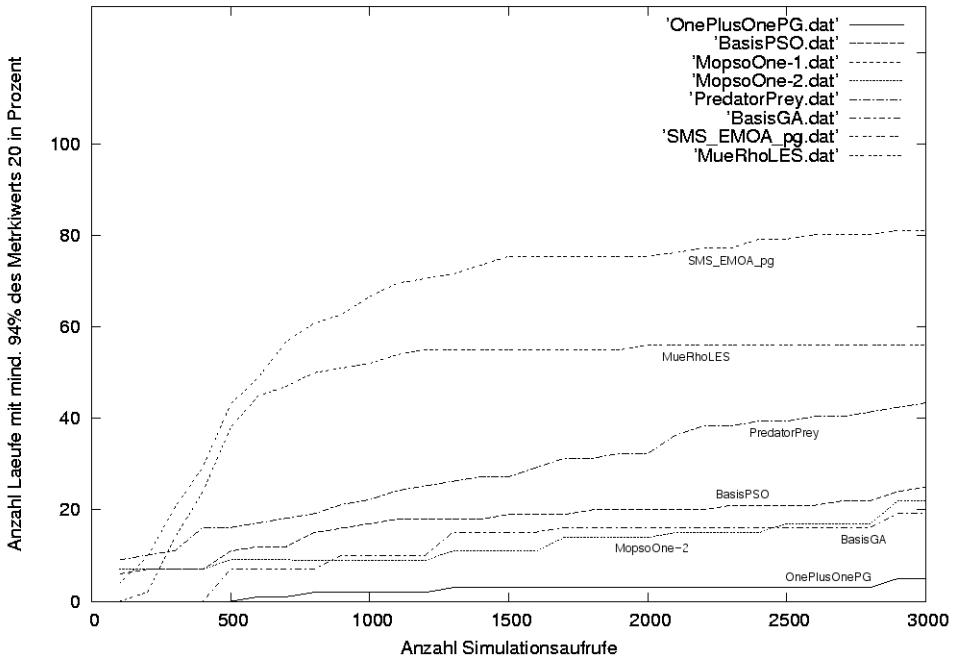

Abbildung 85: Auswertung des Experiments mit dem PercentageMetricPlotter nach *S-Metrik*. Ursprung aus Abbildung 84 bei vergrößerter Skalierung: Übersicht mit Ergebnissen bis 3000 Evaluationen. Bei Betrachtung der Ergebnisse nach wenigen Evaluationen zeigt sich ein etwas anderes Bild.

Wenn man die im Plot kleinstmögliche analysierbare Zeiteinheit (nach 100 Evaluationen) betrachtet ist zu erkennen, dass der `PredatorPrey`, gefolgt von `MopsoOne` mit evaluierten Parametereinstellungen und `BasisPSO`, bei kurzer Laufzeit die besten Ergebnisse findet.

7.4.2.4 Beobachtung einzelner Algorithmen

Dass der `BasisGA` erst nach 500 Evaluationen einen Metrikwert > 0 annimmt (siehe Abbildung 85), lässt sich durch die gewählte Populationsgröße erklären: Für jedes neu generierte Individuum wird ein Simulationsaufruf gestartet, um die Fitness zu bestimmen. Da aber erst nach der Generierung einer kompletten Folgegeneration mit 400 Individuen das Pareto-Archiv aktualisiert wird, können Änderungen der Pareto-Front nur nach jeweils 400 Simulationsaufrufen festgestellt werden. Während der ersten 400 Simulationsaufrufe bleibt das Pareto-Achiv also leer, so dass die in 100er Intervallen festgehaltenen Stände ebenfalls leer sind.

Bei dem außerordentlich guten Abschneiden des `SMS_EMOA_pg` muss berücksichtigt werden, dass dieser Algorithmus die S-Metrik zur Optimierung verwendet (siehe auch Kapitel 5.1.9). Daher ist es nicht verwunderlich, dass gerade diese Pareto-Fronten bei der Bewertung durch die hier verwendete Metrik belohnt werden.

