PG 431 — Metaheuristiken

„Neue Ideen für die Optimierung“

Endbericht

Teilnehmer

Selcuk Balci, Sören Blom, Daniel Blum,
Vedran Divkovic, Dirk Hoppe, Djamila Lindemann,
Ulf Schneider, Bianca Selzam, Thomas Tometzki,
Marko Tosic, Igor Vatolkin und Stefan Walter

Betreuer

Thomas Bartz–Beielstein, Jörn Mehnen und Karlheinz Schmitt

Der vorliegende Endbericht stellt die Vorgehensweise und Arbeitsergebnisse der Projektgruppe 431 - Metaheuristiken im zweiten Halbjahr, Wintersemester 2003/2004, dar. Zunächst werden Aspekte zur Entwicklung des Tools „MooN“, sowie dessen konzeptioneller Aufbau vorgestellt. Auf die Präsentation der implementierten Plug-Ins folgt die Analyse des Tools mittels Versuchsplanung und statistischer Auswertungen. Weiterhin werden die semesterbegleitenden Vorträge aller PG-Teilnehmer vorgestellt.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Dieses Kapitel dient als Einführung in die Arbeit der PG 431. Zunächst wird ein Überblick über diesen Bericht gegeben. Anschließend folgen die Motivation der PG-Teilnehmer für die Arbeit innerhalb der Projektgruppe sowie der Zeitplan des zweiten Semesters.

1.1 Aufbau dieses Endberichts

Dieser Endbericht ist in drei Teile gegliedert. Der erste Teil fasst die Arbeit der PG 431 im Wintersemester 2003/2004 zusammen. Die Arbeit des Sommersemesters 2003 wurde bereits im Zwischenbericht [BBB+03] beschrieben. Das erste Kapitel dient als Einleitung und beschreibt die Motivation der PG-Teilnehmer, sowie den Zeitplan, den die Projektgruppe während des Semesters verfolgt hatte.

Das zweite Kapitel befasst sich mit der Entwicklung von unserem Softwareprodukt MooN. Es werden nicht nur bestimmte Tools vorgestellt, die uns während der Implementierungsphase die Arbeit erleichterten, sondern auch Fragen zur Lizenzierung von MooN geklärt sowie die Implementierung und die graphische Oberfläche präsentiert.

Der konzeptionelle Aufbau von MooN und die Vorstellung der realisierten Plug-Ins erfolgt im dritten Kapitel. Zunächst werden die Plug-In-Struktur sowie die interne Ablaufsteuerung vorgestellt, anschließend die von uns realisierten Heuristik-, Problem- und ExitCondition-Plug-Ins einzeln beschrieben. Abschließend finden sich Erklärungen zur Ausgabe, Visualisierung und Auswertung der gewonnenen Daten.

Das vierte Kapitel befasst sich tiefer mit der statistischen Auswertung und den Versuchen, die mit Hilfe von MooN durchgeführt wurden. Hierbei werden sowohl die Fragestellungen zu den Versuchen, als auch deren tatsächliche Durchführung berücksichtigt.

Der zweite Teil dieses Endberichts enthält kurze Zusammenfassungen der Seminarvorträge, die von jedem PG-Teilnehmer während des Semesters gehalten wurden.

Der dritte Teil letztendlich besteht aus dem Glossar, welches einen Überblick über die wichtigsten von uns verwendeten Konzepte und Begriffe verschafft.

1.2 Motivation und Erfahrungen der Teilnehmer

Die Motivationen zur Teilnahme an der Projektgruppe waren sehr unterschiedlich. Doch neben dem Erwerb des Scheins haben sich auch andere Erwartungen erfüllt.

Viele Teilnehmer waren freudig überrascht, so viel über Gruppenarbeit gelernt zu haben. So hatte sich die Arbeitsweise der Gruppe mit der Zeit verändert, war effektiver und entspannter geworden. Insbesondere konnte das Organisieren und Leiten von Sitzungen sowie die Darstellung von erlangten Ergebnissen in Präsentationen geübt werden, wobei die Rückmeldungen aus der Gruppe und von den Betreuern eine stetige Verbesserung der Fähigkeiten in diesen Bereichen bewirkten.

Auch inhaltlich gab es in den beiden Semestern einiges zu lernen. Die Teilnehmer haben durch die eigene Implementierung von Heuristiken und Problemen sowie durch die Seminare und Versuche einen großen Einblick in die Thematik der „modernen“ Optimierung mit Heuristiken bekommen. Zusätzlich konnten Erkenntnisse über Modellierung, Programmierung und Teamworktools (TWiki, BugTracker) gesammelt werden.

Auch der kompetente Eindruck der Betreuer hat sich bestätigt. So ist vor allem die vornehme Zurückhaltung der Betreuer zu nennen, die es den Teilnehmern gerade in schwierigen Situationen ermöglichte, eigene Erfahrungen in der gemeinschaftlichen Planung und Durchführung eines größeren Softwareprojekts zu erlangen. Weiterhin gaben die Betreuer häufig nützliche Hinweise und Anregungen und waren immer erreichbar und hilfsbereit. Sie

ließen den Teilnehmern viel Freiraum in ihrer Arbeitsweise und der Gestaltung von MooN, so dass die Arbeit interessant blieb.

Zusammenfassend lässt sich sagen, dass sich die Erwartungen der Teilnehmer erfüllt haben und die PG neben Stress und Aufgaben auch viel Spaß zu bieten hatte.

1.3 Zeitplan

Aufteilung in Teilgruppen

Auf Grund guter Erfahrungen wurde entschieden, im 2. Semester stärker in Teilgruppen zu arbeiten. Dabei hatte jede Teilgruppe einen Gruppenkoordinator, der als Ansprechperson zur Verfügung stand. Zusätzlich legte jede Gruppe eine Seite im TWiki an, auf der der aktuelle Arbeitsstand für jeden einzusehen war. Die Gruppen entstanden bei Bedarf und lösten sich nach Beendigung des Gruppenzieles wieder auf. Meist gehörte jedes PG-Mitglied mehreren Gruppen gleichzeitig an. Das Gantt-Chart 1.1 zeigt, welche Teilgruppen von der 42. Kalenderwoche 2003 bis zur 8. Kalenderwoche 2004 zu welcher Zeit aktiv waren.

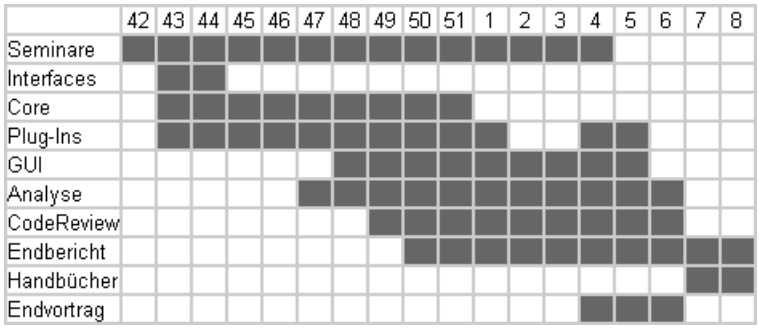

Abbildung 1.1: Untergruppen während des Semesters.

Seminare

Die Seminarphase des 2. Semesters gestaltete sich so, dass jedes der zwölf Mitglieder der Projektgruppe einen wissenschaftlichen Vortrag hielt. Fast jede Woche fand einer der Vorträge statt. Die Themen umfassten neuere

Metaheuristiken, Implementationsaspekte einiger Metaheuristiken, das NFL-Theorem und Konkurrenzprodukte zu MooN.

Interfaces

Diese Teilgruppe entwarf und programmierte die Interfaces von MooN. Dazu gehören die Schnittstellen des Programmkerns zu den Heuristiken und Problemen. Außerdem wurden Interfaces für Individuum, Population und Abbruchbedingung implementiert.

Core

Diese Teilgruppe programmierte den Kern des Programms, der den Ablauf steuert. Hier werden die erstellten Gesamtläufe verwaltet, neue Heuristiken und Probleme als Plug-Ins ins System aufgenommen, die Gesamtläufe ausgeführt und relevante Daten in Dateien geschrieben.

Plug-Ins

Diese Teilgruppe hat eine Menge Heuristiken und Probleme mit passenden Schnittstellen implementiert. Bereits implementierte Probleme wurden an die Schnittstelle von MooN angepasst. Außerdem wurden Operatoren und Abbruchbedingungen implementiert. Der größte Teil der Aufgaben war bis zur 50. Woche erledigt. Danach wurden noch Ant Colony Optimization, Travelling Salesperson Problem und Pickup and Delivery Problem implementiert und die Heuristiken um Operatoren erweitert, die auf Permutationen arbeiten.

GUI

Die GUI-Gruppe hat sich um die graphische Oberfläche von MooN gekümmert. Man kann damit einen Gesamtlauf zusammenstellen, speichern, abändern und ausführen lassen. Die Plug-Ins lassen sich einfach installieren und deinstallieren. Auch eine Visualisierung der Laufzeitdaten während der Ausführung ist eingebaut.

Versuche und Analyse

Die Analysegruppe testete die Güte der Heuristiken auf Testfunktionen aus der Literatur. Zusätzlich wurden günstige Parametereinstellungen für einige Heuristiken ermittelt. Dies geschah nach Versuchsplänen, die anhand der Leitlinie Design of Experiments erstellt wurden. Zur Auswertung und Visualisierung wurde das Statistiktool `R` verwendet.

CodeReview

Die Gruppe CodeReview stellte Kriterien auf, an die der Code angepasst werden sollte. Diese „best practices“ sollten von allen Projektmitgliedern in deren Code umgesetzt werden.

Endbericht

Die Mitglieder der Teilgruppe Endbericht stellten eine Gliederung auf. Anschließend beteiligte sich jeder Teilnehmer der PG daran, die Texte zu verfassen. Die Gruppe Endbericht sammelte die Texte, korrigierte sie und stellte sie zum Endbericht zusammen.

Handbücher

Zum Produkt MooN gehören auch ein Benutzerhandbuch und ein Entwicklerhandbuch, die beide in englischer Sprache verfasst sind. Ersteres beschreibt die Benutzung von MooN mit den darin enthaltenen Problemen und Heuristiken. Das Entwicklerhandbuch enthält die Beschreibung der Java-Klassen des Frameworks und eine Anleitung, wie man eigene Plug-Ins an das Programm anbindet.

Endvortrag

Die Mitglieder dieser Teilgruppe erstellten eine Präsentation, welche Einblick bietet in die Gruppenarbeit der Projektgruppe, in MooN, in die vorgenommen Experimente und vieles mehr. Der Endvortrag wurde am Ende der 6. Kalenderwoche gehalten.

Planung und Umsetzung

Das zweite Semester der Projektgruppe war angesetzt für den Zeitraum der 42. Woche 2003 bis zur 6. Woche 2004. Die Aufteilung in Teilgruppen konnte die Effizienz der Arbeit immens steigern. So gab es im zweiten Semester keine langen Diskussionen der gesamten Gruppe mehr und viele Aufgaben wurden parallel erledigt. Nach der gründlichen Planung von MooN im 1. Semester entschieden wir uns im 2. Semester, sofort mit der Implementierung zu beginnen. Am Anfang lag der Schwerpunkt auf dem Zusammenspiel der Komponenten des Programms ohne GUI. Da die GUI und Analyse viel mehr Zeit in Anspruch nahmen als erwartet, konnten Handbücher und Endbericht erst am Ende der 8. Kalenderwoche 2004 fertig gestellt werden.

Kapitel 2

Entwicklung von MooN

Dieses Kapitel beschäftigt sich mit der Entwicklung von MooN während des zweiten Semesters. Der Anfang bietet einen Einblick in die Diskussion und Entscheidung darüber, unter welche Lizenz MooN gestellt werden soll. Danach werden die Programme und Methoden, die bei der Implementierung eingesetzt wurden, vorgestellt, gefolgt vom Zeitplan der Implementierung. Abschließend werden der Aufbau und die Bedienphilosophie der graphischen Oberfläche von MooN dargestellt.

2.1 Lizenz

Zum Abschluss der Implementierungsphase stellte sich die Frage, wie der entstandene Quellcode am besten zu lizenzieren ist. Letztendlich konnten sich die Projektteilnehmer einstimmig auf die „GNU General Public License“ (GPL) (siehe [Fre03]) einigen. In diesem Abschnitt sollen die Überlegungen, die hinter dieser Entscheidung standen, aufgezeigt werden.

Anforderungen

Die Anforderungen, die eine Lizenz zu erfüllen hat, ergaben sich im Laufe der Diskussion um den Anwenderkreis der Software und aus der Frage der Weiternutzung nach dem Ende der PG.

Aufgrund der Tatsache, dass die in der PG entwickelten Inhalte den Teilnehmern gehören und der Quellcode einen wesentlichen Teil davon ausmacht, ergab sich die grundsätzliche Notwendigkeit durch irgendeine Art der Lizenzierung zunächst die Urheberschaft zu sichern.

Die entwickelte Software ist eindeutig auf den wissenschaftlichen und weniger auf den kommerziellen Bereich ausgerichtet. Daher ist es nicht notwendig, die Nutzung der Software einzuschränken oder die (binäre) Erweiterung, z. B. durch Plug-Ins, in irgendeiner bestimmten Weise zu beschränken, um damit monetäre Interessen zu verfolgen. Der erhoffte wissenschaftliche Nutzen wird sich nicht nur durch die Anwendung des Tools einstellen, sondern vor allem auch dadurch, dass das Framework erweitert und das Design studiert wird. Dazu ist die Verfüg- und die Abänderbarkeit der Quellen sicherlich von Vorteil.

In diesem Zusammenhang wurde ebenfalls mehrheitlich der Meinung entsprochen, dass die im Zeitrahmen der Projekgruppe entwickelte Software keineswegs ein vollständiges und abgeschlossenes Produkt ist, sondern dass die enwickelten Ideen und Konzepte oftmals nur prinzipiell umgesetzt bzw. prototypisch realisiert werden konnten. Es war also durchaus im Interesse der Teilnehmer, dass Dritte die Möglichkeit zur Verbesserung haben, solange sie diese Verbesserungen in die ursprüngliche Software zurückfließen lassen.

Lizenzarten

Um die Entscheidung über die Lizenzierung in der Gruppe treffen zu können, wurde ein kurzer Vortrag zum Thema gehalten. Im wesentlichen wurden dabei Lizenzen in fünf verschiedene Gruppen eingeteilt (siehe [Fre02]):

- **Proprietär**
  Nutzung, Modifikation und Weitergabe sind beschränkt. Quellcode meist nicht oder nicht für die Allgemeinheit verfügbar. Diese Art von Lizenz dient den meist monetären Interessen der Firma, die das Produkt zur Verfügung stellt.

- **Shareware**
  Die Verbreitung der Software soll durch freies Weitergeben gefördert werden. Meistens ist die Nutzung nur innerhalb einer „Probierphase“ frei. Der Quellcode ist meistens nicht verfügbar.

- **Freeware**
  Dies ist die kostenlose Verbreitung von Software (oft als Basisversionen eines größeren Paketes). Modifikationen sind nicht erlaubt.

- **Public Domain**
  Das Programm verzichtet auf eine Lizenzierung; es gehört der „Öffentlichkeit“. Manchmal wird dabei zwischen dem Status der Software in Binärform und dem der Quellen ein Unterschied gemacht.

- **Open Source**
  Die Quellen des Programms sind offen. Nutzung, Weitergabe und Modifikation sind meistens erlaubt. Hier gibt es Unterschiede bezüglich der Vorgaben bzgl. Lizenzierung von Modifikationen. Die GPL erfordert z. B. das alle Modifikationen an GPL-lizensiertem Code ebenfalls unter die GPL gestellt werden, während z. B. die BSD Lizenz keine Vorgaben diesbezüglich macht.

Entscheidung für die GPL

Die Entscheidung für die GPL ergab sich daraus, dass eine Open Source Lizenz den Anforderungen der PG am nächsten kam und das sie sehr verbreitet unter den Open Source Lizenzen ist (z. B. Linux). Außerdem wurde begrüßt, dass evtl. von Dritten vorgenommene Erweiterungen ebenfalls wieder unter die GPL gestellt werden müssen, so dass die Zukunft des Quellcodes von den Teilnehmern der PG weiterverfolgt werden kann. Daneben fehlten juristische Kenntnisse, um eine eigene Lizenz zu formulieren. Free- und Shareware-Lizenzen hätten wahrscheinlich ebenfalls zu einer Verbreitung der Software geführt, dies hätte sich aber nicht auf den Quellcode ausgedehnt. Da die Projektgruppe ihre Urheberschaft an dem Code kenntlich machen wollte, fiel auch die Alternative der „Public Domain“ weg.

2.2 Verwendete Tools

2.2.1 XML

Unsere Spezifikation für MooN erforderte es, das Speichern und Lesen von stukturierter Information möglich zu machen (siehe 3.2). Es stellte sich die Frage, wie bzw. in welchem Format eine Laufkonfiguration gespeichert oder gelesen werden kann.

Wir haben uns letztendlich für eine in der Markup-Sprache XML formulierte Spezifikation entschieden. Dazu wurden ein Parser und so genannte XML-Schemata zur Validierung der Konfiguration gebraucht. Ein XML-

Parser ist ein Programm, das eine XML-Datei einmal durchliest und daraus ein Baum von Objekten mit deren Werten und Attributen in einer bestimmten Programmiersprache (in unserem Fall Java) zusammenstellt.

Es gibt sehr viele Parser auf dem Markt. Sie unterscheiden sich nicht nur in der Programmiersprache, für die sie entwickelt wurden, sondern auch in der Geschwindigkeit, sowie darin, ob sie validieren oder nicht und ähnlichem. Ein nichtvalidierender Parser testet nur, ob das Dokument wohlgeformt ist. Ein validierender überprüft zusätzlich, ob das Dokument die so genannte „Document Type Definition“ (Regeln-Dokumenttyp-Definition, kurz DTD) einhält. In der DTD steht vor allem, was für Elemente und Attribute in solch einem Dokument vorkommen und was für Werte diese Elemente annehmen können.

Unsere Wahl fiel schließlich auf den „JDOM“-Parser, weil er sehr leicht zu bedienen ist. Zur Validierung der XML-Dateien mussten wir noch einen zusätzlichen Parser, „Xerces“ von IBM/Apache einsetzen.

2.2.2 Log4J

Grundlagen

Log4J ist Teil des Apache Jakarta Projektes (siehe [HKS04]). Es ist ein Werkzeug für Java, um Daten formatiert in Dateien oder auf dem Bildschirm auszugeben. Die Ausgaben sind hierarchisch nach Log-Leveln geordnet, welche über eine Konfigurationsdatei eingestellt werden können. Log4J besteht aus drei wesentlichen Komponenten: Logger, Appender und Layout. Zur eingehenderen Beschreibung von Log4J sei auf die Ausführungen in [Gül03] verwiesen.

Ein Logger ist ein Objekt, an das die auszugebenden Meldungen übergeben werden. Jedem Logger und jeder Ausgabe wird ein Log-Level zugeordnet. Für die fünf Log-Level gilt folgende hierarchische Rangfolge:

$$DEBUG < INFO < WARN < ERROR < FATAL$$

Eine Ausgabe wird nur an den Logger übermittelt und von diesem weitergeleitet, falls ihr Level mindestens so hoch ist wie der des Loggers. Wenn z. B. beim Logger der Log-Level auf $WARN$ gesetzt wurde, werden keine Ausgaben geschrieben, die mit $DEBUG$ und $INFO$ gekennzeichnet sind. Falls dem Logger kein Level zugeordnet wurde, erbt er den Level seines nächsten Vorfahrens.

Appender geben die Meldungen eines Loggers aus. Das Format der Ausgabe und der Ort werden außerhalb des Codes über eine Konfigurationsdatei eingestellt, die vom Java-Programm eingelesen wird. An einen Logger können mehrere Appender angehängt werden. Ein Logger erbt normalerweise alle Appender seines nächsten Vorfahrens. Dieses Verhalten kann jedoch für einen Logger ausgeschaltet werden, sodass dieser, wie auch seine Nachfahren, mit einer neuen Liste von Appendern beginnt.

Das Layout wird an den Appender angehängt und bestimmt das Format der auszugebenden Daten. Das Layout kann eine einfache Ausgabe oder eine HTML-Tabelle sein. Die Layouts werden den Appendern über die Konfigurationsdatei zugeordnet.

Anwendung in MooN

Alle Kommandozeilenausgaben, die während der Entwicklung erzeugt wurden, wurden mit Log4J ausgegeben. Dies bietet den großen Vorteil, die Ausgaben gemäß ihrer Wichtigkeit in Kategorien einteilen zu können. Einfachere Systemausgaben kann man mit $DEBUG$ oder $INFO$ kennzeichnen. Ausgaben, die auf kritische Situationen hinweisen, werden in die Kategorie $WARN$ eingeordnet. Ausgaben zu Fehlern gehören in die Kategorie $ERROR$ und schwere Systemfehler in die Kategorie $FATAL$.

Nun hat der Benutzer die Möglichkeit, über die Konfigurationsdatei den Level des Loggers so einzustellen, dass dieser nur Meldungen ab einer bestimmten Wichtigkeit ausgibt. Dann können, wenn das Programm Fehlermeldungen erzeugt, diese gesondert betrachtet werden, ohne gleichzeitig die unwichtigeren Debug-, Info- und Warn-Mitteilungen anschauen zu müssen. Es besteht also die Möglichkeit, die Ausgaben des Programms an- und abzuschalten, ohne sie ständig aus dem Code zu entfernen und wieder einzufügen. Es muss lediglich die Konfigurationsdatei abgeändert werden.

Sollen nur die Ausgaben aus einem bestimmten Teil des Programms betrachtet werden, so hat bietet sich die Möglichkeit, einen Logger auszuwählen und nur seine Ausgaben in eine selbst gewählte Datei umzuleiten. Soll die Ausgabe in einem anderen Format, z. B. in einer Tabelle, angezeigt werden, so bietet Log4J auch dafür Gelegenheit.

Insgesamt sorgte die Anwendung von Log4J in der Entwicklung von MooN für eine übersichtlichere und komfortablere Ausgabe. Im fertigen Produkt wird der Teil, der Log4J verwendet, ausgeschaltet, aber enthalten bleiben. Falls MooN nach der Projektgruppe weiterentwickelt wird, kann dieser Teil wieder aktiviert und weiterverwendet werden.

2.2.3 JUnit

JUnit (siehe [Obj04b]) ist ein Java Framework für „Unit testing“. Dies ist eine Testmethode, die sich aus dem Vorgehensmodell des *extreme programming* (XP) entwickelt hat. Sie setzt besonders stark auf die schrittweise Implementierung von Funktionalität und der damit oft einhergehenden Refaktorisierung von Code. Um sicherzustellen, dass dabei bisherige Funktionalität nicht verloren geht, ergibt sich die Notwendigkeit von Tests, die automatisiert ausführbar und feingranular genug sind, um die kleinen Iterationsschritte der Implementierung abzudecken.

Grundlagen

Unit Tests werden vor der eigentlichen Implementierung geschrieben und testen üblicherweise die Funktionalität einer Methode. JUnit ist ein Framework, das Unit Testing in Java vereinfachen soll. Dazu stellt JUnit einige Klassen und Interfaces zum Schreiben von Tests zu Verfügung. Die wichtigsten (siehe [Obj04a]) sind:

- `JUnit.framework.TestCase`
  Abstrakte Klasse, deren Implementierungen Sammlungen von Tests gegen einen Satz von Objekten darstellen. Benötigte Testobjekte können in der Methode `setUp()` initialisiert bzw. mit `tearDown()` vernichtet werden. Einzelne Tests können vom Entwickler beliebig hinzugefügt werden.

- `JUnit.framework.TestSuite`
  Kapselung von `TestCases` zu größeren Einheiten. JUnit führt eigentlich (s. u.) immer nur Instanzen von `TestSuite` aus.

- `JUnit.framework.Assert`
  Eine Sammlung von „assert“ Methoden (Assertion, engl. „Behauptung“, „Versicherung“), die Vergleiche von produzierten mit erwarteten Testergebnissen vornehmen, z. B.
  `Assert.assertEquals(obj1, obj2)`
  oder `Assert.assertNotNull(obj)`.

Ein Entwickler schreibt seine Tests als Implementierung von `TestCase` und fügt für die Objekte benötigte Tests hinzu. Der Code wird erst dann als funktionsfähig betrachtet, wenn alle Tests komplett erfolgreich absolviert

sind. Werden neue Features hinzugefügt, werden die betreffenden `TestCases` erweitert und es wird sichergestellt, dass Änderungen am Code alle bisher funktionierenden Tests erfolgreich durchlaufen. Es entspricht der Philosophie hinter Unit Testing, diese häufig durchzuführen (nach dem Motto: „Code a little, test a little“).

Vorteile dieser Methode (vgl. [BG98]) sind

- die Verfügbarkeit von Testtreibern von Anfang an,
- die Möglichkeit, sicherzustellen, dass Codeänderungen keine bisherige Funktionalität zerstören
- und dass die Entwickler „in Schnittstellen“ denken und nicht „in Implementierung“.

Letzteres gilt, da sie sich vor allem darauf konzentrieren, die erwartete Funktionalität einer Methode sicherzustellen und dabei nicht auf die Interna der Methode schauen.

JUnit ist mittlerweile in vielen IDEs (Eclipse, Netbeans, JBuilder) integriert und vereinfacht damit die Entwicklung und automatisierte Ausführung von JUnit Tests noch weiter.

Erfahrungen in der PG

Die in der PG entwickelte Software „MooN“ teilt sich in zwei große Bereiche: Die Heuristiken und die Ausführungsumgebung. Da die Heuristiken alle randomisierte Elemente haben, war ein automatisches Testen mit exakt vorprogrammierten Tests nur für einige Teile des Codes denkbar. Im Bereich der Ausführungsumgebung boten sich wesentlich mehr Möglichkeiten zum Einsatz von JUnit.

Zum Beginn der Implementierungsphase wurde das JUnit Framework vorgestellt und diskutiert. Dabei stieß vor allem die Idee, Tests vor und während der eigentlichen Implementierung zu schreiben, auf Skepsis. Daher wurde der Einsatz von JUnit zum Testen in das Ermessen der jeweiligen Entwickler gestellt. In den Fällen, in denen es zum Einsatz kam, wurden jedoch gute Erfahrungen gemacht. Vor allem bei systemnahen Klassen wie dem Plug-In-Management konnten so schnell Fehler ausfindig gemacht werden, die sich etwa auf einem Linuxsystem nicht zeigten, dafür dann aber auf dem Windowssystem eines anderen Entwicklers. Auch konnten selbst größere Änderungen zu einem relativ späten Zeitpunkt (wie z. B. die Einführung

einer Plug-In-Überprüfung beim Laden derselben) vorgenommen werden, ohne befürchten zu müssen, dass die bisherige Funktionalität in Mitleidenschaft gezogen worden wäre. Ein weiterer Vorteil war es, dass man neben der Dokumentation auch bereits Beispiele für die Nutzung von bestimmen Methoden anderer Entwickler in Form der Tests hatte. Die schnelle Eingrenzung von Fehlern auf bestimmte Codeteile machte es möglich, eindeutige Fehlermeldungen an andere Entwickler zu geben, was die Behebung oftmals stark beschleunigte.

Der Einsatz von JUnit war somit erfolgreich und hätte durchaus als fester Bestandteil in die Implementierung mit aufgenommen werden können.

2.2.4 phpBugTracker

Um Fehler von MooN während der Implementierungsphase zu verwalten, setzten wir das Tool phpBugTracker ein. Eine Unterstützung durch geeignete Software zu diesem Zweck schien uns ratsam, da bei zwölf parallel arbeitenden Entwicklern durch eine uneinheitliche Dokumentierung und Behandlung von Fehlern größere Probleme auf Grund von fehlender oder mehrfacher Bearbeitung zu erwarten waren. Neben dem Erkennen und Dokumentieren von Fehlern sollte weiterhin eine übersichtliche Verfolgung über den Status ihrer Bearbeitung, sowie der zuständigen Entwickler, möglich sein.

Diese Anforderungen ließen unsere Wahl auf den phpBugTracker fallen. Er ist ein PHP-basiertes Tool, das den Vorteil hatte, auf dem Server ohne großen Administrationsaufwand installiert werden zu können. Neben der Erstellung von Fehlerberichten gab es hier die Möglichkeit, automatisch Mails an die Entwickler zu versenden, denen neue Fehler zugeordnet worden waren. Auch die Einordnung der Fehler nach gewissen Kriterien, welche zum Teil aus Dropdown-Menüs ausgewählt werden konnten, war möglich.

Zu Beginn der Implementierungsphase wurde das Tool jedoch nicht so intensiv genutzt wie im Vorfeld erwartet. Dies lag zum einen an der nicht perfekt arbeitenden Mailfunktion, durch die nicht alle Mails sofort zugestellt wurden, zum anderen an den teilweise zu detaillierten Möglichkeiten der Fehlerbeschreibung. Abgesehen von den technischen Schwierigkeiten aber war der Einsatz im Nachhinein betrachtet noch etwas verfrüht, denn etwa ab der zweiten Hälfte der Implementierung stiegen die eingetragenen Fehler stetig an. Dies lag an den etwas geänderten Aufgaben, da Code nun nicht nur neu erstellt, sondern auch stark genutzt und getestet wurde und somit viele Fehler zu Tage traten. Die Dokumentation der Fehler und Zuordnung zu den

entsprechenden Entwicklern erwies sich dabei als äußerst hilfreich. Dies geschah zum einen, um keine Fehler wieder zu vergessen, die zum Teil nur durch längere Klickfolgen reproduzierbar waren. Andererseits sollte die Zuordnung von Fehlern die Entwickler zur Beseitigung derselben motivieren.

2.3 Implementierung

2.3.1 Konzept

Aufteilung in Packages

Zur Aufteilung der Implementierungsaufgaben wurden aus dem Klassendiagramm des ersten Semesters vier Pakete abgeleitet: `Core`, `Interfaces`, `Plug-Ins` und `GUI`.

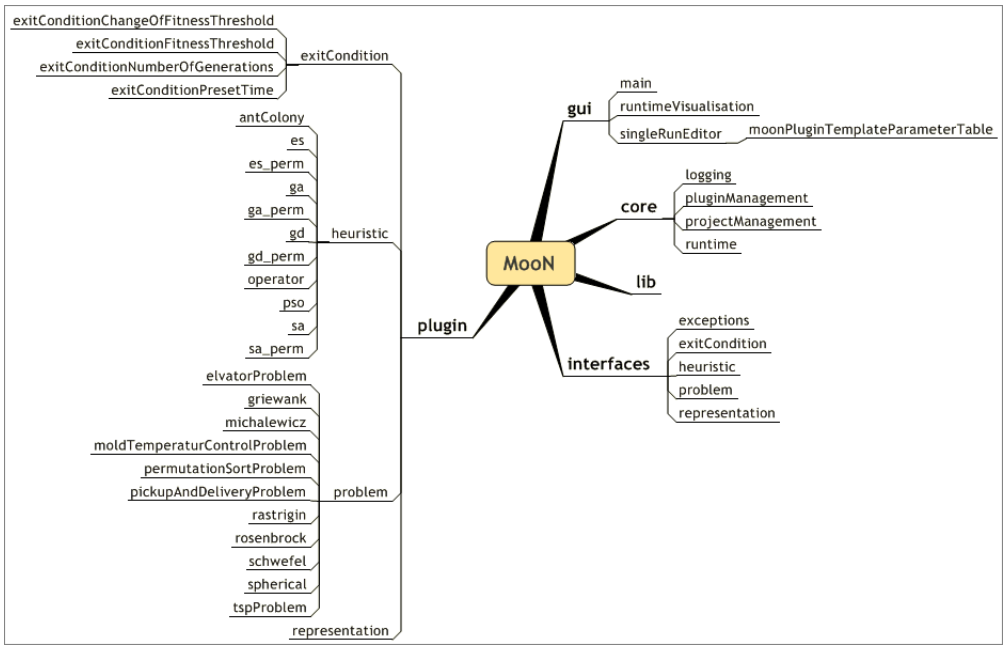

Abbildung 2.1: Paketstruktur von MooN.

Core

`Core` ist der Kern des Programms, welcher die Heuristiken und Probleme miteinander interagieren lässt.

Im Projekt-Management werden Gesamtläufe als XML-Dateien verwaltet. Zur Erzeugung des Gesamtlaufs zwecks Manipulation oder Ausführung wird die zugehörige XML-Datei ausgelesen. Des Weiteren stellt das Projekt-Management Informationen bereit, welche Plug-Ins, d. h. Heuristiken, Probleme und Abbruchbedingungen, zur Verfügung stehen sowie über deren Parameter. Die Plug-Ins sind für die Erstellung von Gesamtläufen notwendig.

Das Plug-In-Management ermöglicht die Installation neuer Plug-Ins. Dazu muss dem Plug-In-Management eine Java-Archiv-Datei (.jar-File) übergeben werden, die die zum Plug-In gehörenden Dateien enthält. Das Plug-In-Management speichert das Plug-In ab und zeigt ab diesem Zeitpunkt dessen Verfügbarkeit an. Es erkennt den Typ eines Plug-Ins und kann auch Informationen über Parameter aus dem Plug-In auslesen. Bei einer Aufforderung durch das Projekt-Management gibt das Plug-In-Management die `class`-Objekte des Plug-Ins aus.

Die Runtime kümmert sich um die Abarbeitung der Einzelläufe des Gesamtlaufs. Sie erhält die fertig instanziierten Plug-Ins vom Projekt-Management und initialisiert diese.

Das Logging umfasst die Ausgabe von Systemmitteilungen und Laufzeitwerten. Letztere sind Daten, die zur Laufzeit von der Heuristik erzeugt werden und später von externen Werkzeugen weiterverarbeitet werden können.

Interfaces

Es gibt mehrere Interfaces innerhalb des Klassendiagramms von MooN. Die Interfaces *Heuristic*, *Problem* und *ExitCondition* sind die Schnittstellen der Plug-Ins zum Core. Zusätzlich gibt es die Interfaces *Representation* und *Exceptions*.

GUI

Die GUI bietet als graphische Oberfläche den einfachsten Weg, MooN zu bedienen. Eine Alternative hierzu bietet die Bedienung über die Konsole. In der GUI kann leicht ein Gesamtlauf erstellt werden, der aus mehreren Einzelläufen besteht. Zu jedem Einzellauf können je ein Problem, eine Heuristik und eine Abbruchbedingung gewählt und deren voreingestellte Parameter

geändert werden. Wird ein Gesamtlauf ausgeführt, so werden die zuvor ausgewählten Daten in eine Datei geschrieben und die Optimierung der Läufe durch die Laufzeitvisualisierung der GUI angezeigt.

Plug-Ins

Heuristiken, Probleme und Abbruchbedingungen wurden in MooN als Plug-Ins realisiert, um sie dynamisch in das fertige Tool nachladen zu können. Viele Plug-Ins, wie beispielsweise die Heuristiken PSO, ACO und das Problem TSP, werden als Plug-Ins schon in MooN enthalten sein. Das Paket `Representation` ist für Klassen gedacht, die lediglich Datenstrukturen zur Kommunikation zwischen den einzelnen Plug-Ins untereinander enthalten.

Lib

In das Paket `Lib` sollen Bibliotheken von Dritt-Anbietern kommen wie z. B. der XML-Parser JDom von Apache.

2.3.2 Arbeiten in Teilgruppen

Planung

In der 43. Kalenderwoche teilten wir unsere Implementierungsaufgaben auf die vier oben beschriebenen Pakete ein. Zur Erstellung von `Core`, `Interfaces` und `Plugins` bildeten wir gleichzeitig Untergruppen. Die Implementierung der GUI verschoben wir auf später.

		45	46	47	48	49	50	51	1	2	3	4	5	6
Core	Core 0.1: Zusammenspiel aller Komponenten	■												
	Core 0.2: Logger / Abspeichern		■	■	■									
	Core 0.3 externe Tools, Visualisierung, skriptb. Fkt.			■	■	■								
Plugins	Heuristiken 1: GA, ES, PSO	■												
	Heuristiken 2: weitere Heuristiken		■	■	■									
	Probleme 1: einfache Testfunktionen	■												
	Probleme 2: Temperierbohrungsproblem	■	■											
	Probleme 3: weitere (VRP, Fahrstuhlproblem,...)			■	■	■								
GUI	graphische Benutzeroberfläche				■	■	■	■						

Abbildung 2.2: Planung der Implementierung.

In der 45. Kalenderwoche wurde ein Plan für die weitere Implementierung erstellt. Danach sollte das Produkt MooN noch im Jahr 2003 fertig gestellt werden. Somit verblieben im Jahr 2004 sechs Wochen, um die Funktionen von MooN zu testen, die Heuristiken zu untersuchen, den Endvortrag vorzubereiten und um den Endbericht, das Benutzerhandbuch und das Entwicklerhandbuch zu schreiben.

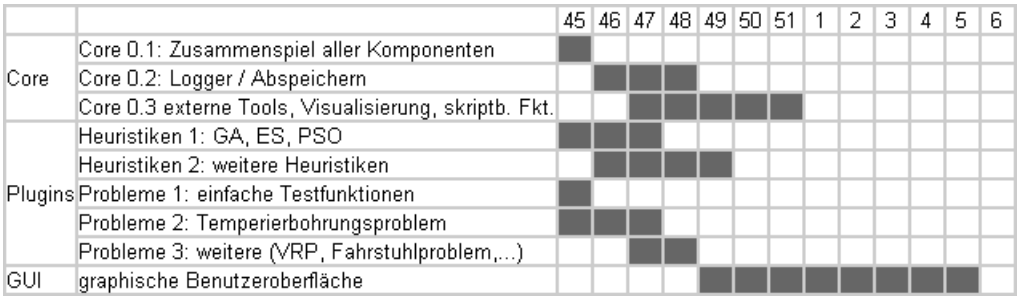

Abbildung 2.3: Tatsächliche Durchführung der Implementierung.

Core

Da `Core` ein großer Teil des Programms ist, wurde er Stufe um Stufe erstellt. Insgesamt gab es 3 Stufen.

Core der Stufe eins erlaubt das Zusammenspiel des `Core`, als Steuerung eines Gesamtlaufs, mit einem Problem, einer Heuristik und einer Abbruchbedingung. Dazu wird eine XML-Datei ausgelesen und der darin beschriebene Gesamtlauf als Objekt erzeugt. Die Plug-Ins, welche vom Gesamtlauf benötigt werden, wurden vorher aus jar-Dateien heraus installiert und nun instanziiert. Dann wird der Gesamtlauf ausgeführt.

Core der Stufe zwei kann zusätzlich die erzeugten Laufzeitdaten in eine Datei schreiben. Außerdem können Läufe erstellt, abgeändert und gespeichert werden.

In der dritten Stufe des Core sind die externen Tools `R` und `Gnuplot` zur Analyse und Visualisierung anbindbar. Zusätzlich kann MooN nicht nur über die graphische Oberfläche, sondern auch per Skript über die Konsole gesteuert werden. Das Skript eines Laufes ist bei MooN die XML-Datei, in der der Gesamtlauf codiert ist.

Stufe eins ist, wie der Zeitplan es vorsah, in der 45. Kalenderwoche realisiert worden. Grundlage für Stufe zwei ist die Speicherung der Konfigura-

tion eines Gesamtlaufes als XML-Datei, in der auch später Änderungen der Konfiguration gespeichert werden. Die Speicherung als XML-Datei und das Loggen der wichtigsten Daten eines Gesamtlaufes in eine Datei wurde bis zur 48. Kalenderwoche realisiert. Die Skriptsteuerung der Stufe drei bestand einerseits in der Einrichtung der XML-Datei in Stufe 2. Außerdem wurde in der 48. Kalenderwoche die Klasse `Moon` implementiert, die von der Kommandozeile aus aufgerufen werden kann, um Läufe zu starten und Plug-Ins zu installieren. Die Anbindung der externen Tools `R` und `Gnuplot` zur Analyse und Visualisierung der erzeugten Daten entstand erst in der 51. Kalenderwoche. Nach der 51. Kalenderwoche wurden die bisher realisierten Features von MooN weiter verfeinert.

Interfaces

Die Interfaces von MooN wurden in der 43. Woche implementiert. Danach löste sich die Gruppe auf und ist somit in der Planung des weiteren Vorgehens ab der 45. Woche nicht mehr enthalten.

Plug-Ins

Die ersten Heuristiken, einfache Testfunktionen und eine Abbruchbedingung wurden in der 45. Kalenderwoche implementiert. GA, ES und PSO wurden bis zur 47., alle weiteren Heuristiken bis zur 49. Kalenderwoche fertig gestellt. Bis zur 48. Kalenderwoche wurden die Probleme implementiert und in der 49. Woche gab es bereits drei von vier Abbruchbedingungen als Plug-Ins. Somit waren, wie es auch geplant war, nach der 49. Woche genügend Plug-Ins für MooN vorhanden. Ab der 50. Woche fand die Erweiterung der Heuristiken statt, um auch auf Permutationen optimieren zu können. Zusätzlich wurden ACO, TSP und PAD implementiert.

GUI

Die Erstellung der GUI begann erst spät, da das Zusammenspiel von Core und Plug-Ins für wichtiger erachtet wurde und die GUI dafür nicht notwendig war. Leider dauerte die Implementierung der GUI länger als geplant. Daher konnten zwar die Experimente auf den Heuristiken und der Abschlussvortrag bis zur 6. Woche abgeschlossen werden, aber die Frist für die Handbücher und den Endbericht mussten auf die 8. Woche verschoben werden.

2.4 GUI

Ziel des ersten Schritts der GUI-Erstellung war es, zunächst eine Art XML-Viewer zu schaffen, in dem die bereits fertig erstellten XML-Dateien geladen und angezeigt werden konnten. Ein wichtiges Augenmerk war hierbei die spätere Erweiterbarkeit der GUI zu einem vollständigen Tool. Es zeigte sich schnell, dass eine Trennung zwischen `CompleteRun` als geladenem Objekt und `SingleRun` als einzelner Teil dessen sinnvoll ist. Der `CompleteRun` sollte nur in einer Übersicht dargestellt werden. Zur Bearbeitung sollten einzelne `SingleRuns` ausgewählt und in einer gesonderten Maske editiert werden können. Als praktikabel erwies sich die Darstellung des `CompleteRuns` in einer einfachen Tabelle, in der nur die Namen der einzelnen Komponenten angezeigt werden.