Interessant zu beobachten ist auch der Verlauf des `PredatorPrey`. Dieser verschlechtert sich im Verlauf des Experiments zwischendurch mehrfach, wobei aber insgesamt dennoch eine steigende Tendenz zu verzeichnen ist (Abbildung 84). Diese Verschlechterungen werden durch den Selektionsoperator ermöglicht.

7.4.2.5 Zufall-Phänomen beim OnePlusOnePG

Der `OnePlusOnePG` zeigte ein erstaunliches Phänomen. Obwohl der Algorithmus mit 50.000 Generationen gestartet wurde, lieferte er gelegentlich schlechte Ergebnisse. Wie in Kapitel 7.4.2.2 bereits erwähnt, kann bei dem gewählten Gebäude der Fitnesswert für ein Kriterium nur weniger als sechs betragen. Tatsächlich findet der `OnePlusOnePG` in den betroffenen Fällen nur eine nicht dominierte Lösung, die stets nur Fitnesswerte größer als $5,99$ hat.

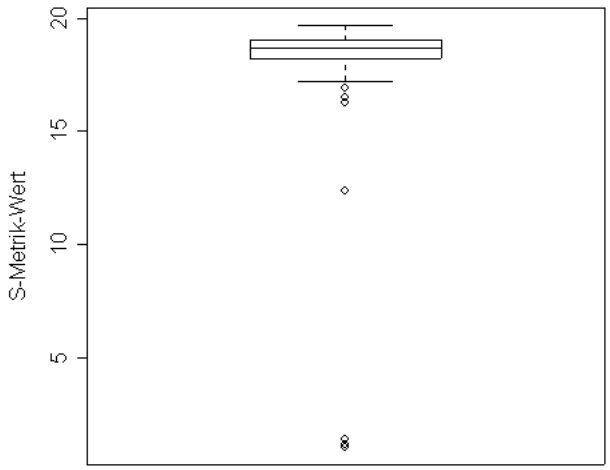

Abbildung 86: Box-Plot der *S-Metrik*-Werte vom `OnePlusOnePG` mit 100 verschiedenen Random-Seeds und 50.000 Generationen auf *SRing*: Es gibt Random-Seed-bedingt sehr starke Ausreißer.

Bei den vielen Läufen, die der Ermittlung guter Einstellungen für den Algorithmus dienten, zeigte sich, dass der gewählte Random-Seed einen deutlich höheren Einfluss auf die Ergebnisse hat, als der einzig wirkliche Parameter, die Standardabweichung. Abbildung 86 zeigt ein Box-Plot der *S-Metrik*-Werte eines `OnePlusOnePG`-Laufs mit 100 Random-Seeds, 50000 Generationen und einer Standardabweichung von zehn. Da bei der Berechnung der Metrikwerte der Referenzpunkt $(7,0; 7,0)$ verwendet wurde, ergibt sich ein Metrikwert von mindestens eins (siehe 7.4.2.2). Am Box-Plot wird deutlich, dass fast alle Läufe mit Metrikwerten bewertet werden, die größer als 17 sind. Es gibt lediglich wenige Ausreißer, die im Extremfall mit Metrikwerten von ungefähr

eins bewertet werden. Dies zeigt, dass beim `OnePlusOnePG` in Abhängigkeit von der Initialisierung seiner Zufallszahlen diese extremen Einschnitte in der Qualität nur vereinzelt auftreten. Die meisten Ergebnisse variieren in einer Art, wie es von vielen Algorithmen bekannt ist. So beträgt der minimale Metrikwert im durchgeführten Versuch $1,01$, während das Maximum bei $19,71$ und das arithmetische Mittel bei 18 liegt. Nach der Entdeckung dieses Phänomens haben wir den Algorithmus u. a. mit den gleichen Random-Seeds auf mehreren Testfunktionen gestartet, das Phänomen dort jedoch nicht erzeugen können. Bisher haben wir für dieses Phänomen auch keine klare Ursache gefunden.

7.4.2.6 Verwendete Designs

Im Folgenden werden die wie in Kapitel 7.4.2.2 beschrieben ermittelten Parameter für weitergehende Experimente oder Verifikationszwecke aufgeführt. Die Ergebnisse, die in die Auswertung dieses Experiments einflossen, wurden mit genau diesen Werten gewonnen.