Die Darstellung des `SingleRun` ist hingegen sehr ausführlich. Die einzelnen Komponenten sollten möglichst komfortabel editiert werden können. Der Übersichtlichkeit halber werden die einzelnen Komponenten des `SingleRuns` in getrennten GUI-Elementen dargestellt, die in einer Reiterkartei ausgewählt werden können. Ein Screenshot der resultierenden GUI ist in 2.4 abgebildet.

2.4.1 Der `CompleteRun`

Der `CompleteRun` besteht aus zwei Teilen:

- der Beschreibung und
- den einzelnen `SingleRuns`.

Das Feld für die Beschreibung befindet sich oben links. Es steht ein mehrzeiliges Textfeld zur Eingabe einer Erläuterung zur Verfügung. Im rechten oberen Teil steht die Liste mit den `SingleRuns`. In dieser wird der `SingleRun` nur durch seine Beschreibung und den Namen der drei Plug-Ins, aus welchen er besteht, repräsentiert. Wird ein `SingleRun` ausgewählt, so kann dieser im unteren Teil bearbeitet werden (siehe auch unten).

In der Liste kann der Benutzer nicht nur vorhandene `SingleRuns` zur Bearbeitung auswählen. Einzelne `SingleRuns` können mittels der dritten Buttonreihe oder des Untermenüs „Single run“ hinzugefügt oder gelöscht, sowie in ihrer Reihenfolge geändert und kopiert werden. Ist ein `CompleteRun` vollständig, so kann er ausgeführt und wieder beendet werden. Dafür steht die zweite Buttonreihe und der Menüpunkt „Optimization“ zur Verfügung.

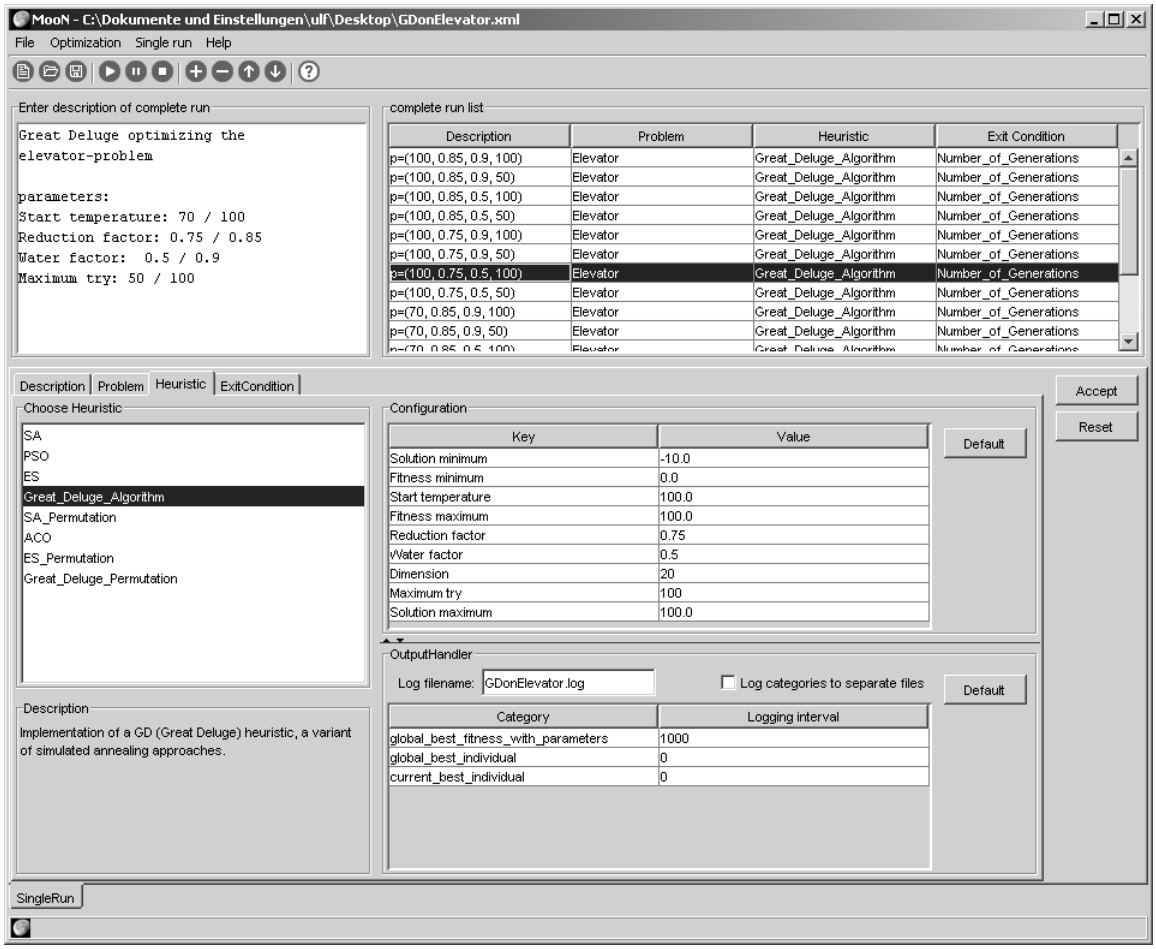

Abbildung 2.4: Graphische Benutzeroberfläche von MooN.

2.4.2 Der `SingleRun`

Die Konfiguration eines `SingleRuns` ist etwas aufwändiger. Er besteht aus

- der Beschreibung und der Anzahl der Wiederholungen,
- drei Plug-Ins, nämlich einer Heuristik, einem Problem und einer Exit-Condition sowie
- dem OutputHandler.

Die Beschreibung und die Anzahl der Wiederholungen

Im ersten Reiter des von uns „SingleRunEditor“ genannten GUI-Elementes stehen zwei Optionen zur Verfügung. Es sind ein Feld zur Eingabe der Wiederholungen des `SingleRuns` und ein großes Textfeld für eine aussagekräftige Beschreibung vorhanden. Die Anordnung lässt Spielraum für eine spätere Erweiterung von MooN um weitere Funktionalitäten, einen `SingleRun` betreffend. So könnte hier beispielsweise ein Feld für eine Skriptsteuerung eingefügt werden, in dem besondere Aktionen zur Ausführung vor und nach der Durchführung des `SingleRuns` bestimmt werden.

Die Plug-Ins

Ein `SingleRun` besteht aus drei Plug-Ins (Heuristik, Problem und Exit-Condition). Eine ausführliche Beschreibung dieser Thematik erfolgt in den Abschnitten 3.1 und 3.2. Die Konfiguration der drei Plug-In-Typen kann aufwändig sein, weshalb jedem Typ eine eigene Seite im SingleRunEditor zugeteilt wurde. Sie sind allerdings nahezu identisch aufgebaut, was in ihrem ähnlichen internen Aufbau begründet ist.

Zunächst kann man ein Plug-In aus einer Übersicht über alle installierten Plug-Ins des enstsprechenden Typs auswählen. Daraufhin wird eine Beschreibung des Plug-Ins angezeigt, sowie dessen Parameter. Auf eine einfache und sichere Konfigurierbarkeit der Parameter wurde besonders geachtet. So ist es in MooN schwer, einen Parameter falsch zu konfigurieren. Ist z. B. der Typ `double` gefordert, so kann kein beliebiger Text eingegeben werden, sondern nur Zeichen (Ziffern, `.` und `E`), die einen gültigen `double`-Wert ergeben. Eine ausführliche Beschreibung der Parameter und ihrer Bedeutung ist als ToolTip realisiert. Der Benutzer kann so komfortabel und sicher vor Fehlern das Plug-In konfigurieren, selbst wenn er die Bedeutung eines Parameters

nicht kennt. Standardmäßig sind alle Plug-Ins mit so genannten „Default-Parametern“ konfiguriert, sodass Plug-Ins sofort getestet werden können. Mit dem „Default“-Knopf können die Parameterwerte auf die Ursprungswerte zurückgesetzt werden.

Der OutputHandler

Auf der Seite der Heuristik taucht ein weiters Feld auf, dass es dem Benutzer ermöglicht, die Ausgabe des Zustandes einer Heuristik in einer oder mehreren Dateien zu speichern. Da das zuständige Konstrukt (der „Output-Handler“, siehe Abschnitt 3.8) primär von der Heuristik abhängt, wurde die Konfiguration desselben dort untergebracht. Verschiedene Heuristiken loggen verschiedene Kategorien von Werten, sodass diese Anordnung sinnvoll erschien.

2.4.3 Die Laufzeitvisualisierung

Als besonderes Feature kann während des Optimierungsprozesses eine Grafik angezeigt werden, die den momentanen Zustand der Optimierung visualisiert. Diese wird im Abschnitt 3.9 näher beschrieben.

Kapitel 3

Aufbau von MooN

Dieses Kapitel beschreibt den konzeptionellen und strukturellen Aufbau von MooN. Im ersten Teil soll ein Überblick gegeben werden. Dazu wird das Hauptkonzept von MooN, die „Plug-Ins“, beschrieben. Anschließend wird das Zusammenspiel der Plug-Ins, also der Ablauf von Optimierungen in MooN ausführlich erläutert.

Im zweiten Teil dieses Kapitels geht es um die von uns entwickelten Plug-Ins selbst. Zunächst werden Grundkonzepte wie die verwendeten Problemrepräsentationen und Operatoren zusammengefasst. Dann gibt es eine kurze Übersicht zum Aufbau der Beschreibungen der einzelnen Plug-Ins. Darauf werden die von uns implementierten Plug-Ins ausführlich vorgestellt, zunächst die Heuristiken, dann die Probleme und schließlich die Abbruchbedingungen.

Der dritte Teil befasst sich mit den von MooN erzeugten Ergebnissen. Zuerst werden die von MooN erzeugten Daten spezifiziert. Das zugehörige Konzept des „OutputHandler“ wird erklärt. Dann wird die in der GUI von MooN integrierte Laufzeitvisualisierung von Laufdaten vorgestellt. Abschließend werden die Fragen, wie Ergebnisdaten statistisch ausgewertet werden können, sowie welche Visualisierungsmöglichkeiten sich daraus ergeben, erörtert.

3.1 Struktur und Plug-Ins

Übersicht

Eine der wichtigsten Designentscheidungen, die die PG im ersten Semester für MooN traf, war, einen hochgradig modularen Aufbau für Probleme und Heuristiken zu wählen, der auf Plug-Ins basiert. Für eine detailliertere Sicht auf den Entscheidungsprozess sei auf den Zwischenbericht [BBB+03] verwiesen.

MooN stellt ein Tool dar, das zwei Hauptaufgaben wahrnimmt:

- Die Verwaltung von Heuristiken, Problemen und Abbruchbedingungen als Plug-Ins.
- Die Verwaltung und Durchführungen von Optimiervorgängen, so genannten Läufen.

Plug-Ins sind in sich geschlossene Einheiten, die als Module in die Programminstanz eingebunden und dann in Läufen benutzt werden können. Jedes Plug-In muss eine vom Interface `moon.interfaces.PluginTemplate` abgeleitete Klasse besitzen. Es gibt drei solcher Interfaces und damit drei Arten von Plug-Ins:

- Heuristiken, welche aus dem Interface *Heuristic* abgeleitet sind.
- Probleme, die aus dem Interface *Problem* abgeleitet sind.
- Abbruchbedingungen, welche aus dem Interface *ExitCondition* abgeleitet sind.

Um die Wiederverwendbarkeit von Code zu gewährleisten, stellt MooN den Java Sourcecode unter der GNU GPL zur Verfügung. Die Plug-Ins sind zwar monolithisch, gemeinsam verwendbare Klassen, wie etwa Mutationsoperatoren, können jedoch bibliothekartig zur Erstellung neuer Plug-Ins genutzt werden. Zudem ist es möglich, bestehende Codeteile wieder zu verwenden oder zu modifizieren.

Eine Übersicht der in MooN enthaltenen Plug-Ins erfolgt in den Kapiteln 3.5, 3.6 und 3.7.

Die Parameterschnittstelle

Plug-Ins sind in MooN in der Regel konfigurierbar, d. h. sie verfügen über Parameter. Um diese verändern, verwalten und speichern zu können, wurde eine einheitliche, flexible Parameterschnittstelle entworfen. Jedes Plug-In verfügt über Methoden, die mitteilen, welche Parameter es benötigt, welchen Typs sie sind und welche Standardbelegung („Defaultwert“) sie haben.

Eine Methode sorgt dafür, dass einem initialisierten Plug-In Parameter zugewiesen werden können. Das Zusammenspiel und die Abläufe während der Lebenszeit eines Plug-In-Objekts werden im Kapitel 3.2 beschrieben.

Erstellung eines Plug-Ins

Um ein neues Plug-In in MooN einzubinden, muss ein Entwickler zunächst den Source Code in Java programmieren. Hierbei kann er das Plug-In komplett neu erstellen oder bestehende Codeteile aus MooN verwenden, etwa um zwei Heuristiken zu einem Hybrid zu verbinden. Der Entwickler muss die in der MooN-API definierten Schnittstelle für den jeweiligen Plug-In-Typ einhalten.

Ein Plug-In wird als Java-Archive (`.jar`-Datei) realisiert. Der Entwickler fasst alle Klassen seines Plug-Ins so zusammen. Hierzu existieren diverse Tools. Zusätzlich muss das Archiv eine Datei `plugin.properties` enthalten, die auf die Hauptklasse des Pakets, das ist die Klasse die das oben erwähnte Interface implementiert, verweist. Außerdem sollte diese Datei noch eine Beschreibung des Plug-Ins enthalten, die in der GUI angezeigt wird.

Verwalten von Plug-Ins in MooN

Plug-Ins können mit der GUI in MooN geladen und verwaltet werden. Hierzu ist lediglich die `.jar`-Datei zu spezifizieren. MooN verwaltet eine Liste von gültigen Plug-Ins. So können auch Plug-Ins gelöscht werden, etwa um sie durch eine neuere Version zu ersetzen.

3.2 Ablaufsteuerung

Läufe und deren Struktur

Die Durchführung von Optimierungen gliedern wir in *Läufe*. Als *Einzellauf* bezeichnen wir dabei die einmalige Anwendung einer Heuristik auf ein Prob-

lem. Sie erzeugt dazu Lösungen für das Problem und erhält von diesem die zugehörigen Fitnesswerte zurück. Die Optimierung wird beim Erreichen der Abbruchbedingung beendet. Daten werden während des Einzellaufs durch den Outputhandler aufgezeichnet. Einzelläufe können zu einem *Gesamtlauf* zusammengefasst werden, da sie oft sinnvoll gruppiert werden können, z. B. wenn nur einzelne Parameter unterschiedlich eingestellt werden, die übrigen jedoch gleich bleiben.

Die XML-Datei

Jeder Gesamtlauf wird in einer hierarchisch aufgebauten XML-Datei gespeichert. Eine Datei steht hierbei für einen Gesamtlauf. Dieser beinhaltet eine Menge von Einzelläufen, deren Struktur sich in die vier Komponenten Heuristik, Problem, Abbruchbedingung und Outputhandler aufspaltet. Diese wiederum verwalten die Einstellungen ihrer jeweiligen Parameter. Die Struktur der XML-Datei ist durch eine Grammatik, d. h. ein XSD-Schema, festgelegt.

Laden eines Laufs

Üblicherweise wird als erster Schritt einer Optimierung mit Hilfe der GUI oder mit einem Editor eine XML-Datei erstellt, die den Lauf beschreibt. Diese wird dann von der Core-Komponente `XML-Reader` eingelesen und dabei zeitgleich auf syntaktische Korrektheit überprüft. In der XML-Struktur sind für die jeweiligen Plug-In-Elemente Heuristik, Problem und Abbruchbedingung Informationen darüber gespeichert, welche Klassen und Plug-Ins benötigt werden. Die konkret benötigten Plug-In-Objekte werden mit Hilfe des `Plug-In-Managements` unter Angabe von Plug-In-Name und Klasse initialisiert und mit den Parametern gefüllt, welche im XML-File spezifiziert sind. So entsteht ein Objekt, welches den Gesamtlauf repräsentiert und alle Einzelläufe mit den jeweiligen Plug-In-Objekten beinhaltet. Dieses Objekt wird dann an die `Runtime` weitergereicht.

Durchführung eines Laufs

Die `Runtime` verarbeitet nach und nach alle Einzelläufe, die zu dem ihr übergebenen Gesamtlauf gehören. Dabei werden drei verschiedene Phasen durchlaufen — die Initialisierung, die Durchführung und die Beendigung. Die Initialisierung wird bei allen dsrei beteiligten Plug-In-Objekten durchgeführt. Dabei werden z. B. Startpopulationen bei Heuristiken erzeugt, bei

Problemen Dateien eingelesen und vieles mehr. Es werden auch Verbindungen untereinander erzeugt, so dass die Heuristik beispielsweise Zugriff auf das Problem zur Fitnessberechnung hat. Nach der Initialisierung ist der Einzellauf startbereit, und es kann die erste Generation berechnet werden. Dabei überprüft die Heuristik durch Abfrage des Outputhandlers, ob Informationen geloggt werden sollen. Nach Abschluss jeder Generation wird überprüft, ob die Abbruchbedingung erreicht wurde. Solange dies nicht der Fall ist, werden weitere Generationen berechnet. In der letzten Phase werden die `cleanUp()`-Methoden der Plug-In-Objekte aufgerufen, die einen geordneten Abschluss des Objektlebens ermöglichen und hierfür bestehende Verbindungen beenden etc. Weiterhin werden verbleibende Puffer des Outputhandlers geleert und alle geöffneten Dateien geschlossen.

Starten und Stoppen

Während der Durchführung ist es möglich, einen Lauf anzuhalten. Dabei wird der Thread, in dem die Optimierung durchgeführt wird, mittels „sleep“ angehalten. Danach kann er entweder abgebrochen oder fortgesetzt werden.

Laufzeitvisualisierung

Parallel zur Durchführung im GUI-Modus ist es möglich, eine Laufzeitvisualisierung zuzuschalten. Sie wird im Abschnitt 3.9 näher beschrieben.

3.3 Repräsentation und Operatoren

Repräsentation

Um ein Optimierungsproblem einer Lösung zuzuführen, muss man es zunächst in geeigneter Weise beschreiben. Das Hauptproblem ist das Finden einer geeigneten Codierung einer Lösung. Sie wird auch Repräsentation genannt. MooN wurde so konzipiert, dass Heuristiken und Probleme, die die gleiche Repräsentation benutzen, miteinander kombinierbar sind. Exemplarisch haben wir uns bei der Implementierung der eigenen Plug-Ins auf zwei Varianten beschränkt - die Codierungen als reellwertiger Vektor bzw. als Permutation. Basisklassen für die Repräsentation als Boole'scher Vektor stehen dem Entwickler zur Verfügung, werden aber von unseren Plug-Ins nicht benutzt.

Eine Besonderheit stellt die Art und Weise dar, wie Permutationen in MooN codiert werden. Sie werden als Arrays gespeichert, welche die Zahlen 1 bis n enthalten. Diese definieren die bijektive Abbildung ϕ, wobei der Wert j an der Arrayposition i bedeutet, dass $\phi(i) = j$ ist. Diese Repräsentation ist zwar gut geeignet, um Mutationen zu unterstützen, bietet jedoch keinen Mechanismus, um Crossoveroperatoren effizient durchzuführen. Deshalb wurde die Darstellung erweitert. Jeder Arrayposition wird ein reeller Wert von 0 bis 1, der `sortValue`, zugeordnet. Die Crossoveroperatoren arbeiten in herkömmlicher Weise auf den `sortValues`. Jeder Operator muss sicherstellen, dass die Permutation am Ende nach aufsteigenden `sortValues` sortiert und auf das Intervall $[0; 1]$ skaliert ist.

Operatoren

Die Operatoren, die von reellwertigen Heuristiken benutzt werden, stammen aus der zugehörigen Literatur und wurden nach Bedarf geschrieben. Deshalb werden sie zusammen mit den jeweiligen Plug-Ins erläutert. Die Operatoren für die Permutationen hingegen wurden von PG-Teilnehmern nach Absprache mit einem Betreuer unabhängig von den Heuristiken entworfen. Später wurden die Crossover- und Mutationsvarianten den zu implementierenden Heuristiken zugeordnet. Da zudem viele Operatoren von mehreren Heuristiken benutzt werden, sind diese hier zusammenhängend beschrieben. Die folgenden Beschreibungen der Operatoren gehen von einer Darstellung der Permutation als Array aus, z. B. [1 2 3 4 5 6] für die identische Permutation auf sechs Elementen. Die Begriffe „Position“ und „Intervall“ werden hierbei intuitiv benutzt.

Die vier von uns implementierten Mutationsoperatoren sind folgende:

- `InterchangeMutationOfPermutation` ist die einfachste Mutation einer Permutation. Es werden zufällig gleichverteilt zwei Positionen ermittelt und deren Elemente vertauscht.
  Beispiel: [1 **2** 3 **4** 5 6] → [1 **4** 3 **2** 5 6]

- `ShiftMutationOfPermutation` verschiebt ein Element der Permutation. Ein Element und eine Zielposition werden zufällig gleichverteilt ausgewählt und das Element an diese Position bewegt. Alle Elemente dazwischen werden entsprechend um eine Position nach vorne oder hinten verschoben.
  Beispiel: [1 2 3 4 **5** 6] → [**5** 1 2 3 4 6]

- `TwoOptMutationOfPermutation` spiegelt einen Teil der Permutation. Es werden zwei Positionen zufällig gleichverteilt ausgewählt und alle Elemente inklusive der beiden gewählten gespiegelt.
  Beispiel: [1 2 **3 4 5** 6] → [1 2 **5 4 3** 6]

- `FourOptMutationOfPermutation` vertauscht zwei Intervalle, wobei die benötigten vier Positionen so gewählt werden, dass eines der beiden Intervalle auch aus einem Element bestehen kann.
  Beispiel 1, die Positionen 2,3,6,8 wurden gewählt:
  [1 **2 3** 4 5 **6 7 8** 9] → [1 **6 7 8** 4 5 **2 3** 9]
  Beispiel 2, die Positionen 2,2,5,7 wurden gewählt:
  [1 **2** 3 4 **5 6 7** 8 9] → [1 **5 6 7** 3 4 **2** 8 9]

Als Crossoveroperatoren stehen zur Verfügung:

- `OnePointCrossoverOfPermutation` führt ein Ein-Punkt-Crossover auf den `sortValues` zweier Individuen durch. Anschließend werden die entstehenden beiden Permutation neu sortiert und skaliert.

- `IntermediateCrossoverOfPermutation` nimmt Position für Position das arithmetische Mittel der `sortValues` zweier Permutationen. Das resultierende Individuum wird ebenfalls sortiert und skaliert.

- `DiscreteCrossoverOfPermutation` nimmt für jede Position des erzeugten Individuums mit 50% Wahrscheinlichkeit den `sortValue` eines der beiden gekreuzten Individuen an. Auch hier wird anschließend sortiert und skaliert.

3.4 Über die Plug-In-Beschreibungen

Die von uns implementierten Plug-Ins sollen übersichtlich dargestellt werden. Deshalb haben die folgenden Texte eine einheitliche Gliederung. Der Abschnitt „Beschreibung“ gibt einen allgemeinen Überblick über das betreffende Plug-In, seine Funktionsweise und Eigenschaften. Wesentliche Parameter und deren Auswirkungen werden im Abschnitt „Parameter“ erörtert.

„Parameterübersicht“ enthält eine Tabelle, die die Parameterschnittstelle des Plug-Ins auflistet. Sie soll sowohl als Übersicht, als auch als Referenz dienen.

- Die Spalte **Parameter** gibt den Bezeichner des Parameters an.
- In der Spalte **Typ** findet sich die Art des Parameters. Hierbei wird die Java-interne Repräsentation, also z. B. `double` oder `java.util.Date` angegeben.
- **Defaultwert** gibt über die Plug-In-interne Voreinstellung des Parameters Auskunft. Dieser ist vom Benutzer im Quellcode veränderbar, der Wert in der Tabelle gibt somit nur die Standardkonfiguration des Plug-Ins wieder.
- In der Spalte **Wertebereich** ist der semantisch sinnvolle Bereich angegeben, den dieser Parameter einnehmen kann. Eine Einstellung außerhalb dieses Bereiches kann zum Abbruch des Programmes oder zu falschen bzw. sinnlosen Ergebnissen führen. Hierbei ist zu beachten, dass manche Parametereinstellung bzw. deren Kombinationen zwar semantisch korrekt, aber aus Sicht der Optimierung nicht sinnvoll sind. Ist ein Eintrag in dieser Spalte leer, so umfasst der Wertebereich alle Werte des Typs, etwa alle double-Werte.
- Schließlich ist in der letzten Spalte noch eine **Kurzbeschreibung** des Parameters zu finden.

3.5 Heuristiken

3.5.1 Genetischer Algorithmus

Beschreibung

Genetische Algorithmen (GA) als globale Suchverfahren setzen Strategien aus der Evolutionstheorie ein. Die Fitness ist die einzige Information, die der GA zur Beurteilung eines Suchpunktes benötigt. Unsere Implementation arbeitet mit einer Anzahl zu Individuen codierten reellwertigen Objektvariablen, die eine Population bilden. Die genetischen Operatoren Selektion, Crossover und Mutation erzeugen aus den Individuen einer zufallsgenerierten Startpopulation die jeweils nächste Generation. Hierbei tendieren die Individuen dazu, sich um einen Punkt im Suchraum zu konzentrieren. Die genetischen Operatoren basieren auf stochastischen Prozessen und haben auf

Grund ihrer Wirkungsmechanismen im Suchraum unterschiedliche Auswirkungen. Zur eingehenderen Beschreibung von GA sei auf die Ausführungen in [Gol89] oder [Nis97] verwiesen.

Der in diesem Plug-In implementierte reellwertige Genetische Algorithmus unterscheidet sich von den in den oben angegebenen Monographien in einigen wesentlichen Punkten, es handelt sich also nicht um einen Standard-GA. Zunächst ist die verwendete Selektion bei Genetischen Algorithmen unüblich. Sie entspricht einer „truncation selection“. Weiterhin werden in dieser Variante durch Mutation und Crossover erzeugte Individuen zur Population hinzugefügt, sodass ein weiterer Operator zur Kontrolle der Populationsgröße nötig war (siehe unten).

Der Ablauf lässt sich in sechs Schritte zusammenfassen:

1. **Erzeugung einer Ausgangspopulation**
   Am Anfang wird mittels der Methode `initialize()` eine zufällige Population einer vorgegebenen Größe, also eine Menge von Individuen, erzeugt. Es wird eine ArrayList von Objekten mit reellwertigen Objektvariablen innerhalb vorgegebener Grenzen zufällig gleichverteilt initialisiert. Mit den Individuen dieser Population beginnt der Evolutionsprozess. Im nächsten Schritt werden die Fitnesswerte der erzeugten Individuen berechnet. In der Regel sind die so entstandenen Lösungen nicht sehr gut.

2. **Selektion**
   In diesem Schritt wird die natürliche Selektion nachgebildet. Die „überlebensfähigsten“ Lösungen bestehen weiter, andere wiederum „sterben“, d. h. werden aus der Population entfernt. Die Selektion wird in zwei Schritten nachgebildet. Im ersten Schritt werden die Individuen nach der Fitness sortiert, im zweiten Schritt erfolgt dann die Auswahl der besseren Lösungen. Es werden also die „schlechten“ Individuen entfernt. Die Selektion zielt darauf ab, die besseren Lösungen in der Population herauszufiltern. Man könnte auch sagen, dass sie den Evolutionsprozess steuert und fokussiert.

3. **Crossover**
   Neue Lösungen werden generiert, indem die Parametersätze der Individuen paarweise miteinander kombiniert werden. Sinn der Kreuzung ist es, Lösungen zu kombinieren und daraus neue, bessere zu erzeugen. Im Zusammenhang mit der Selektion lautet die grundlegende Annahme

dahinter: „*gut + gut = besser*“ (so genannte „Buildingblock Hypothese“). Hier wird die bekannte Variante „1-Punkt-Crossover“ verwendet. Abhängig von der Crossoverwahrscheinlichkeit werden zwei Individuen an einer zufällig gleichverteilt gewählten Stelle gekreuzt, oder nicht. Die entstehenden Individuen werden zu der Population hinzugefügt.

4. **Mutation**
   Einige Lösungen werden leicht verändert (mutiert). Mit der Mutation verfolgt man das Ziel, gänzlich andere Lösungen zu finden, also Lösungen, die mit der Kreuzung allein nicht generiert werden können. Die Mutation soll gewährleisten, dass die Vielfalt (d. h. viele unterschiedliche Lösungen) in der Population erhalten bleibt. Ähnlich der Kreuzung werden hintereinander alle Individuen der Population betrachtet. Bei jedem Individuum wird abhängig von der Mutationswahrscheinlichkeit entschieden, ob es mutiert wird oder nicht. Im Falle einer Mutation werden zufällig gleichverteilt die Werte innerhalb des Individuums um maximal eine vorgegebene Mutationsschrittweite verändert.

5. **Resize**
   Ausgehend von den in der Selektion ausgewählten Individuen sind durch die Anwendung des Crossover- bzw. Mutationsoperators neue Individuen in die Population hinzugekommen. Abhängig von der gewählten Crossover- und Mutationswahrscheinlichkeit kann die nun entstandene Population eine Größe haben, welche über oder unter der geforderten Populationsgröße liegt. Dazu werden im ersten Fall die überzähligen Individuen entfernt, und zwar mit dem schon beschriebenen Selektionsoperator. Es wird hier aber nicht auf die Selektionsgröße `Selection size` selektiert, sondern auf die Populationsgröße `Population size`. Im zweiten Fall werden neue Individuen erzeugt, bis die geforderte Populationsgröße erreicht ist.

6. **Nächste Generation**
   Nachdem Selektion, Crossover, Mutation und Resize durchgeführt worden sind, ist eine neue, veränderte Population entstanden. Um die nächste Generation zu erzeugen, wird wieder zu Schritt zwei gesprungen.

Parameter

Ein wichtiger Parameter dieses Plug-Ins ist `Population size`, die Populationsgröße. Sie bestimmt nicht nur, wie viel Rechenzeit (Funktionsauswertungen) eine Generation beansprucht. Das Verhältnis `Population size` zu `Selection size`, das als Selektionsdruck bezeichnet wird, bestimmt zudem das Verhalten der Heuristik maßgeblich. Die Parameter `Probability of crossover` und `Probability of mutation` werden intern verwendet. Sie bestimmen, mit welcher Wahrscheinlichkeit der jeweilige Operator gebraucht wird.
`Maximal mutationincrement` ist ein interner Parameter des Mutationsoperators. Er hat entscheidenden Einfluss auf die Güte der gefundenen Lösungen. Wählt man ihn zu groß, so kann es passieren, dass man dem Optimum nicht nah genug kommt. Wählt man ihn klein, sinkt die Konvergenzgeschwindigkeit. Dieses Problem wird bei anderen Mutationsoperatoren durch eine adaptive Schrittweite gelöst, was bei einem Genetischen Algorithmus nicht vorgesehen ist.

Parameterübersicht

Parameter	**Typ**	**Default**	**Bereich**	**Kurzbeschreibung**
Dimension	integer	10	$x > 0$	Dimension des Problems
Population size	integer	1000	$x > 0$	Anzahl der Individuen in einer Population
Selection size	integer	100	$x \geq 0$	Anzahl der Individuen, die in einer Population eine Generation überleben
Probability of crossover	double	0.9	$0 \leq x \leq 1$	Wahrscheinlichkeit für die Durchführung eines Crossover
Probability of mutation	double	0.1	$0 \leq x \leq 1$	Wahrscheinlichkeit, mit der die Mutation eines Individuums stattfindet
Maximal mutation-increment	double	0.1	$x > 0$	Maximale Länge einer/s Mutation/Schrittes im Suchraum
Solution minimum	double	-10.0		Minimaler Initialwert der Parameter aller Individuen
Solution maximum	double	10.0		Maximaler Initialwert der Parameter aller Individuen

3.5.2 Genetischer Algorithmus (Permutation)

Beschreibung

Dieses Plug-In realisiert den oben beschriebenen (nicht-Standard) GA auf Permutationen. Im Prinzip ist der Ablauf identisch (siehe 3.5.1). Anstelle der reellwertigen Mutation wird hier jedoch die `FourOptMutationOfPermutation` (siehe 3.3) benutzt.

Parameter

Die Bedeutung der Parameter bei dieser Permutationsvariante sind die gleichen wie bei dem reellwertigen GA.

Parameterübersicht

Parameter	Typ	Default	Bereich	Kurzbeschreibung
Dimension	integer	10	$x > 0$	Dimension des Problems
Population size	integer	1000	$x > 0$	Anzahl der Individuen in einer Population
Selection size	integer	100	$x \geq 0$	Anzahl der Individuen, die in einer Population eine Generation überleben
Probability of crossover	double	0.9	$0 \leq x \leq 1$	Wahrscheinlichkeit für die Durchführung eines Crossover
Probability of mutation	double	0.1	$0 \leq x \leq 1$	Wahrscheinlichkeit, mit der die Mutation eines Individuums stattfindet

3.5.3 Evolutionsstrategie

Beschreibung

Evolutionsstrategien (ES) gehören, wie auch die Genetischen Algorithmen, zur Klasse der Evolutionären Algorithmen, die für die Optimierung schwieriger Funktionen und Probleme eingesetzt werden. Sie wurden erstmals in [Rec73] und [Sch75] vorgestellt und operieren auf Populationen von Individuen, welche reellwertig kodierte Lösungen für das Problem darstellen. Anhand einer Fitnessfunktion werden diese bewertet, um mit den besten Individuen neue Lösungen zu erzeugen. Individuen von diesem Typ besitzen neben den Objektparametern, welche die kodierte Lösung repräsentieren, auch Strategieparameter, die den Ablauf der Optimierung steuern. Mit Hilfe der genetischen Operatoren Rekombination, Mutation und Selektion werden aus einer

Anzahl von Lösungen neue Individuen erzeugt, welche die nächste Generation der Population darstellen. Dieses Verfahren wird so lange iteriert, bis eine Abbruchbedingung erfüllt ist.
Die von uns implementierte Evolutionsstrategie ist folgendermaßen aufgebaut:

1. **Initialisierung**
   Die Initialisierung der Heuristik findet durch das Erzeugen zweier Startpopulationen in der Methode `initialize()` statt. Die Individuen dieser Population bekommen hierbei sowohl für die Elemente des Lösungsvektors als auch für die Strategieparameter den Wert 0.0 zugewiesen. Anschließend werden den Lösungsvektoren zufällige Werte zugewiesen, die sich zwischen den zulässigen Grenzen `Minimum value` und `Maximum value` befinden. Die Schrittweiten der Initialindividuen werden zudem einheitlich auf die zuvor berechnete Standardschrittweite gesetzt. Zuletzt werden die Fitnesswerte der Startindividuen berechnet, die im Allgemeinen durch die zufällige Initialisierung der Individuen noch nicht besonders gut sind.

2. **Rekombination**
   Aus der Menge der aktuellen Individuen werden zufällig gleichverteilt zwei Eltern ausgewählt. Auf ihren Lösungsvektoren wird die diskrete Rekombination angewandt, d. h. für jede Stelle im Lösungsvektor des Kindes wird entschieden, von welchem Elternteil der entsprechende Eintrag übernommen wird. Die Schrittweiten dagegen werden mit Hilfe der intermediären Rekombination gekreuzt, d. h. jeder Eintrag im Vektor des Kindes erhält das arithmetische Mittel der beiden Eltern-Einträge. Im Gegensatz zum Crossover beim Genetischen Algorithmus wird bei einer Evolutionsstrategie immer gekreuzt. In der Literatur lassen sich weitere mögliche Rekombinationsoperatoren finden, die allerdings nicht implementiert wurden.

3. **Mutation**
   Um eine möglichst breite Lösungsvielfalt zu erreichen, wird das neu entstandene Individuum nach der Kreuzung noch mutiert, d. h. seine Schrittweiten werden mit einer normalverteilten Zufallszahl multipliziert und zum Lösungsvektor addiert. Die Schrittweiten selbst werden ebenfalls mutiert, wobei die internen Parameter der globalen und lokalen Mutation eine wichtige Rolle spielen. Der Benutzer kann diese aller-

dings nicht verändern, sie werden bei der Initialisierung der Heuristik aus den gegebenen Parametern berechnet. Wie auch bei der Rekombination ist die Mutation unabhängig von einer Mutationswahrscheinlichkeit, d. h. diese wäre gleich 1.

4. **Selektion**
   Bei einer Evolutionsstrategie unterscheidet man im Allgemeinen bei der Selektion zur Ersetzung zwischen einer Komma- und einer Plus-Strategie. Bei der Komma-Strategie wird die Elternpopulation der Größe $\mu \leq \lambda$ komplett verworfen und durch die μ besten Kinder ersetzt. Dies hat den Vorteil, dass sich die Heuristik nicht so leicht auf eine Richtung festlegt und offener ist für andere Lösungswege. Die Plus-Strategie dagegen wählt aus der Vereinigungsmenge der $\mu + \lambda$ Eltern und Kinder die μ besten Individuen aus und übernimmt diese in die nächste Generation. Auf diese Weise werden Individuen nur verworfen, wenn sie durch bessere ersetzt werden. Bei beiden Strategien werden die relevanten Individuen mit *quicksort* nach ihrer Fitness aufsteigend sortiert und die μ besten in die Elternpopulation übernommen.

Nach der Anwendung der genetischen Operatoren ist eine Generation vollzogen, und die Heuristik wird ab der Rekombination iteriert.

Parameter

Wichtige Parameter der ES sind die Populationsgrößen `Population size` und `Temporary population size` sowie die Dimension der Individuen. Sie bestimmen die Rechendauer eines Generationendurchlaufs und somit die Komplexität der Heuristik. Der Selektionsdruck, d. h. das Verhältnis der Populationsgröße zur temporären Populationsgröße, bestimmt das Tempo der Optimierung und stellt somit eines der wichtigsten Werkzeuge zur Parameteroptimierung dar.

Die Werte `Maximum value` und `Minimum value` grenzen den zulässigen Zahlenbereich für die Lösungsvektoreinträge ein. Je größer dieser Bereich gewählt wird, umso länger kann es unter Umständen dauern, bis eine optimale Lösung gefunden wird. Die booleschen Parameter `Global step size` und `Comma selection` werden intern gebraucht, um den Ablauf der Heuristik zu steuern. Letzterer legt die Ersetzungsstrategie fest, was sich auch auf den Verlauf der Optimierung auswirkt. Näheres hierzu wurde bereits im Abschnitt „Selektion“ erläutert.

Parameterübersicht

Parameter	Typ	Default	Bereich	Kurzbeschreibung
Population size	integer	15	$x > 0$	Anzahl der Individuen in einer Population
Temporary population size	integer	100	$x > 0$	Anzahl der Individuen in einer Kinderpopulation
Dimension	integer	3	$x > 0$	Dimension des Problems
Maximum value	double	500		Maximaler Initialwert der Parameter aller Individuen
Minimum value	double	-500		Minimaler Initialwert der Parameter aller Individuen
Global step size	boolean	true		Gibt an, ob eine globale oder individuelle Schrittweite verwaltet wird
Comma selection	boolean	false		Entscheidet, ob die Plus- oder Komma-Strategie gewählt wird

3.5.4 Evolutionsstrategie (Permutation)

Beschreibung

Um die Evolutionsstrategien nicht nur auf reellwertige, sondern auch auf Permutationsprobleme anwenden zu können, wurde die von uns implementierte ES (siehe [Rec73] und [Sch75]) auch an Permutationen angepasst. Da der Ablauf mit jenem der zuvor beschriebenen ES identisch ist, sollen hier nur die Abweichungen zum Standardalgorithmus erläutert werden.

Die ESPermutation verwendet keine Strategieparameter wie Schrittweiten und Rotationswinkel, weshalb die Klasse `StrategyStandardES` komplett verworfen wurde. Die Klassen `IndividualPermutationES` und `PopulationPermutationES` konnten fast vollständig übernommen werden. Die einzige Änderung besteht darin, dass nun statt Lösungsvektoren Lösungspermutationen verwaltet werden.

Da die in der StandardES verwendeten Operatoren nur für reellwertige Vektoren funktionierten, werden speziell an Permutationen angepasste Operatoren verwendet (siehe 3.3). In der hier realisierten PermutationES wird der Kreuzungsoperator `DiscreteCrossoverOfPermutation` für die Rekombination verwendet, während durch `FourOptMutationOfPermutation` die Mutation realisiert wird.

Parameter

Da die PermutationES bis auf wenige Ausnahmen über dieselben Parameter wie die StandardES verfügt, sei hier auf 3.5.3 verwiesen. Diese Heuristik verwaltet keine Strategieparameter, weshalb der Parameter `Global step size` überflüssig und daher verworfen wurde. Weiterhin wurden bei der PermutationES die Parameter `Maximum value` und `Minimum value` nicht mehr gebraucht. Lediglich die Parameter `Population size`, `Temporary population size`, `Dimension` und `Comma selection` werden bei dieser Heuristik verwaltet.

Parameterübersicht

Parameter	Typ	Default	Bereich	Kurzbeschreibung
Population size	integer	15	$x > 0$	Anzahl der Individuen in einer Population
Temporary population size	integer	100	$x > 0$	Anzahl der Individuen in einer Kinderpopulation
Dimension	integer	3	$x > 0$	Dimension des Problems
Comma selection	boolean	false		Entscheidet, ob die Plus- oder Komma-Strategie gewählt wird

3.5.5 Simulated Annealing

Beschreibung

Der Begriff „annealing“ beschreibt den technischen Prozess, bei dem ein Material für eine festgelegte Zeitdauer einer erhöhten Temperatur ausgesetzt wird, um die Bildung metastabiler Zustände wie Gitterunregelmäßigkeiten, -fehlstellen o. ä. zu vermeiden. Eine niedrige Abkühlungsrate garantiert schließlich, dass ein Zustand minimaler Energie erreicht werden kann. Grundlage der Methode des Simulated Annealing ist die Analogie zur Thermodynamik, insbesondere zum Verhalten von kristallisierenden Flüssigkeiten oder erstarrendem Schmelzen. Die Energie E eines Systems im thermischen Gleichgewicht gehorcht einer Boltzmann-Wahrscheinlichkeitsverteilung $p(E) = e^{\frac{-E}{kT}}$ mit T als Temperatur sowie der Boltzmann-Konstanten k. Danach kann ein System selbst bei niedriger Temperatur T noch mit einer geringen Wahrscheinlichkeit einen hohen Energiezustand einnehmen. Entsprechend besteht die Chance, dass das System ein lokales Energieminimum wieder verlässt.