BasisGAPG Das Design, das zur Erstellung des Vergleichs für den `BasisGAPG` gewählt wurde, beinhaltet die folgenden Parameter:

- Populationsgröße: 400
- Generationen: 125
- Mutationswahrscheinlichkeit: 0,01
- Rekombinationswahrscheinlichkeit: 0,6

BasisPSO Der `BasisPSO` wurde folgendermaßen konfiguriert:

- Cycles: 150
- Generationen: 20
- Partikelanzahl: 17
- c1: 1
- c2: 0.5
- inertia: 0.2
- Archivgröße: 1000

MopsoOne Der `MopsoOne`, welcher sehr viele Einstellmöglichkeiten bietet, wurde mit zwei verschiedenen Einstellungen getestet, die erste ist durch eigene Versuche entstanden, die zweite ist der Vorschlag gemäß Coello Coello [11]:

1.
   - Generationen: 1000
   - Partikelanzahl: 50

- c1: 1.2
- c2: 1.2
- Trägheitswert inertia: 0.6
- Veränderung des Trägheitswertes während des Laufes: deaktiviert (0)
- Maximale Geschwindkeit eines Partikels: 5
- Anzahl Hyperkuben: 150
- Archivgröße: 150

2.
- Generationen: 125
- Partikelanzahl: 400
- c1: 1
- c2: 1
- Trägheitswert inertia: 0.4
- Veränderung des Trägheitswertes während des Laufes: deaktiviert (0)
- Maximale Geschwindkeit eines Partikels: 100
- Anzahl Hyperkuben: 30
- Archivgröße: 200

MueRhoLES Für den `MueRhoLES`-Algorithmus, der recht viele Einstellmöglichkeiten bietet, wurden die folgenden Parameter gewählt:

- Generationen: 500
- Größe der Elternpopulation (μ): 15
- Größe der Offspringgeneration (λ): 100
- Anzahl der Eltern pro Individuum (ρ): 3
- Lebenszeit eines Individuums (κ): unbeschränkt (0)
- tau1: 0.1
- tau2: 0.2
- Archivgröße: unbeschränkt (0)

OnePlusOnePG Beim einfachen `OnePlusOnePG`-Algorithmus mussten nicht viele Einstellungen festgelegt werden:

- Generationen: 50000
- Standardabweichung bei der Mutation: 10
- Archivgröße: unbeschränkt (0)

PredatorPrey Für den `PredatorPrey`-Algorithmus wurden die folgenden Parameter gewählt:

- Archivgröße: 500
- Generationen: 25000
- Selektionsmethode: „0“, nach Laumanns (vergl. auch Kap. 5.1.7 und [12])
- Varianz: 0,5

SMS_EMOA_pg Das Design des `SMS_EMOA_pg` wurde mit den folgenden Parametern gestartet:

- Generationen: 50000
- μ: 100
- Selektionsmethode: „0“, die empfohlene Variante mit komplettem Funktionsumfang (siehe auch Kap. 5.1.9)

8 Zusammenfassung, Fazit

Die Aufgabe der Projektgruppe, ein Softwareprodukt zu erstellen, das als Experimentier-Umgebung für die Mehrzieloptimierung mit heuristischen Verfahren dient, wurde im Sommersemester 2004 mit *NObELTJE* erfüllt.
Das zweite Semester wurde zur Erforschung der Heuristiken genutzt.
Neben der Implementierung der verschiedenen Ansätze, zweier evolutionären Strategien, zweier Particle Swarm Optimization Ansätzen, einem genetischen Algorithmus, einem Räuber-Beute-Algorithmus und einem Tabu Search Ansatz, wurde der SMS-EMOA-Algorithmus um Selektionskriterien erweitert. Zur Beurteilung der Ergebnisse der genannten Algorithmen auf Testfunktionen und Anwendungsproblemen wurden ausgewählte Metriken, die Euklid-Metrik, die C-Metrik und die S-Metrik nach der Implementierung von Fleischer von der PG447 implementiert. Neben diesen mathematischen Beurteilungen wurde *NObELTJE* außerdem mit Visualisierungs-Tools, die die Ergebnisse und deren Metrik-Werte darstellen, bzw. Statistik-Tools mit Anbindung an die Sprache *R* versehen. Als Optimierungsprobleme wurden einige Testfunktionen, u. a. auch die ZDT-Funktionen, sowie die Anbindung an das Fahrstuhl- und das Temperierbohrungsproblem implementiert. *NObELTJE* wurde unter der GPL Lizenz veröffentlicht. Weitere Informationen dazu sind im Kapitel Lizenz (6.3) verfügbar.

Zu Beginn des zweiten Semesters wurden die Gruppen „Visualisierung“, „Fahrstuhl“, „Temperierbohrung“, „Robustheit“ und „Metriken-Selektion“ gebildet. Ihre Arbeit wird im Folgenden zusammengefasst.

8.1 Visualisierung

Die Visualisierungsgruppe beschäftigte sich mit Statistik, Darstellungsformen und grafischen Analyseverfahren (siehe Kapitel 7.1). Einige dieser Darstellungstechniken und statistischen Verfahren wurden in *NObELTJE* implementiert (siehe Kapitel 5.3) und konnten in der Forschung der Projektgruppe verwendet werden. Viele der Abbildungen in diesem Endbericht wurden mit diesen Tools erzeugt. Verfahren, die sich nicht automatisieren ließen, konnten erfolgreich angewendet werden, um Daten der anderen Gruppen zu analysieren.

8.2 Anwendungsproblem Fahrstuhl

Ziel im zweiten Semester war es, einen Fahrstuhlsimulator aus der Industrie in das *NObELTJE*-Projekt einzubinden (siehe Kapitel 5.4.1.10) und Analysen hinsichtlich der Echtzeitoptimierung durchzuführen (siehe Kapitel 7.4.2.2).

Umgesetzt wurde dies mit dem S-Ring-Modell (siehe Kapitel 5.4.2.3). Nach ausführlichen Tests wurden anschließend Simulationsläufe mit allen im Projekt zur Verfügung stehenden und relevanten Algorithmen durchgeführt und ein Diagramm erstellt (siehe Kapitel 7.4.2.3), das einen guten Überblick über deren Leistungsfähigkeit gibt.
Die so gewonnenen Erkenntnisse unterstützen den Leser bei der Auswahl der richtigen Algorithmen und Parameter zur näheren Untersuchung der Echtzeit-Simulation.

8.3 Anwendungsproblem Temperierbohrung

Die Integration des Simulators für die Temperierbohrungen in das nach dem ersten Semester vorhandene Projekt war der wesentliche Punkt, der von diesem Teil unserer Projektgruppe bearbeitet wurde.

Diese Aufgabe wurde erfolgreich gelöst und mit einigen Experimenten abgeschlossen, die das Verhalten diverser Algorithmen auf diesem praxisrelevanten Problem zeigen sollten. Dabei ergab sich, dass sämtliche von der PG implementierten Algorithmen in der Lage sind, auf diesem Problem Lösungen zu finden. Die Qualität der Algorithmen variierte allerdings stark. Durch schnelle gute Ergebnisse tat sich vor allem der mehrkriterielle Particle Schwarm Optimization Ansatz `MopsoOne` hervor, wie in der Abbildung 82 zu sehen ist.

8.4 Robustheit

Die Gruppe Robustheit sollte die Algorithmen auf die Robustheit der Ergebnisse bei unterschiedlichen Parametereinstellungen untersuchen, also Parametertuning durchführen.

Dazu wurde der `MopsoOne` auf verschiedenen Testfunktionen angewendet. Bei allen untersuchten Testfunktionen fiel auf, dass die Parameter `inertia` und `c2` besonders großen Einfluss auf die Güte der Ergebnisse hatten, und die richtige Wahl dieser Parameter gute Ergebnisse garantieren sollte.

Bei dem Vergleich der Algorithmen `MopsoOne, MueRhoLES` und `OnePlusOnePG` erkannten wir, dass der `OnePlusOnePG` bei einfachen Testfunktionen (*Schaffer*, *Deb*) kaum schlechter abschneidet als `MopsoOne` und `MueRhoLES`, bei der *ZDT1* jedoch deutlich schlechter ist.