Ausgehend vom „Metropolis Algorithmus“ [MRR+53] entwickelte S. Kirkpatrick die Heuristik „Simulated Annealing“ [KGV83]. Der Ablauf der hier implementierten Variante lässt sich wie folgt zusammenfassen:

Da dieses Verfahren nur auf einem Individuum arbeitet, wird eine zufällige reellwertige Ausgangskonfiguration erzeugt und die benachbarten Lösungen auf die vom Problem stammende Zielfunktion untersucht. Ob die neue Konfiguration als Ausgangspunkt für einen neuen Suchlauf akzeptiert wird, hängt von der Qualität der Konfiguration und einem Kontrollparameter ab. Der Kontrollparameter ist ein abnehmender Wert zwischen 1 und 0, der in Verbindung mit einer Zufallszahl zwischen 0 und 1 entscheidet, ob ein neuer, eigentlich ungünstigerer Zustand akzeptiert werden kann oder ob er abgelehnt wird.

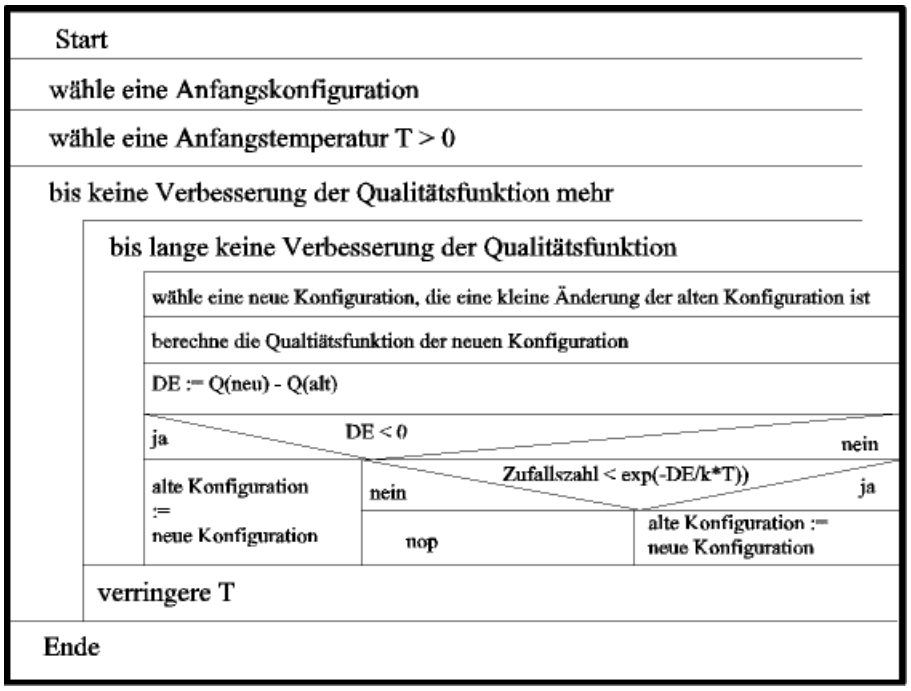

Abbildung 3.1: Struktogramm des SA. Aus [Mot03].

So werden am Anfang des Verfahrens noch fast wahllos neue Konfigurationen übernommen, da die Wahrscheinlichkeit p, mit der eine schlechtere Variante akzeptiert wird, noch sehr hoch ist. Nimmt p ab, sinkt auch die Bereitschaft, eine schlechtere Lösung zu akzeptieren. Geht p langsam gegen

0, so werden nur noch kleinere Variationen zugelassen, analog zu der Metallschmelze, die immer zähflüssiger wird und bei der sich die Sprünge zu ungünstigeren energetischen Zuständen immer mehr reduzieren.

Damit der Algorithmus immer wieder einen Ausweg aus lokalen Minima findet, muss die Abkühlung, das heißt das Sinken der Wahrscheinlichkeit, entsprechend langsam stattfinden. Setzt man die Wahrscheinlichkeit von Anfang an auf 0, so hat man ein simples Hillclimbing-Verfahren.

Das Struktogramm 3.1 zeigt den Simulated Annealing-Algorithmus.

Parameter

Der wichtigste Parameter dieses Plug-Ins ist der Temperaturreduktionsfaktor (`Reduction factor`). Dieser Kontrollparameter steuert die Abkühlgeschwindigkeit. Langsames Abkühlen gibt den Molekülen Zeit, sich selbst zu ordnen und Kristalle zu formen, um so ggf. auch eine völlig neue Struktur zu schaffen. Dies ermöglicht das Entkommen aus einem lokalen Minimum, was bei einer zu hohen Abkühlgeschwindigkeit unter Umständen nicht mehr möglich wäre. Ein weiterer wichtiger Parameter für die Geschwindigkeit des Algorithmus ist `Maximum try`. Dieser gibt an, wie oft der Algorithmus durchlaufen wird, um eine bessere Lösung zu finden, bevor die Temperatur abgesenkt wird, damit kein unendlicher Suchvorgang stattfindet. Die anderen Parameter sind Initialparameter für den Algorithmus und geben den zulässigen Bereich von Lösungen an.

Parameterübersicht

Parameter	Typ	Default	Bereich	Kurzbeschreibung
Dimension	integer	10	$x > 0$	Dimension des Problems
Maximum try	double	100	$x > 0$	Gibt an, wie oft der Algorithmus durchlaufen wird
Start temperature	double	10	$x > 0$	Anfangstemperatur
Reduction factor	double	0.9	$0 \leq x \leq 1$	Kontrollparameter, der die Abkühlungsgeschwindigkeit steuert
Solution minimum	double	-10		Minimaler Initialwert der Parameter des Individuums
Solution maximum	double	10		Maximaler Initialwert der Parameter des Individuums

3.5.6 Simulated Annealing (Permutation)

Beschreibung

Dieses Plug-In realisiert Simulated Annealing auf Permutationen. Im Prinzip ist der Ablauf identisch zu dem schon beschriebenen SA (siehe 3.5.5). Der Benutzer hat hier aber die Wahl einer der bereits beschriebenen Mutationsoperatoren auf Permutationen (siehe 3.3):

- `InterchangeMutationOfPermutation`
- `ShiftMutationOfPermutation`
- `TwoOptMutationOfPermutation`
- `FourOptMutationOfPermutation`

Parameter

Die Parameter dieser Permutationsvariante des Simulated Annealing haben dieselbe Bedeutung wie die schon beschriebenen Parameter des reellwertigen Simulated Annealing. Ein zusätzlicher Parameter ist `Mutation number`, der den Mutationsoperator auswählt.

Parameterübersicht

Parameter	Typ	Default	Bereich	Kurzbeschreibung
Dimension	integer	10	$x > 0$	Dimension des Problems
Maximum try	double	100	$x > 0$	Gibt an, wie oft der Algorithmus durchlaufen wird
Start temperature	double	10	$x > 0$	Anfangstemperatur
Reduction factor	double	0.9	$0 \leq x \leq 1$	Kontrollparameter, welcher die Abkühlungsgeschwindigkeit steuert
Mutation number	integer	1	$x \in 1, 2, 3, 4$	Gibt an, welcher der vier möglichen Mutationsoperatoren angewendet wird

3.5.7 Sintflutalgorithmus

Beschreibung

Beim Simulated Annealing ist die Steuerung der Temperaturverringerung nicht unproblematisch. Wie bei allen heuristischen Verfahren muss hier erst ein Vorgehen gefunden werden, welches möglichst gute Lösungen hervorbringt. Man kann die Temperatur schnell oder langsam sinken lassen, linear, progressiv oder degressiv, usw. Um diese Problematik zu umgehen hat G. Dueck 1993 eine Vereinfachung des Simulated-Annealing veröffentlicht ([DSW93]). Die Grundidee ist im Seminar 5.2.3 beschrieben. Der hier implementierte Sintflut-Algorithmus verfolgt diese Grundidee, jedoch wurden einige Modifikationen vorgenommen, um die nötige Effizienz zu gewährleisten.

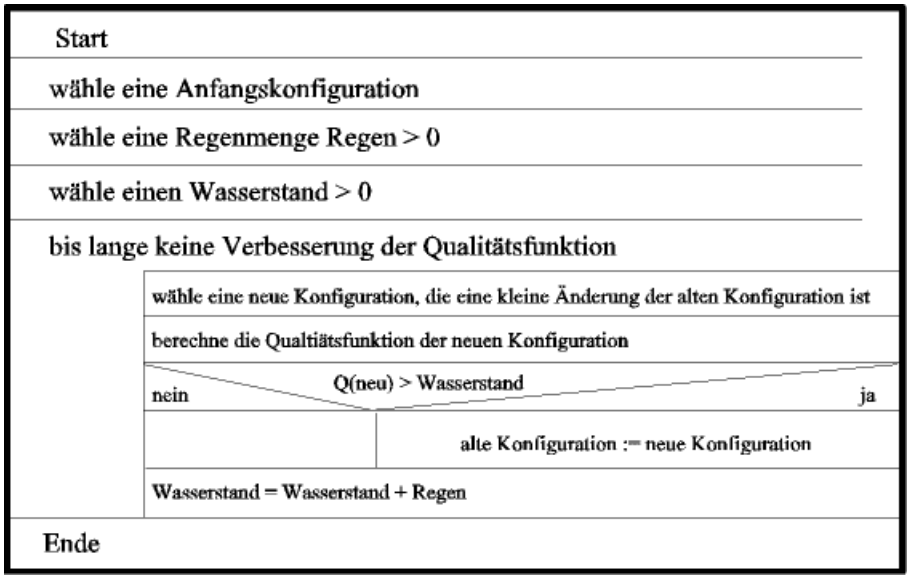

Abbildung 3.2: Struktogramm des Sintflutalgorithmus. Aus [Mot03].

Der einzige Parameter der zu berücksichtigen ist, ist der „Regen“. Das Verfahren akzeptiert alle Lösungen, deren Fitness über einem bestimmten Wasserstand, dem Threshold, liegt. Im Gegensatz dazu akzeptiert das Simulated Annealing unter Umständen auch ungünstigere Zustände, welche unter dem Threshold liegen. Der Wasserstand wird hier im Gegensatz zu einer linearen Wasserstandserhöhung, wie sie bei Dueck eingesetzt wird, aus Effizi-

enzgründen mit einem Reduktionsfaktor `Water factor` degressiv abgesenkt. Dies geschieht nach den Vorschriften $red\_factor = red\_factor * Water factor$ und $Wasserstand = (Fitness_{max} - Fitness_{min}) * red\_factor + Fitness_{min}$.

Analog zum Simulated Annealing gibt dabei die aktuelle Temperatur die „Beweglichkeit“ des Individuums an. Dieser neue Algorithmus benötigt laut Dueck doppelt solange für die Optimierung wie das Simulated Annealing. Das Struktogramm 3.2 zeigt den hier implementierten Sintflutalgorithmus.

Parameter

Die wichtigsten Parameter dieses Plug-Ins sind der Temperaturreduktionsfaktor (`Reduction factor`) und der Wasserstandreduktionsfaktor (`Water factor`). Die Bedeutung der zum Simulated Annealing gleichen Parameter sind dieselben. Jedoch hat `Water factor` zusätzlich einen bedeutenden Einfluss auf das Finden neuer Lösungen, denn ausgehend vom `Fitness maximum` als Initialwasserstand muss jede neue Lösung unter diesem liegen (hier nimmt der Wasserstand ab und nicht zu, anders als in der anschaulichen Idee des „Sintflut“).

Parameterübersicht

Parameter	Typ	Default	Bereich	Kurzbeschreibung
Dimension	integer	6	$x > 0$	Dimension des Problems
Maximum try	integer	100	$x > 0$	Gibt an, wie oft der Algorithmus durchlaufen wird
Start temperature	double	300	$x > 0$	Anfangstemperatur
Reduction factor	double	0.9	$0 \leq x \leq 1$	Kontrollparameter, welcher die Abkühlungsgeschwindigkeit steuert
Solution minimum	double	-10		Minimaler Initialwert der Parameter des Individuums
Solution maximum	double	10		Maximaler Initialwert der Parameter des Individuums
Fitness minimum	double	0		Vermuteter minimaler Fitnesswert des Problems
Fitness maximum	double	100		Vermuteter maximaler Fitnesswert des Problems
Water factor	double	0.8	$0 \leq x \leq 1$	Wasserstandsreduktionsfaktor, welcher die Sintflutgeschwindigkeit steuert

3.5.8 Sintflutalgorithmus (Permutation)

Beschreibung

Dieser Algorithmus realisiert den Sintflutalgorithmus auf Permutationen. Im Prinzip ist der Ablauf identisch zu dem obigen Algorithmus (siehe 3.5.7). Der Benutzer hat hier aber die Wahl einer der bereits beschriebenen (siehe 3.3) Mutationsoperatoren auf Permutationen:

- `InterchangeMutationOfPermutation`
- `ShiftMutationOfPermutation`
- `TwoOptMutationOfPermutation`
- `FourOptMutationOfPermutation`

Parameter

Die Parameter dieser Permutationsvariante haben dieselbe Bedeutung, wie im oben beschriebenen reellwertigen Fall. Ein zusätzlicher Parameter ist `Mutation number`, der den Mutationsoperator auswählt.

Parameterübersicht

Parameter	Typ	Default	Bereich	Kurzbeschreibung
Dimension	integer	10	$x > 0$	Dimension des Problems
Maximum try	double	100	$x > 0$	Gibt an, wie oft der Algorithmus durchlaufen wird
Start temperature	double	10	$x > 0$	Anfangstemperatur
Reduction factor	double	0.9	$0 \leq x \leq 1$	Kontrollparameter, welcher die Abkühlungsgeschwindigkeit steuert
Fitness minimum	double	0		Vermuteter minimaler Fitnesswert des Problems
Fitness maximum	double	100		Vermuteter maximaler Fitnesswert des Problems
Water factor	double	0.8	$0 \leq x \leq 1$	Wasserstandsreduktionsfaktor, welcher die Sintflutgeschwindigkeit steuert
Mutation number	integer	1	$x \in 1, 2, 3, 4$	Gibt an, welcher der vier möglichen Mutationsoperatoren angewendet wird

3.5.9 Ant Colony Optimization

Beschreibung

Die Ant Colony Optimization (ACO) [DD99] war uns bereits in der Seminarphase zu Beginn der Projektgruppe vorgestellt worden ([BBB+03]). Für die Wahl von ACO als eine der zu implementierenden Heuristiken sprachen im wesentlichen zwei Dinge:

1. ACO arbeitet vor allem auf diskreten Problemen.
2. Der Ansatz von ACO unterscheidet sich von dem der Evolutionären Algorithmen und ist somit besonders geeignet, um die Generalität unseres Frameworks zu testen.

Die Heuristik ahmt das Verhalten der Ameisen nach, kürzeste Wege zwischen Futterquellen und Ameisenhaufen zu etablieren. Dabei hinterlassen Ameisen Pheromone, wenn sie einen bestimmen Weg entlang gehen und lassen sich bei der Wahl der Wege vom Pheromongehalt durch frühere Ameisen beeinflussen. Um den Pheromongehalt nicht mehr benutzter Wege langsam abzusenken, werden äußere Umwelteinflüsse (z. B. Wind), zusammengefasst unter dem Begriff *Evaporation*, mit in die Heuristik aufgenommen.

Die Heuristik ist darauf ausgelegt, Probleme, die sich als Graphen repräsentieren lassen, zu lösen. Sie ermittelt kürzeste Touren in dem (vollständigen) Graphen. Dazu verwaltet sie eine Matrix, auf dem der Pheromongehalt aller Kanten des Graphen vermerkt ist. Es wird der Lauf von Ameisen „simuliert“, indem von einem zufällig ausgewählten Startknoten eine Ameise läuft, bis sie alle Knoten genau einmal besucht hat. Die Wahl der Knoten (und damit der zu wählenden Kanten) wird randomisiert getroffen, wobei die zu wählenden Kanten nach ihrem Pheromongehalt gewichtet werden (Roulette-Rad-Technik).

Ist die vom Benutzer der Heuristik vorgegebene Anzahl von Ameisen über den Graphen gelaufen, wird die Pheromonmatrix für jede Ameise aktualisiert. Dazu wird der Pheromongehalt einer Kante, die auf der Tour der Ameise liegt um den vorgebenen Wert erhöht, relativ zur Güte der Tour (also $\frac{Pheromonwert}{Fitness}$). Danach wird geprüft, ob eine der Ameisen eine Tour mit Kosten gelaufen ist, die unterhalb des bisherigen Minimums liegt. Diese Tour wird dann als momentan bestes Zwischenergebnis gespeichert. Abschließend wird die Evaporation dadurch realisiert, dass alle Einträge der Matrix um einen vorgebenen Wert verringert werden.

Um der Objektorientierung unseres Ansatzes gerecht zu werden, wurden folgende Klassen im Paket `moon.plugin.heuristic.antColony` erstellt:

- `AntColony` ist die „Hauptklasse“, die das Interface `Heuristic` implementiert.

- `Ant` kapselt die Informationen, die eine Ameise auf einer Tour bereitstellt. Im wesentlichen wird hier die Tour abgespeichert mit der Funktionalität, die Matrix entsprechend aktualisieren zu können.

- `RouletteWheel` ermöglicht die gewichtetete Zufallsauswahl aus beliebig vielen Werten. Dies ist der zentrale Teil der Wegbestimmung der Ameisen.

Parameter

Die (inhaltlich) wichtigen Parameter dieser Heuristik sind die Anzahl der Ameisen und der Pheromongehalt. Alle Ameisen produzieren unabhängig voneinander je eine Lösung. Erst danach wird die Pheromonmatrix von jeder Ameise geändert. Der `Pheromon amount` gibt dabei an, wieviel Pheromon sie auf ihrer Tour ablegen dürfen. Die Evaporation reduziert auf allen Kanten das Pheromon. Dies hat zur Folge, dass Kanten, die von Ameisen nicht mehr besucht werden, mit der Zeit unattraktiver, sozusagen „vergessen“ werden. Darüber hinaus verhindert die Evaporation, dass sich auf manchen Kanten so viel Pheromon anhäuft, dass sie alle benachbarten Kanten derart dominierien, dass keine Suche mehr statfindet.

Zum Funktionieren der Heuristik ist es von entscheidender Bedeutung, dass der Parameter für die Problemdimension korrekt gesetzt ist, da ansonsten keine gültigen Lösungen für das Problem generiert werden können. Die Notwendigkeit dazu ergibt sich aus dem Design des Frameworks und ist inhaltlich offenkundig unbedeutend (da fest und bekannt).

Parameterübersicht

Parameter	Typ	Default	Bereich	Kurzbeschreibung
Dimension	integer	3	$0 < x$	Dimension des Problems
Evaporation	double	0.1	$0 \leq x \leq 1$	Evaporation („Verwehen“ des Pheromons) pro Generation
Number of ants	integer	3	$1 \leq x$	Anzahl der Ameisen
Pheromon amount	double	10.0	$0 < x$	Menge an Pheromon, die, gewichtet mit der Fitness, von einer Ameise auf einer Kante hinterlassen wird
Initial pheromon amount	double	0.1	$0 \leq x$	Initiale Wertebelegung der Pheromonmatrix
Influence of pheromons	double	1.0	$0 \leq x$	Gewichtung der initialen Wertebelegung der Pheromonmatrix
Influence of edge weights	double	5.0	$0 \leq x$	Gewichtung der Pheromonmatrix bei der Ermittlung des nächsten Schritts einer Ameise

3.5.10 Particle Swarm Optimization

Beschreibung

Particle Swarm Optimization (PSO) basiert auf dem Verhalten mehrerer Partikel in einem Schwarm. Die Partikel suchen nach Optima und bewegen sich in einem mehrdimensionalen Raum mit unterschiedlichen Geschwindigkeiten. Die Partikel passen ihr Verhalten einem Lernprozess an, dabei werden grundlegende Prinzipien des Verhaltens in einem sozialen System berücksichtigt — es kommt zu Auswertung, Vergleich und Imitation ([EKS01]). Die beste erreichte Position eines Partikels sowie die beste erreichte Position aller Partikel beeinflussen die Berechnung der neuen Geschwindigkeit des Partikels.