Bei der Untersuchung der Populationsgrößen des `MopsoOne` bei verschiedenen Anzahlen von Funktionsauswertungen erkannten wir, dass bei einer größeren Anzahl der zur Verfügung stehenden Funktionsauswertungen auch eine größere Anzahl der Partikel vorzuziehen ist.

8.5 Metriken-Selektion

Diese Kleingruppe sollte nach einer Metrik suchen, die als Selektionskriterium in den SMS-EMOA integriert werden sollte und die Lösungen des SMS-EMOA, der mit unterschiedlichen Selektionskriterien auf den ZDT-Funktionen angewendet wurde, vergleichen.

Mit der `selectDomPoints`-Metrik hat die PG447 eine Metrik gefunden, die als Selektionskriterium benutzt werden kann. Der Vergleich des SMS-EMOA (siehe Kapitel 7.2) in der Originalfassung mit dem Selektionskriterium `selectDomPoints` ergab, dass die neue Fassung nicht schlechter als die originale ist und sogar Vorteile im Rechenaufwand und auf den ZDT-Funktionen besitzt.

Literatur

[1] Die *NObELTJE* Homepage: http://www.isf.de/projects/pg447/index.html

[2] Das Eclipse Projekt: http://www.eclipse.org

[3] Webseite des JUDE UML Tools: http://JUDE.esm.jp/

[4] Die Gnuplot Dokumentation: http://www.Gnuplot.info/docs/Gnuplot.pdf

[5] V. Nissen. Einführung in Evolutionäre Algorithmen. Vieweg, Braunschweig, 1997.

[6] D. E. Goldberg, K. Deb. A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. In: G. J. E. Rawlins (Hrsg.), Foundations of Genetic Algorithms, I, Morgan Kaufmann, 69-93, 1991.

[7] H.–P. Schwefel. Kybernetische Evolution als Strategie der experimentellen Forschung in der Strömungstechnik. Diplomarbeit, Hermann Föttinger-Institut für Hydrodynamik, Technische Universität Berlin, März 1965.

[8] I. Rechenberg. Cybernetic Solution Path of an Experimental Problem. Royal Aircraft Establishment, Library Translation 1122, August 1965.

[9] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan. A Fast and Elitist Multi-Objective Genetic Algorithm: NSGA-II. Parallel Problem Solving from Nature, 4(1), 849-858, 2000.

[10] K. E. Parsopoulos, M. N. Vrahatis. Particle Swarm Optimization Method in Multiobjective Problems. ACD Symposium of Applied Computing (SAC), 603-607, 2002.

[11] C. A. Coello Coello, M. S. Lechuga. MOPSO: A Proposal for Multiple Objective Particle Swarm Optimization. Congress on Evolutionary Computation (CEC 2002), IEEE Service Center, Piscataway, New Jersey, Volume 2, 1051-1056, Mai 2002.

[12] M. Laumanns. Ein verteiltes Räuber-Beute-System zur mehrkriteriellen Optimierung. Diplomarbeit, Universität Dortmund, 1999.

[13] F. Glover, F. Laguna. Tabu Search. Kluwer Academic Publishers, ISBN 079239965X, 1997.

[14] M. P. Hansen. Tabu Search in Multiobjective Optimisation: MOTS, Proceedings of the 13th International Conference on Multiple Criteria Decision Making (MCDM 1997), Cape Town, South Africa, 1997.

[15] A. Hertz, B. Jaumard, C. Ribeiro, W. F. Filho. A multi-criteria tabu search approach to cell formation problems in group technology with multiple objectives, RAIRO/Operations Research, 303-328, 1994.

[16] M. Emmerich, N. Beume, B. Naujoks. An EMO Algorithm using the Hypervolume Measure as Selection Criterion. In C. A. Coello Coello et al. (Hrsg.): Evolutionary Multi-Criterion Optimization, Third International Conference, EMO 2005, Guanajuato, Mexico, Springer, Lecture Notes in Computer Science, Volume 3410, 62-76, 2005.