Den Algorithmus kann man grob so darstellen:

for $i = 0$ **to** swarmSize
    initialisiere jeden Partikel mit zufälligen Positions-
    und Geschwindigkeitswerten und setze beste erreichte
    Partikelposition auf seine momentane Position**;**
end
berechne die global beste erreichte Position**;**
while Abbruchbedingung (z. B. Anzahl der Generationen) nicht erfüllt ist
    erhöhe den Iterationsschritt**;**
    for $i = 0$ **to** swarmSize
        berechne die neue Position und Geschwindigkeit
        des Partikels nach PSO-Formel
        falls Geschwindigkeit die gesetzte Grenze überschreitet,
        setze sie auf maximal/minimal mögliche Geschwindigkeit**;**
        aktualisiere die beste erreichte Position dieses Partikels**;**
    end
    aktualisiere die beste erreichte Position aller Partikel**;**
end

Die im Algorithmus erwähnte PSO-Formel lautet nach [EKS01]:

$$\vec{v}_i(t) = \vec{v}_i(t-1) + \varphi_1(\vec{p}_i - \vec{x}_i(t-1)) + \varphi_2(\vec{p}_g - \vec{x}_i(t-1))$$
$$\vec{x}_i(t) = \vec{x}_i(t-1) + \vec{v}_i(t)$$

Hierbei werden die Geschwindigkeit $\vec{v}_i(t)$ und die Position $\vec{x}_i(t)$ des Partikels i zum Zeitpunkt t berechnet. In die Berechnung gehen seine Position und Geschwindigkeit zum Zeitpunkt $t-1$ sowie die beste bisher gefundene Position dieses Partikels p_i und aller Partikel p_g ein. Die Parameter φ_1 und φ_2 steuern den Einfluss oben genannter Positionen.

Folgende Klassen sind bei der Implementierung entstanden:

- `StrategyStandardPSO` enthält Strategieparameter von einem einzelnen Partikel. Dahin gehört die besterreichte Position von dem Partikel mit diesen Strategiewerten.

- `IndividualPSO` implementiert einen Partikel.

- `PopulationPSO` ist für den Schwarm (Population der Partikel) zuständig. Dahin gehört die beste erreichte Position aller Partikel.

- `StandardPSO` ist der PSO-Algorithmus.

Parameter

Je nach Problem/Testfunktion muss man die `Problem dimension` einstellen. Die Größe des Schwarms `Swarm size` ist in vielen Fällen entscheidend. Zu wenige Partikel nähern sich zu langsam dem Optimum, da sie dem Suchraum nicht so schnell erforschen können. Daraus folgt auch, dass sie oft in lokalen Optima hängen bleiben. Andererseits können zu viele Partikel den Algorithmus aufgrund vieler durchzuführenden Berechnungen verlangsamen.

`Velocity bound` stellt sicher, dass die Geschwindigkeiten der Partikel nicht zu groß werden. Zu kleine Werte erschweren es den Partikeln, schnell nach Optima zu suchen, zu große Werte können die Partikel dazu bringen, sich voneinander zu entfernen.

Die PSO-Formel benutzt bei Berechnung `Inertia weight`, `Cognitive term` und `Social term`, die Einstellung dieser Parameter kann für einige Testfunktionen deutliche Verbesserungen bringen, der Algorithmus optimiert jedoch gut mit Standardwerten.

`Initialisation from` und `Initialisation to` setzen die Grenzen für mögliche Positionen der Partikel bei der Initialisierung. Die Positionen werden zufällig, aber unter Berücksichtigung dieser Grenzen gesetzt. Falls noch allgemeine Grenzen für Positionen des Partikels gewünscht sind, kann man die Werte von `Maximal solution` und `Minimal solution` entsprechend setzen. Die Parameter und deren Wirkung sind in [EKS01] beschrieben, eine übersichtliche Zusammenfassung findet sich auch unter [Hu03].

Parameterübersicht

Parameter	Typ	Default	Bereich	Kurzbeschreibung
Problem dimension	integer	3	$x > 0$	Dimension des Problems
Swarm size	integer	50	$x > 0$	Anzahl der Partikel im Schwarm
Maximal solution	double	0.0		Maximal zulässige Lösung (Position des Partikels), wobei 0.0 bedeutet, dass es keine Grenzen gibt
Minimal solution	double	0.0		Minimal zulässige Lösung (Position des Partikels), wobei 0.0 bedeutet, dass es keine Grenzen gibt
Initialisation from	double	100		Maximal zulässige Lösung (Position des Partikels) bei der Initialisierung
Initialisation to	double	-100		Minimal zulässige Lösung bei der Initialisierung
Inertia weight	double	1.2	$x > 0$	Trägheitswert, legt den Einfluss der vorherigen Geschwindigkeit auf die aktuelle fest
Cognitive term	double	2.02	$x > 0$	Einfluss der besten gefundenen Position eines Partikels auf sein weiteres Verhalten
Social term	double	2.02	$x > 0$	Einfluss der besten gefundenen Position aller Partikel auf das weitere Verhalten des einzelnen Partikels
Velocity bound	double	1.0	$x > 0$	Geschwindigkeitsbegrenzung für die Partikel

3.6 Probleme

3.6.1 Temperierbohrungen

Beschreibung

Ein optimales Auslegen der Temperierbohrungen von Spritz- und Druckgusswerkzeugen (siehe [MWM02]) ist essentiell, sowohl für die Minimierung der Kühlzykluszeit eines Werkzeugs, als auch um die Werkstückqualität zu erhöhen und die Wahrscheinlichkeit des Werkzeugbruchs durch temperaturbedingte Spannungen zu verringern. Das optimale Layout der Temperierkreisläufe ist im Allgemeinen eine recht komplizierte Aufgabe, da die Werkzeuge und Formflächen häufig sehr komplex strukturiert sind. Es müssen gleichzeitig verschiedene Anforderungen, wie eine gute Formflächenapproximation oder eine gleichmäßige Werkzeugtemperierung bei einer geringen Anzahl von Bohrungen, möglichst gut erfüllt werden.

Die Qualität des Formteils hängt von einer möglichst gleichmäßigen und werkstoffspezifisch optimalen Temperierung ab. Eine zu niedrige Werkzeugtemperatur verringert zwar die Zykluszeiten, verringert aber eventuell auch die Werkstückqualität. Eine zu hohe Werkzeugtemperatur hingegen verlängert die Abkühlzeit und kann das Bauteil ebenfalls schädigen. Eine optimale Werkzeugtemperierung richtet sich nach den spezifischen Werkstoffeigenschaften, der Struktur der Formfläche sowie der zu erwartenden Schwindung und Eigenspannung im Werkstück nach dem Entformen. Die Ziele einer Werkzeugtemperierung sind somit

- eine mittlere Formnest-Wandtemperatur,
- eine gleichmäßige Temperaturverteilung sowie
- eine kurz gehaltene Kühlzeit, welche genau einzuhalten ist.

Das Temperieren von Druckguss- und Spritzgusswerkzeugen wird über ein System von Kanälen realisiert, durch die ein Medium (beispielsweise Wasser oder eine Emulsion) fließt. Diese Kanäle werden als Tiefbohrungen mit Hilfe spezieller Tiefbohrmaschinen gefertigt. Es gilt, mit möglichst wenigen Bohrungen auszukommen. Einzelne Bohrungen lassen sich zu Temperierbohrungszyklen kombinieren. Hierbei ist es möglich, einzelne Bohrungssegmente mit dem Verwenden von Stopfen mehrfach zu nutzen.

Die Temperierbohrungen werden heutzutage meist von der Erfahrung des Konstrukteurs bestimmt. Er stützt sich dabei hauptsächlich auf sein Erfahrungswissen. Ein intuitives Maß zur Beurteilung der Qualität der Temperierbohrungen stand bisher nicht zur Verfügung. Entsprechend gibt es auch keine Algorithmen zur automatischen oder halbautomatischen Auslegung guter Temperierbohrungen.

Das Plug-In

Um das Temperierbohrungsproblem in MooN integrieren zu können, wurde eine externe Software hinzugezogen, die uns vom Institut für Spanende Fertigung (ISF) für unsere Arbeit zur Verfügung gestellt wurde. Es handelt sich um die Simulationssoftware „Evolver“, die aus einem gegebenen Kühlkreislauf verschiedene Gütemaße für die resultierende Kühlung berechnet. Um diese Software nutzbar zu machen, haben wir eine einfache Adapterklasse geschrieben, die über eine Socketverbindung mit dem Evolver kommuniziert.

Zudem wird aus den zurückgelieferten Gütemaßen mittels einer gewichteten Summe ein (unikriterieller) Fitnesswert berechnet.

Der Evolver

Das Programm Evolver bietet die Möglichkeit, die Temperierbohrungen bei Spritz- und Druckgusswerkzeugen zu untersuchen und damit bessere Lösungen zu produzieren. Der Software wird über eine Socketverbindung eine Anzahl von Stützpunkten im Raum mittels ihrer dreidimensionalen Koordinaten übergeben. Diese werden als Stützstellen eines Polygonzugs interpretiert, der den aus Bohrkanälen zusammengesetzten Kühlkreislauf darstellt. Dabei gibt es einige Restriktionen, die man nicht außer Acht lassen darf.

- Die Kanäle dürfen sich nicht überschneiden, damit die Kreisstruktur erhalten bleibt.

- Ein Mindestabstand zwischen Bohrungen und Objekt muss eingehalten werden.

Es wäre wünschenswert, die Wärmeleitung und die Abkühlung beobachten zu können, wie sie sich mit verschiedenen Bohrungen verbessern oder verschlechtern. Das ist zwar technisch möglich, jedoch benötigt so eine Berechnung eine enorme Rechenleistung. Evolver macht sich hierbei einen Trick von Nutzen. Es wird statt der Wärme ein Lichtstrahl beobachtet, der sich einem Wärmestrahl sehr ähnlich verhält. Die Helligkeit repräsentiert dabei die Wärme. Der Vorteil von diesem Verfahren ist die geringere Rechenzeit. So kann der Evolver in kurzer Zeit sehr viele Bohrungsmöglichkeiten simulieren. Die Ergebnisse der Simulation werden über die Socketverbindung als ein Vektor von Gütemaßen zurückgeliefert.

Parameter

Die Parameterschnittstelle muss die Kommunikation zwischen dem Plug-In und dem Simulationsprogramm ermöglichen. Dazu müssen die `IP address` und der `Port` angegeben werden. Außerdem muss dem Evolver die Anzahl der Datenpunkte pro Individuum bekannt sein, d. h. die `Number of points`.

Parameterübersicht

Parameter	Typ	Default	Bereich	Kurzbeschreibung
IP address	String	„127.0.0.1“	IP-Adressen	Internetadresse des Rechners, auf dem der Evolver läuft
Port	integer	6201		Port, auf dem der Evolver läuft
Number of points	integer	2	$x > 0$	Anzahl der Stützpunkte für den Kühlpolygonzug

3.6.2 Fahrstuhlproblem

Beschreibung

Das Fahrstuhlproblem beschäftigt sich damit, eine Fahrstuhlsteuerung in einem Gebäude mit mehreren Stockwerken und Fahrstühlen zu optimieren. Der Projektgruppe wurde ein am Lehrstuhl entwickeltes SRing-Modell zur Verfügung gestellt, so dass lediglich eine Adapterklasse implementiert werden musste, um das Problem an MooN anzupassen. Das zu optimierende Kriterium im SRing ist die durchschnittliche Wartezeit der Passagiere. Weitere Informationen zum SRing-Modell findet man in [MAB+01].

Parameter

`Floors number` und `Servers number` beschreiben das Gebäude (die Anzahl der Etagen bzw. der Aufzüge), `Arrival rate` gibt an, wie wahrscheinlich es ist, dass in einem Zeitschritt ein Passagier ankommt. In jedem Zeitschritt wird eine Etage betrachtet, danach geht es zur nächsten usw., bis alle Etagen betrachtet wurden. Für eine mögliche Lösung, die Gewichte für ein neuronales Netz darstellt, werden `Iterations number` Iterationen vom SRing durchgeführt.

Parameterübersicht

Parameter	Typ	Default	Bereich	Kurzbeschreibung
Iterations number	integer	1000	$x > 0$	Anzahl der Iterationen von SRing für jede Generation
Floors number	integer	6	$x > 0$	Anzahl der Etagen im Gebäude
Servers number	integer	2	$x > 0$	Anzahl der Fahrstühle
Arrival rate	double	0.3	$0 \leq x \leq 1$	Durchschnittliche Ankunftsrate der Passagiere

3.6.3 Pickup and Delivery Problem

Beschreibung

Das Vehicle Routing Problem (VRP) ist eine der schwierigsten kombinatorischen Optimierungsaufgaben. Ausgangssituation ist eine Menge von Kunden, die von einem Depot aus beliefert werden sollen. Die Entfernungen (=Kosten) zwischen den Kunden und dem Ausgangspunkt, dem Depot, sind hierbei bekannt. Das Lösen eines VRP beinhaltet die Berechnung einer Menge von Routen eines LKWs. Diese Routen sind so ausgelegt, dass alle Kunden genau einmal angefahren werden, sodass deren Bedarfe gedeckt werden. Das Abliefern der Waren wird erschwert durch Restriktionen, und zwar der Kapazität des LKW und einer Zeitschranke. Ziel ist es, eine Route mit minimalen Kosten zu finden. Das Pickup and Delivery Problem ist ein Spezialfall des VRP. Es erfordert zusätzlich zum Abliefern das Abholen von Waren unter den gleichen Restriktionen. Daher hat das Pickup and Delivery Problem zwei zusätzliche Kapazitäten:

- Pickup Capacity
- Delivery Capacity

Die Summe dieser Kapazitäten darf nicht größer als die gesamte Kapazität sein. Es wird stets zunächst geliefert und dann abgeholt. Dies geschieht, damit die Kapazität des Fahrzeugs sich vergrößert und damit mehr abgeholt werden kann. Man kann so eine vorzeitige Verletzung der Kapazitätsrestriktionen vermeiden. Falls diese verletzt werden, muss man zum Lager zurückfahren. Die Kosten vom Kunden i zum Lager und die Kosten vom Lager zum Kunden $i+1$ werden dann zu den Kosten der Route hinzuaddiert. Auch wenn die Zeitrestriktion verletzt wird, muss wieder zum Lager gefahren werden. In dem Fall werden alle Werte auf den Anfangszustand zurückgesetzt.

Benutzer von MooN müssen beachten, dass Heuristiken die Dimension des Problems auf $2 * Customernumber - 1$ setzen, da im schlimmsten Fall die Tour (Permutation über Kunden und Lager) so verläuft, dass der LKW nach jedem Kunden wieder zum Lager fährt. Bei der Initialisierung des Problems wird aus den angegebenen Koordinaten in der Ebene eine Distanzmatrix zwischen den Kunden gebildet.

Es folgt eine Übersicht über die implementierten Klassen:

- `CustomerInfoType` speichert die Informationen über einen einzelnen Kunden: seine Koordinaten, Produktmengen zur Abholung und Lieferung und die Zeit der Bedienung.
- `ResourceVector` ist eine Hilfsklasse zur Speicherung der Abhol- und Lieferkapazitäten bei einem Kunden oder einem LKW.
- `PickupAndDeliveryProblem` ist die eigentliche Problemklasse.

Parameter

Die unter `Customer informations file` angegebene Datei enthält alle Kundeninformationen. Das ist eine Textdatei, bei der jede Zeile aus fünf durch Leerzeichen getrennten Zahlen besteht: x-Koordinate, y-Koordinate, Abholmenge, Liefermenge und Zeit der Bedienung. Beispiel:

```
25 25 0 5 15.5
25 50 5 0 20.0
0 0 0 0 0.0
```

Am Ende, hier als dritter „Kunde“, muss das Lager mit seinen Koordinaten angegeben werden.

Da die Lösungen des Problems (Permutationen) von der Kundenzahl abhängen, gibt `Customer number` diese Zahl an. Zudem beschreibt `Maximal operation time` die maximale Tourlänge und `Maximal vehicle capacity` die maximale Kapazität des LKWs.

Parameterübersicht

Parameter	Typ	Default	Bereich	Kurzbeschreibung
Customer informations file	File	pad.para		Datei mit Koordinaten
Customer number	integer	50	$x > 0$	Anzahl der Kunden
Maximal operation time	double	200.0	$x > 0$	Maximale Tourlänge
Maximal vehicle capacity	integer	160	$x > 0$	Maximale Kapazität des LKWs

3.6.4 Travelling Salesperson Problem

Beschreibung

Das Travelling Salesperson Problem (TSP) ist ein bekanntes Optimierungsproblem, das aus der Berechnung einer optimalen Städtetour besteht, wobei jede Stadt nur einmal besucht werden darf. Das Ziel ist die Minimierung der Kosten, also der Summe der Wege zwischen den Städten einer Tour. Für MooN haben wir ein symmetrisches TSP implementiert, d. h. der Weg zwischen zwei Städten ist in beiden Richtungen gleich teuer. Das von uns erstellte TSP besteht aus zwei Klassen. In `NodesInfoType` werden die Koordinaten der Städte gespeichert, während `TspProblem` die eigentliche Problemklasse darstellt. Damit die gewählten Heuristiken auf dem TSP richtig arbeiten können, müssen deren Dimensionen auf `Nodes number` gesetzt werden, damit die Permutation der Städte jede Stadt genau einmal enthält.

Parameter

Die unter `Nodes informations file` angegebene Datei enthält alle Koordinaten der Städte. Diese werden in einer Textdatei gespeichert, wobei jede Zeile die x- und y-Koordinaten einer Stadt enthält. Der Parameter `Nodes number` hingegen speichert die Städtezahl, da diese die Länge der Lösungspermutation bestimmt.

Parameterübersicht

Parameter	Typ	Default	Bereich	Kurzbeschreibung
Nodes informations file	File	pad.para		Datei mit Koordinaten der Städte
Nodes number	integer	50	$x > 0$	Anzahl der Städte

3.6.5 Kugelfunktion

Beschreibung

Die Kugelfunktion ist die einfachste aller Testfunktionen. Sie ist stetig, konvex, quadratisch und unimodal, weshalb jedes brauchbare Optimierungsverfahren das globale Minimum schnell finden sollte. Diese und weitere mehrdimensionale Testfunktionen sind in [Bäc93] beschrieben.
Die Kugelfunktion

$$f(\vec{x}) = \sum_{i=1}^{n} x_i^2$$

hat ihr Minimum $f(\vec{x}) = 0$ bei $x_i = 0 \ \forall i$.

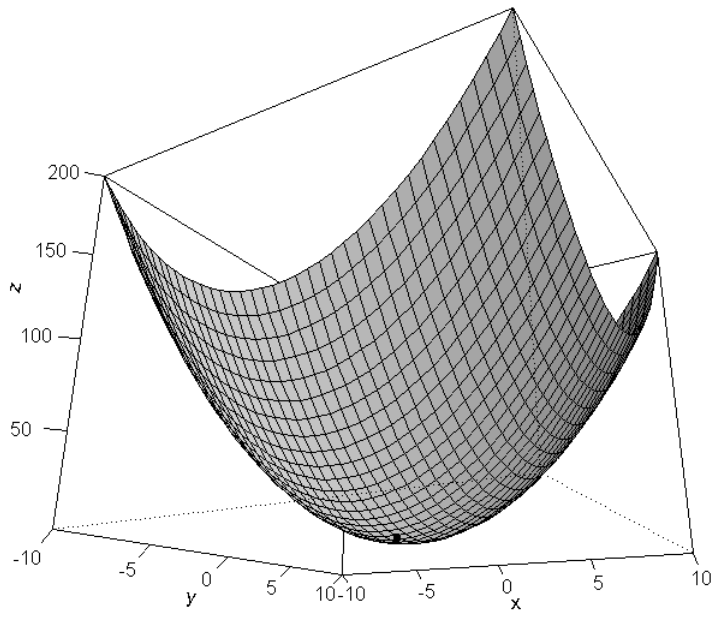

Abbildung 3.3: Die 2-dimensionale Kugelfunktion.

Parameter

Dieses Plug-In hat keine Parameter.

3.6.6 Schwefelfunktion

Beschreibung

Die von uns implementierte zweite von Schwefels Testfunktionen wird als „Boshafte Schwefelfunktion“ bezeichnet, da sie ebenso wie Shekels Fuchsbauten oder die Rastrigin-Funktion extra darauf angelegt ist, Optimierungsverfahren in die Irre zu führen. Sie ist zwar stetig, aber hochgradig nichtlinear, multimodal und besitzt viele lokale Minima mit nahezu gleichem Funktionswert. Wir haben eine leicht modifizierte Variante implementiert, sodass das globale Minimum einen Funktionswert sehr nahe an 0 besitzt:

$$f(\vec{x}) = 418.9829n - \sum_{i=1}^{n} x_i sin(\sqrt{|x_i|})$$

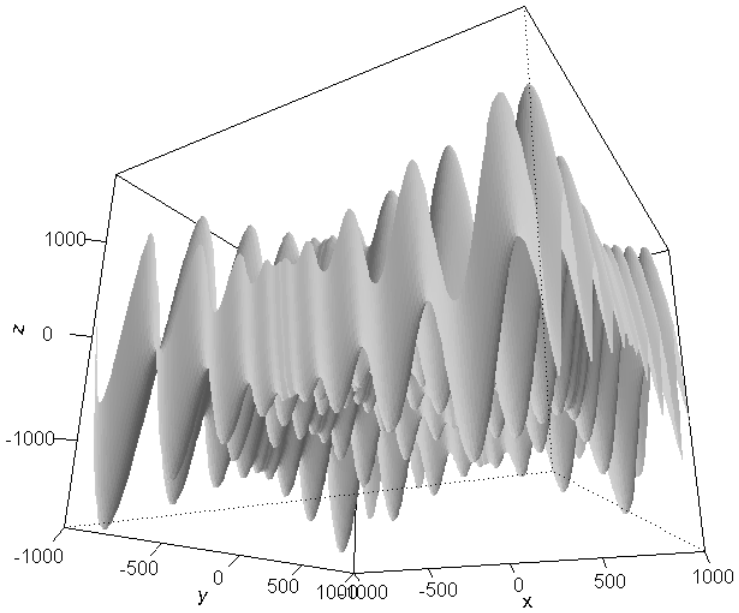

Abbildung 3.4: Die 2-dimensionale Schwefel-Funktion.

Parameter

Dieses Plug-In hat keine Parameter.

3.6.7 Rastriginfunktion

Beschreibung

Die Rastriginfunktion basiert zwar auf der einfachen Kugelfunktion, ist jedoch durch einen zusätzlichen cosinus-basierten Modulationsterm multimodal. Das führt zu sehr vielen kleinen Tälern und Hügeln, was die Suche nach dem globalen Minimum erheblich erschwert.
Die Funktion

$$f(\vec{x}) = 10n + \sum_{i=1}^{n}(x_i^2 - 10cos(2\pi x_i))$$

hat ihr globales Mininum $f(\vec{x}) = 0$ bei $x_i = 0\ \forall i$.

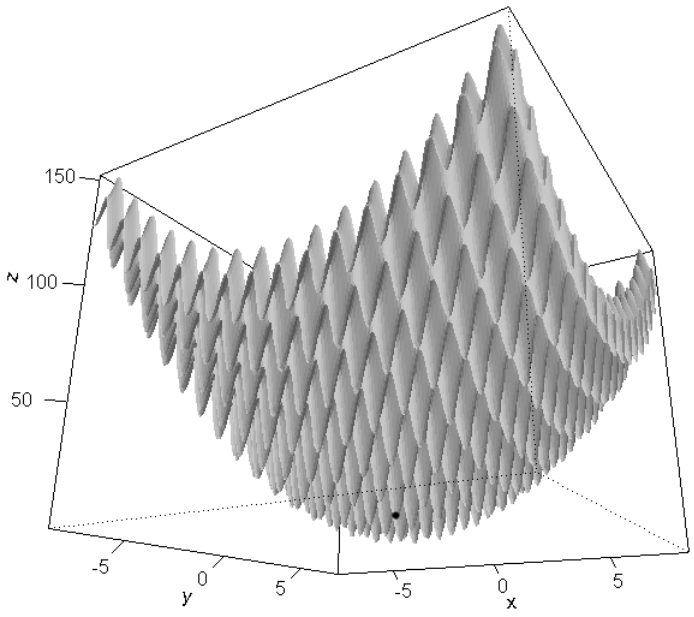

Abbildung 3.5: Die 2-dimensionale Rastrigin-Funktion.

Parameter

Dieses Plug-In hat keine Parameter.

3.6.8 Griewankfunktion

Beschreibung

Die Griewankfunktion ist, wie auch die Rosenbrockfunktion (siehe 3.6.10), eine multimodale Testfunktion für die Minimierung. Obwohl die Anzahl der lokalen Minima mit der Dimension exponentiell wächst, ist das globale Minimum umso leichter zu finden, je höher die Dimension gewählt wird.

Da diese Funktion vorwiegend zum Testen von populationsbasierten Heuristiken wie PSO verwendet wird, haben wir uns entschlossen, sie neben der Rosenbrock-Funktion ebenfalls in MooN zu implementieren. Hierbei wurde die Version aus [LRK01] implementiert, die ein globales Minimum mit dem Wert 0 bei (100,...,100) besitzt:

$$f(\vec{x}) = \frac{1}{4000}\sum_{i=1}^{n}(x_i - 100)^2 - \prod_{i=1}^{n} cos(\frac{x_i - 100}{\sqrt{i}}) + 1$$

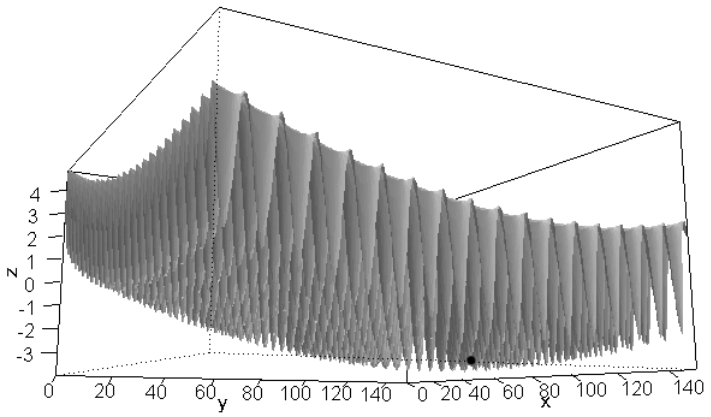

Abbildung 3.6: Die Griewankfunktion für 2 Dimensionen.

Parameter

Dieses Plug-In hat keine Parameter.

3.6.9 Michalewiczfunktion

Beschreibung

Michalewicz und Schoenauer schlagen in [MS96] einen Satz von 11 Funktionen vor, genannt G1-G11, die jeweils durch eine Menge von Nebenbedingungen eingeschränkt sind. Solche Nebenbedingungen kommen in der Praxis oft vor. Deshalb ist es sinnvoll, sich mit eben solchen Funktionen auseinander zu setzen und möglichst gute Wege zu finden, die Nebenbedingungen in die Optimierung mit aufzunehmen. Wir haben uns hierbei exemplarisch für die Erste dieser 11 Funktionen entschieden. Die G1 Funktion ist ein Minimierungsproblem der Dimension 13, das auf [FP90] zurückgeht.
Sie ist definiert als:
$G1(\vec{x}) = 5x_1 + 5x_2 + 5x_3 + 5x_4 - 5\sum_{i=1}^{4} x_i^2 - \sum_{i=5}^{13} x_i$
und es sollen möglichst viele der folgenden 9 Nebenbedingungen eingehalten werden:

$$1.\ 2x_1 + 2x_2 + x_{10} + x_{11} \leq 10$$
$$2.\ 2x_1 + 2x_3 + x_{10} + x_{12} \leq 10$$
$$3.\ 2x_2 + 2x_3 + x_{11} + x_{12} \leq 10$$
$$4.\ -8x_1 + x_{10} \leq 0 \qquad 5.\ -8x_2 + x_{11} \leq 0$$
$$6.\ -8x_3 + x_{12} \leq 0 \qquad 7.\ -2x_4 - x_5 + x_{10} \leq 0$$
$$8.\ -2x_6 - x_7 + x_{11} \leq 0 \qquad 9.\ -2x_8 - x_9 + x_{12} \leq 0$$

Dabei ist $0 \leq x_i \leq 1, i = 1, \ldots, 9$ und $0 \leq x_i \leq 100, i = 10, \ldots, 13$;
Das globale Minimum der Funktion liegt bei:

$$\vec{x} = (1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 3, 3, 1)$$

Dort ist $G1(\vec{x}) = -15$ und sechs der neun Bedingungen sind erfüllt, nämlich alle außer Bedingungen Nr. 4, 5 und 6.

Bei unserer Beispielimplementierung wurden die Nebenbedingungen durch eine Straffunktion realisiert. Diese berechnet für jede Nebenbedingung den Term auf der linken Seite. Als Strafwert, der zum Fitnesswert hinzuaddiert wird, wurde die Distanz des Ergebnisses vom erlaubten Bereich gewählt.

Parameter

Dieses Plug-In hat keine Parameter.

3.6.10 Rosenbrockfunktion

Beschreibung

Die Rosenbrockfunktion ist eine multimodale Testfunktion für die Minimierung. Sie ist in hohem Maße nichtlinear und konvergiert extrem langsam bei Methoden wie dem steilsten Abstieg. Sie wurde für die Implementierung im Rahmen von MooN ausgewählt, um die realisierten Heuristiken (vor allem PSO) hinsichtlich ihrer Performance mit Implementierungen aus der Literatur vergleichen zu können. Für MooN wurde die bezüglich der Dimension verallgemeinerte Version implementiert. Sie entspricht der Definition aus [LRK01] und hat ein globales Minimum mit dem Funktionswert 0 bei (1,...,1):

$$f(x) = \sum_{i=1}^{n-1} (100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2)$$

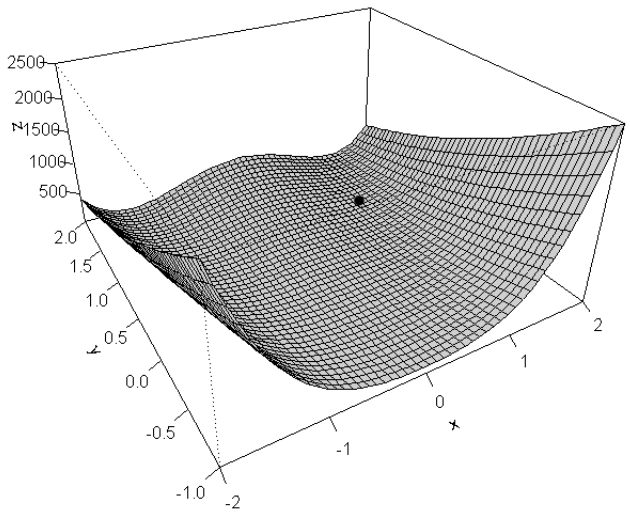

Abbildung 3.7: Die Rosenbrockfunktion für 2 Dimensionen.

Parameter

Dieses Plug-In hat keine Parameter.

3.6.11 Permutation Sort Problem

Beschreibung

Das Permutation Sort Problem ist ein einfaches Testproblem für Heuristiken, deren Lösungsvektoren aus Permutationen bestehen. Der Fitnesswert eines Individuums entspricht der Anzahl der Fehlstellen, d. h. wie oft eine niedrigere Zahl auf eine höhere folgt (bei aufsteigender Sortierung). Das globale Minimum dieser Funktion tritt demzufolge bei einer vollständig geordneten Permutation auf. Die sechsstellige Permutation `1 2 3 6 4 5` hat beispielsweise aufgrund der beiden Fehlstellen `6 4` und `6 5` den Fitnesswert 2.

Parameter

Dieses Plug-In hat keine Parameter.

3.7 Abbruchbedingungen

3.7.1 Anzahl Generationen

Beschreibung

Die `ExitConditionNumberOfGenerations` stellt fest, ob in einem Einzellauf eine vorgegebene Anzahl von Generationen berechnet wurde. Ein Abbruch erfolgt, wenn die eingestellte Anzahl erreicht oder überschritten wurde. Diese Information stammt aus einer Methode im Interface `Heuristic`. Von dieser Abbruchbedingung wird in der Literatur zu Heuristiken sehr oft Gebrauch gemacht, da sie eine vorhersehbare Länge des Laufs garantiert. Von Nachteil ist, dass mit dieser Bedingung keine Garantie für die Güte der Lösung gegeben wird, da die Resultate des Laufs nicht berücksichtigt werden.

Parameter

Einziger Parameter dieses Plug-Ins ist die Anzahl der in dem Lauf zu berechnenden Generationen, mit Defaultwert 100.

Parameterübersicht

Parameter	Typ	Default	Bereich	Kurzbeschreibung
Number of generations	integer	100		Anzahl der zu berechnenden Generationen

3.7.2 Zeitlimit

Beschreibung

Um einem Einzellauf eine festgelegte Zeitspanne zur Optimierung zur Verfügung zu stellen, steht die `ExitConditionPresetTime` zur Verfügung. Sie wird mit einem Zeitpunkt, modelliert durch ein Objekt des Typs Date, initialisiert. Die Abbruchbedingung ist erfüllt, wenn die aktuelle Systemzeit später als die voreingestellte Zeit ist. Da die Abbruchbedingung nur nach vollen Generationen überprüft wird, kann die tatsächliche Lauflänge aber länger sein als eingestellt, jedoch maximal um die Dauer einer Generation.

Der Einsatz dieser Bedingung ist sinnvoll, wenn nur ein begrenztes Zeitkontingent zu Verfügung steht und man nicht genau weiß, wie viel Zeit eine Generation benötigt. Zudem kann sie benutzt werden, um verschiedene Heuristiken fair zu vergleichen, vorausgesetzt, beide erhalten in den ihnen zugeteilten Zeitspannen den gleichen Anteil an Prozessorzeit.

Parameter

Der einzige Parameter ist die Stoppzeit. Diese wird auf 5 Minuten nach dem Zeitpunkt der Initialisierung voreingestellt.

Parameterübersicht

Parameter	Typ	Default	Bereich	Kurzbeschreibung
Termination time	java.util.Date	Dynamisch		Der Zeitpunkt, an dem Optimierung abbricht

3.7.3 Fitnessgrenzwert

Beschreibung

Oft ist es nützlich zu überprüfen, ob eine Heuristik auf einer bekannten Probleminstanz eine gewisse Güte erreichen kann. Hierzu wurde die Abbruchbedingung `ExitConditionFitnessThreshold` realisiert. Die Bedingung vergleicht die beste bis jetzt gefundene Fitness mit einem vorgegebenen Wert. Sie ist vor allem in Benchmark-Situationen sinnvoll.

Parameter

Die Fitnessgrenze ist als reeller Wert einstellbar. Zudem kann gewählt werden, ob es sich um eine obere oder untere Grenze handelt, d. h. ob der Fitnesswert über- oder unterschritten werden soll. Als Defaultwert ist „untere Grenze“ eingestellt.

Parameterübersicht

Parameter	Typ	Default	Bereich	Kurzbeschreibung
Threshold	double	0.0		Fitnessgrenze
Lower bound	boolean	true		Obere oder untere Grenze

3.7.4 Fitnessänderung

Beschreibung

Die `ExitConditionChangeOfFitnessThreshold` bietet die Möglichkeit, einen Einzellauf abzubrechen, wenn sich die beste Lösung seit einer bestimmten Anzahl von aufeinander folgenden Generationen nicht um mehr als einen festgelegten Schwellenwert verbessert hat.

Nach jeder Generation wird die aktuell beste Lösung mit derjenigen der vorherigen Generation verglichen. Hat sich der Fitnesswert nicht um den vorgegebenen Schwellenwert verbessert, wird ein Zähler gestartet, der allerdings wieder auf 0 zurückgesetzt wird, wenn sich der Fitnesswert in einer späteren Generation doch um mehr als den Schwellenwert verringert hat. Die Abbruchbedingung ist also erst dann erfüllt, wenn sich die beste Lösung in einer vorgegebenen Zahl aufeinanderfolgenden Generationen nicht um den Schwellenwert verbessert hat.

Diese ExitCondition bietet sich vor allem für Heuristiken an, deren Lösungen sich nach einer bestimmten Generationenzahl nur noch sehr geringfügig verbessern. Somit kann vermieden werden, dass unnötige Iterationen durchlaufen werden, die fast nichts mehr zur Verbesserung des Fitnesswerts beitragen.

Parameter

Als Parameter werden der Schwellenwert, um den sich die Lösungen verbessern sollen (Defaultwert 0.0005), sowie die Anzahl der Generationen, die für

diese Verbesserung zur Verfügung stehen, benötigt. Der letzte Parameter ist standardmäßig auf 50 eingestellt.

Parameterübersicht

Parameter	Typ	Default	Bereich	Kurzbeschreibung
Threshold	float	0.00005		Schwellenwert der kleinsten Verbesserung
Generations	integer	50		Anzahl der Generationen, nach denen die Mindestverbesserung erfolgt sein muss

3.8 Datenausgabe

Die als Plug-Ins eingebundenen Heuristiken können sehr unterschiedlich in ihrer Herangehensweise zur Optimierung sein. Dies wirkt sich auch in ihrem inneren Aufbau und den von ihnen genutzten Datenstrukturen aus. Einige Heuristiken enthalten nur ein Individuum, während andere ganze Populationen verwalten. Es können auch ganz andere Repräsentationsformen gewählt werden, um Informationen zu speichern, wie z. B. die Pheromonmatrix bei ACO. Diese Unterschiede erfordern ein flexibles Ausgabekonzept, das „Logging“, um eine Untersuchung der eingesetzten Heuristiken zu ermöglichen, unabhängig von den genutzten Datenstrukturen und Vorgehensweisen. Zum einen muss es ermöglicht werden, beliebige Ausgaben zu generieren. Zum anderen muss dies aber sinnvoll kontrolliert werden können. Um nicht ständig alles loggen zu müssen, sollten zu bestimmten Zeitpunkten bestimmte Information gezielt wegschreibbar sein.

Um das Logging zu strukturieren, wurden Kategorien (categories) eingeführt. Eine Kategorie hat einen eindeutigen Namen und steht für bestimmte Informationen. Ein Beispiel wäre die Kategorie `Current_Best_Individual`, die für die Ausgabe des besten Individuums einer Population stehen könnte. Jede Heuristik kann beliebig viele Kategorien anbieten. Eine Übersicht über diese ist durch eine im Interface definierte Methode auslesbar. Namen und Inhalte der Kategorien sind dabei völlig frei wählbar, um hier größtmögliche Flexibilität zu gewährleisten.

Für jede der Kategorien wird festgelegt, in welchen Intervallen (in Generationen gemessen) die Information durch das zuständige Objekt der Klasse `OutputHandler` in entsprechende Log-Dateien geschrieben werden sollen. Die Einstellung dieser Intervalle ist per GUI oder auch direkt in der XML-Datei

möglich. Weiterhin kann der Name der Datei angegeben werden, in welche die generierten Daten geschrieben werden sollen. Das Format von Zeilen innerhalb der entsprechenden Dateien ist:

```
<Kategoriename> <Daten der Kategorie> <Zeilenende>
```

Es besteht darüber hinaus die Möglichkeit, jede Kategorie in eine separate Datei schreiben zu lassen, wenn dies z. B. für die weiteren Analysen in Statistikprogrammen von Vorteil ist.

3.9 Laufzeitvisualisierung

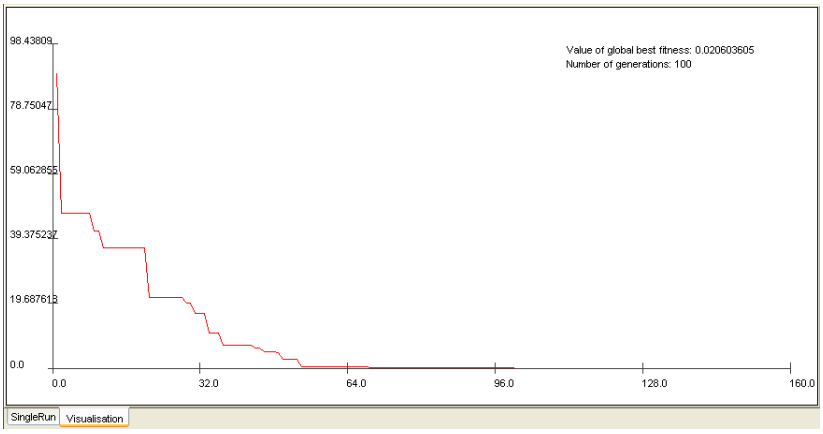

Abbildung 3.8: Laufzeitvisualisierung in MooN.

Bereits einzelne Läufe können sehr viel Zeit in Anspruch nehmen. Mitunter ist aber nicht sicher, ob die gewählten Einstellungen einer Heuristik überhaupt zu einer Optimierung auf dem Problem führen. Damit ergab sich die Anforderung, bereits während der Durchführung einzelner Läufe Informationen dazu anzuzeigen, soweit sie verfügbar sind. Von Interesse sind vor allem die bisher besten gefundenen Fitnesswerte über den durchgeführten Generationen. Dies ermöglicht zum einen eine schnelle visuelle Erfassung, ob überhaupt eine Optimierung stattfindet. Zum anderen erlaubt es auch eine grobe Abschätzung der Optimierungsgeschwindigkeit. Lineare Verbesserungen lassen sich zum Beispiel von hyperbolischen gut unterscheiden. Somit

werden zwei Zielgruppen bedient. Heuristikentwickler können schnell sehen, ob ihre Plug-Ins optimieren. Heuristikanwender können überprüfen, ob die Verläufe der Optimierung den erwarteten Kurven entsprechen. Diese Informationen sind in MooN ohne externe Werkzeuge zu erhalten.

Die Visualisierung wird nach jeder Generation aktualisiert. Hierzu werden die jeweils besten gefunden Fitnesswerte der Heuristik abgefragt. Aus ihnen wird ein Graph mit allen bisher erhaltenen Daten des Laufes erstellt. Eine Skalierung sowie die Erstellung eines passenden Koordinatensystems mit Beschriftung erfolgt automatisch und orientiert sich an den übergebenen Werten. Auf der x-Achse sind die Generationen und auf der y-Achse die Fitness aufgetragen.

3.10 Auswertung mit R

Für die statistischen Auswertungen unserer Experimente haben wir ein für wissenschaftliche Analysen gängiges Tool ausgewählt — *R*. Unsere Zielsetzung war schließlich unter anderem zu verifizieren, dass sich MooN, bzw. die Ausgaben von MooN, zur Analyse in gängigen Programmen eignen.

Dabei stellte sich das sehr anpassungsfähige Output-Handler-Konzept, das näher in Kapitel 3.8 beschrieben wird, als besonders hilfreich heraus. Bei der Auswahl der Experimentiermethode (Design of Experiments mit faktoriellem Design) ergaben sich z. B. als weitere Anforderungen an die Ausgabe, dass die Parameterwerte mit ausgegeben werden. Dies ließ sich schnell und mit minimalem Aufwand in alle Heuristiken einbauen.