[17] M. Fleischer. The Measure of Pareto Optima - Applications to Multi-objective Metaheuristics. In Evolutionary Multi-Criterion Optimization (EMO 2003), LNCS 2632, Springer, 519-533, 2003.

[18] K. Deb. Multi-Objective Optimization using Evolutionary Algorithms. John Wiley & Sons, Ltd, 2001.

[19] Y. Collette, P. Siarry. Principles and Case Studies. In Multiobjective Optimization, Springer, 177-197, 2003.

[20] A Platform and Programming Language Independent Interface for Search Algorithms (PISA). http://www.tik.ee.ethz.ch/pisa/

[21] Kanpur Genetic Algorithm Laboratory (KanGAL). http://www.iitk.ac.in/kangal/

[22] E. Zitzler, K. Deb, L. Thiele. Comparison of multiobjective evolutionary algorithms: Emprical results. Evolutionary Computation, 173-195, Sommer 2000.

[23] J. D. Knowles. Local - Search and Hybrid Evolutionary Algorithms for Pareto Optimization. phdthesis, Department of Computer Science, University of Reading, Reading, UK, Januar 2002.

[24] E. Zitzler, L. Thiele. Multiobjective evolutionary algorithms: A comparative case study and the strength Pareto approach. IEEE Transactions on Evolutionary Computation, 257-271, November 1999.

[25] K. Deb. Multi-Objective Genetic Algorithms: Problem Difficulties and Construction of Test Problems Evolutionary Computation, 7(3):205-230, Herbst 1999.

[26] J. Koehler, D. Ottinger. An AI-Based Approach to Destination Control in Elevators. Schindler Lifts, Ltd, Research and Development, Ebikon, Switzerland.

[27] M.–L. Siikonen. Planning and Control Models for Elevators in High-Rise Buildings. Helsinki University of Technology, Systems Analysis Laboratory, Research Reports A68, 1997.

[28] J. Koehler, K. Schuster. Elevator Control as a Planning Problem. Schindler Lifts, Ltd, Ebikon, Switzerland.

[29] Thomas Bartz–Beielstein, Mike Preuß, Sandor Markon. Validation and optimization of an elevator simulation model with modern search heuristics. Metaheuristics: Progress as Real Problem, 109-128, Kluwer, Boston MA, 2005 (im Druck befindlich)

[30] Thomas Bartz–Beielstein, Department of Computer Science University of Dortmund, D-44221 Dortmund Tel.: +49 231 9700-977, Fax: +49 231 9700-959 E-Mail: Thomas.Bartz-Beielstein@UDo.edu

[31] S. Markon, D. V. Arnold, T. Bäck, T. Beielstein, H.–G. Beyer. Thresholding – A selection operator for noisy ES. Proc. 2001 Congress on Evolutionary Computation (CEC'01), 465-472, IEEE Press, Piscataway NJ, 2001.

[32] Spritzgussform mit Werkstück: http://www.betriebmeister.de/O/121/Y/67234/default.aspx

[33] Die GPL Lizenz: http://www.gnu.org/licenses/gpl.txt

[34] Die inoffizielle deutsche Übersetzung der GPL Lizenz: http://www.gnu.de/gpl-ger.html

[35] Webseite der KEA Toolbox: http://ls11-www.cs.uni-dortmund.de/people/schmitt/kea.jsp

[36] C. H. Yu, S. Stockford. Evaluating spatial- and temporal-oriented multi-dimensional visualization techniques for research and instruction (2003). Aries Technology & Cisco Systems und Arizona State University.

[37] W. M. Spears. An Overview of Multidimensional Visualization Techniques. AI Center - Code 5514. Naval Research Laboratory. Washington, DC 20375.

[38] H. Pohlheim. Visualization of Evolutionary Algorithms: Real-World Application of Standard Techniques and Multidimensional Visualization DaimlerChrysler AG, Research and Technology. Alt-Moabit 96a, 10556 Berlin, Deutschland.

[39] H. Pohlheim. http://www.pohlheim.com/diss/text/diss_pohlheim_ea-27.html