Die Ausgabe war dann eine Datenmatrix, in deren Zeilen jeweils das beste gefundene Individuum eines Laufs und die zu diesem Lauf gehörenden Parametereinstellungen standen. Das weitere Vorgehen mit dieser Ausgabe sah wie folgt aus:

1. Die Ausgabedatei wurde mit `read.table()` eingelesen und die Spalten benannt.

2. Nun folgte die erste Visualisierung der Ergebnisse mittels Histogrammen (`hist()`). Damit konnte eine ungefähre Normalverteilung der Daten abgeschätzt werden. Andererseits zeigten sich zum Teil hier schon auffällige Häufungen.

3. Wenn eine grobe Normalverteilung der Ergebnisse angenommen werden konnte, versuchten wir eine Einpassung in ein lineares Modell mit Hilfe von `lm()`. Die so gewonnenen Koeffizienten einer Regressionsgleichung gaben dann idealerweise einen signifikanten Aufschluss darüber, welche Parameter den stärksten Einfluss hatten. Optisch unterstützend waren in diesem Fall Plots der Halb-Normalen-Quantilen.

4. Wenn die Verteilungsannahme für die lineare Regression nicht erfüllt war, wählten wir als alternatives Analyseverfahren die Methode der Regressionsbäume aus. Dazu musste das Paket `rpart` eingebunden werden (mit `library(rpart)`). Die Konstruktion eines Regressionsbaums für den Einfluss der Parameter A, B, C, D und E auf das Ergebnis Y erfolgte beispielsweise mit

   ```
   regTree <- rpart(Y ~ A + B + C + D + E)
   ```

   die anschließende Ausgabe (auf dem Bildschirm) mit

   ```
   post(regTree, filename="")
   ```

Für weitergehende Details über Regressionsanalysen mit `R` sei auf ([Far02]) verwiesen.

Kapitel 4

Anwendung von MooN

Nach der Planung und Implementierung von MooN stellte die Durchführung und Auswertung verschiedener Optimierungsläufe einen weiteren Teil der PG-Arbeit dar. Bei der im Folgenden beschriebenen Planung dieser Läufe, welche MooN auf den Einsatz in wissenschaftlicher Umgebung prüfen sollten, mussten verschiedene Anforderungen berücksichtigt werden. Zum einen sollte das Testprogramm eine möglichst aussagekräftige Auswertung der Ergebnisse erlauben, andererseits durfte es nicht so umfangreich werden, dass eine Sichtung der Ergebnisse den Rahmen der Projektgruppe gesprengt hätte. Letztlich sollte das Testprogramm die Mindestanforderungen des PG-Antrags abdecken.

4.1 Statistische Auswertungen

In diesem Kapitel werden statistische Methoden beschrieben, um von MooN erzeugte Ergebnisdaten zu untersuchen. Sie wurden von uns für die Auswertung der Versuchsergebnisse benutzt.

4.1.1 Einfache statistische Methoden und Visualisierungen

Histogramme

Um sich einen Überblick über die Daten zu verschaffen, ist oft schon eine Betrachtung der Werteverteilung mittels eines Histogramms (4.1) hilfreich.

Zum Beispiel können so extreme, ungünstige Parametereinstellungen entdeckt werden. Derartige Einstellungen könnten zur Folge haben, dass die guten Ergebniswerte in zwei entfernte Bereiche aufgetrennt werden. In einem solchen Fall ist es sinnvoll, die Ursachen hierfür zu suchen und gegebenenfalls erst für den verantwortlichen Parameter geeignete Werte zu ermitteln, bevor die Analyse mit anderen Methoden fortgeführt wird.

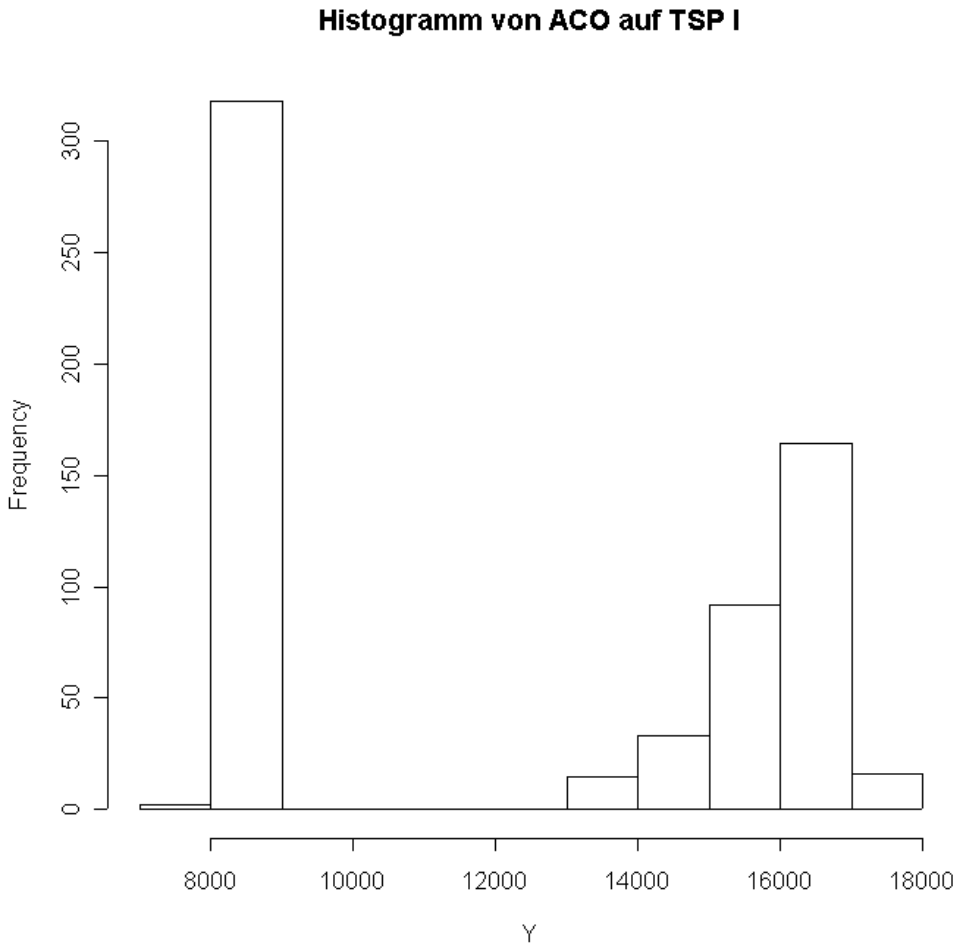

Abbildung 4.1: Beispiel eines Histogramms.

Boxplots

Eine gute Möglichkeit für die Visualisierung der Wertestreuung sind Boxplots (4.2). An ihnen lassen sich mehrere Dinge gut veranschaulichen: die Streuungsbreite, die Verteilung der Werte an sich und die Verteilung im Wertebereich. Zur Erklärung der Elemente: Die Boxen repräsentieren den Bereich, in denen die mittleren 50% der Werte liegen, der Trennstrich in der Box hingegen markiert den Median der Daten. Die Ränder der Boxen werden Quartilen bzw. oberes und unteres Quartil genannt. Die aus der Box heraus ragenden Markierungen, die so genannten „whiskers“, zeigen den Bereich bis zum maximalen bzw. minimalen Datum an, das innerhalb des eineinhalbfachen Abstands zum oberen bzw. unteren Quartil liegt. Alle Werte außerhalb dieses Bereichs werden als Punkte dargestellt und Ausreißer genannt.

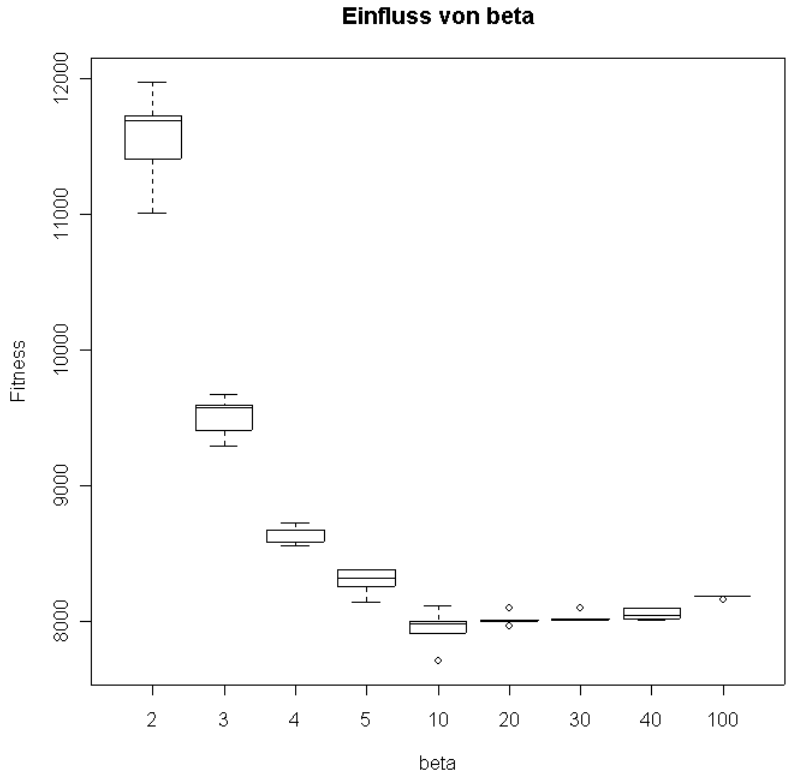

Abbildung 4.2: Beispiel eines Boxplots.

4.1.2 ANOVA (ANalysis Of VAriance)

Wichtigste Prämisse dieser Analyse ist die Annahme einer Normalverteilung der Ergebniswerte. Diese Darstellung der Thematik ist [LK00] entnommen. Ist diese gegeben, kann mit Hilfe eines linearen Modells eine Regressionsgleichung errechnet werden, die so genannte Schätzer für die Gewichtung der Faktoren angibt. Eine solche Regressionsgleichung könnte für vier Faktoren z. B. so aussehen:

$$Y = 1325.25 - 122.23 * A + 0.81 * B + 0.06 * C - 88.73 * D$$

A bis D sind die Faktoren (im Falle einer Heuristik beispielsweise ihre normierten Parameter). An dieser Gleichung ist ablesbar, welcher Faktor den größten Einfluss auf das Ergebnis hat. Die Koeffizienten vor den Faktoren sind die Schätzer.

Für ein Minimierungsziel gilt es, den Wert für Y zu senken. An der Gleichung lässt sich direkt ablesen, in welcher Weise die Parameter für dieses Ziel verändert werden müssen. Faktoren mit negativen Koeffizienten müssen erhöht, Faktoren mit positiven Koeffizienten müssen gesenkt werden. Interpretiert man die Koeffizienten als Vektor, so gibt einem dieser Vektor eine Richtung an, in der die Wahrscheinlichkeit groß ist, bessere Lösungen zu finden. Er gibt die Richtung des „steilsten Abstiegs“ vor.

Außer dem linearen Modell gibt es noch andere, kompliziertere Modelle, um das Verhalten des Funktionswerts in Abhängigkeit von den Faktoren zu beschreiben.

4.1.3 Regressionsbäume

Eine Analysemethode, die auf keinerlei Verteilungsannahmen basiert, ist die der Regressionsbäume (s. Abb. 4.3). Bei dieser Methode wird die Ergebnismenge wiederholt in zwei gleich große Bereiche aufgeteilt, wobei jeweils versucht wird, einen möglichst großen Abstand zwischen den beiden neuen Mengen zu erreichen. Bei jeder Aufteilung wird noch angegeben, welcher Faktor und welche Einstellung für diesen den entscheidenen Einfluss hatte. In den Knoten ist der Mittelwert aller darunter zusammengefassten Elemente (obere Zahl) und die Anzahl der Elemente (n) angegeben.

Da durch dieses Verfahren der einflussreichste Faktor für die erste Teilung verantwortlich ist, lässt sich aus dem Regressionsbaum sofort ablesen, welcher Faktor in welcher Weise als erstes verändert werden sollte. Die Signifikanz der

Veränderungen sinkt beim Durchlaufen des Baumes in Richtung der Blätter. Deswegen ist noch eine weitere Betrachtung notwendig, an der abgelesen werden kann, wie viele Knoten des Baumes relevant sind (genauer gesagt, wie viele Knoten nötig sind, um die Varianz zu erklären).

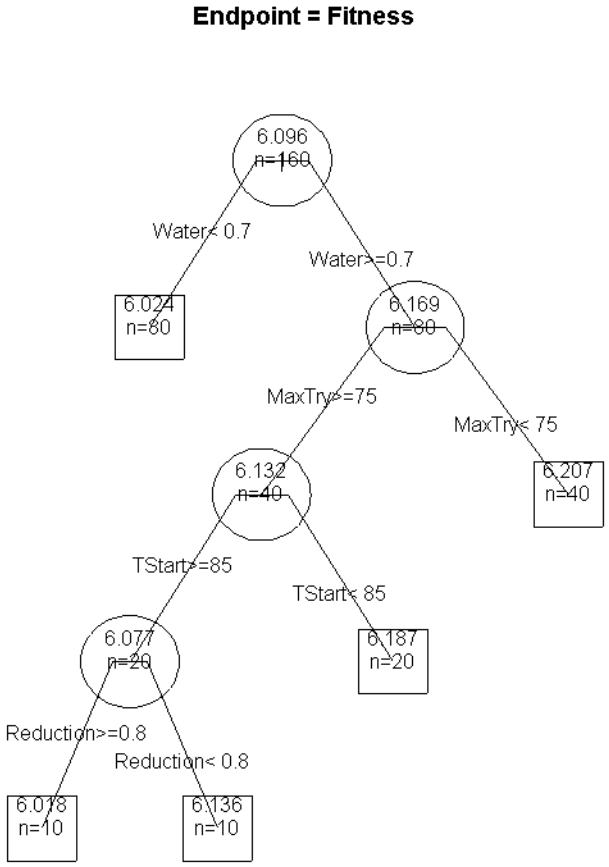

Abbildung 4.3: Beispiel eines Regressionsbaums.

Für weiterführende Betrachtungen zu dieser Thematik, speziell zum Einsatz bei Suchheuristiken, sei auf [BB04b] und [BB04a] verwiesen.

4.2 Fragestellungen

Verifikation

Nachdem so viel Aufwand und Wissen in das Projekt gesteckt worden war, sollte nun die Funktionsweise im Praxistest überprüft werden. Um die korrekte Funktionsweise der implementierten Heuristiken zu überprüfen, sollten diese mit bekannten Leistungsdaten aus der Literatur verglichen werden. Dazu wurden einige dort beschriebene Versuche wiederholt und die Ergebnisse verglichen. Da die Anzahl der implementierten Heuristiken recht groß ist, wurden auf Grund des knappen Zeitrahmens einige Heuristik-Plug-In-Kombinationen stellvertretend herausgesucht.

Exemplarisch sei im Folgenden die Verifikation des PSO beschrieben. Hierzu wurden als Vergleich die Angaben aus [LRK01] gewählt. Die Parametereinstellungen sind in Tabelle 4.1 zu finden. Für jede Problemkonfiguration wurden zur Ergebnisermittlung, wie in dem Vergleichspaper angegeben, 100 Läufe durchgeführt. Davon wurden der Mittelwert und der Standardfehler berechnet. Die Tests ergaben, dass die Güte der Ergebnisse auf dem Niveau der Vergleichsimplementierung aus der Literatur und damit der PSO durchaus konkurrenzfähig ist. Daraus war ersichtlich, dass die Implementierung ohne semantische Fehler ist und damit für weitere Untersuchungen verwendet werden konnte.

Parameter (Literatur)	Parameter (MooN)	Wert
Maximum velocity	Velocity bound	100
Population size	Swarm size	20
Dimension	Problemdimension	10, 20, 30
Inertia weight	Inertia weight	0.55
Minimal solution	Minimal solution	-100
Maximum solution	Maximum solution	100
Phi2	Social term	2.0
Phi1	Cognitive term	2.0
Generations	Generations	1000, 1500, 2000

Tabelle 4.1: Parameter für Verifikation von PSO.

Auch bei den anderen Versuchen zeigte sich, dass unsere Implementierungen meist nahe an den Vergleichswerten lagen; teilweise gab es jedoch auch größere Abweichungen. Dies war auch so zu erwarten, da es sich bei den von uns untersuchten Heuristiken um reine Standardimplementierungen

handelte, während die Vergleichswerte von bereits verbesserten Algorithmen kamen. Der Zweck, die Funktionalität unserer Implementierungen zu testen, wurde aber erfüllt. Die Versuchsergebnisse von PSO sind in den Tabellen 4.2 und 4.3 zu finden.

Dimension	Generationen	Literatur	MooN
10	1000	$2.98 \cdot 10^{-33} \pm 4.21 \cdot 10^{-33}$	$1.80 \cdot 10^{-33} \pm 7.64 \cdot 10^{-33}$
20	1500	$3.03 \cdot 10^{-20} \pm 9.27 \cdot 10^{-21}$	$3.32 \cdot 10^{-19} \pm 1.28 \cdot 10^{-18}$
30	2000	$6.29 \cdot 10^{-13} \pm 7.64 \cdot 10^{-14}$	$6.58 \cdot 10^{-13} \pm 8.85 \cdot 10^{-12}$

Tabelle 4.2: Versuchsergebnis: PSO auf Kugelfunktion.

Dimension	Generationen	Literatur	MooN
10	1000	43.049 ± 11.554	52.09 ± 86.99
20	1500	115.143 ± 19.871	78.98 ± 117.21
30	2000	154.519 ± 24.512	140.80 ± 176.26

Tabelle 4.3: Versuchsergebnis: PSO auf Rosenbrock-Funktion.

Finden von Parametereinstellungen mit DoE

Als nächstes wollten wir zu den uns zur Verfügung gestellten Problemen für je eine Heuristik eine gute Parametereinstellung finden. Wir wählten den Ansatz des *Design of Experiments* (DoE), der es erlaubt, mit einer vergleichsweise kleinen Anzahl von Experimenten eine gute Parameterkombination herauszufinden. Der von uns verfolgte Ansatz verläuft wie folgt: Zunächst wird ein so genanntes *faktorielles Design* aufgestellt. Dazu werden die Parameter als Faktoren aufgefasst, für die je zwei Level, also Parametereinstellungen, möglichst gut „geraten“ werden. Nun werden alle Levelkombinationen der Faktoren in einer Tabelle aufgestellt. Für k Faktoren ergeben sich so 2^k Experimente. Diese Experimente werden nun durchgeführt und die Ergebnisse analysiert.

Für die Analyse benutzten wir das frei verfügbare Statistik-Paket `R`. Mit dieser Software ist es auf einfache Art und Weise möglich, die Auswirkung der Parameter auf das Ergebnis sowie Wechselwirkungen zwischen den Pa-

rametern zu ermitteln. Die gewonnenen Erkenntnisse können graphisch veranschaulicht werden.

Sind die Ergebnisse der ersten Phase ansatzweise normalverteilt, so kann man mit ihnen nun nach der „Methode des steilsten Abstiegs“ neue Experimente planen. Dazu wird, sofern dies vertretbar ist, ein linearer Zusammenhang der Parameter angenommen. Unter dieser Annahme wird nun ein lineares Modell aufgestellt, mit dessen Hilfe der Pfad des steilsten Abstiegs bestimmt werden kann. Entlang dieses Pfads werden nun neue Parametereinstellungen für die nächste Versuchsreihe gewählt. Die Einstellung, die zu den kleinsten Fitnesswerten führt, dient als Ausgangspunkt für ein neues faktorielles Design. Dieses Verfahren kann nun iteriert werden, bis keine Verbesserung mehr möglich ist oder die Optimierung den Ansprüchen genügt.

Zusammenfassend kann man sagen, dass DoE unter gewissen Umständen schnell zu guten Parametereinstellungen führen kann. Es kann zwar nicht garantiert werden, dass tatsächlich eine gute Konfiguration gefunden wird, dennoch ist dieser Ansatz dem „rein zufälligen“ Ausprobieren von Parameterkombinationen vorzuziehen.

4.3 Versuche

4.3.1 Einleitung

Für die Versuche wurden vier Kombinationen aus Heuristiken und Problemen ausgewählt, um daran die Tauglichkeit von MooN für wissenschaftliche Untersuchungen aufzuzeigen. Im Folgenden sind die Versuchsanordnungen, die Auswertung und die Ergebnisse beschrieben.

Die ausgewählten Kombinationen waren:

- Particle Swarm Optimization (PSO) auf der Kugelfunktion (Sphere),
- Ant Colony Optimization (ACO) auf dem Travelling Salesperson Problem (TSP),
- Great Deluge (GD) auf dem Fahrstuhlproblem sowie
- Particle Swarm Optimization (PSO) auf dem Temperierbohrungsproblem (Mold Temperature Control).

4.3.2 Ant Colony Optimization (ACO) — Travelling Salesperson Problem (TSP)

Untersucht wurde das Verhalten bzw. die besten Parametereinstellungen für den ACO-Algorithmus auf einem speziellen TSP-Problem (`berlin52` mit 52 Städten).

Design of Experiments (DoE) — Schritt 1

Es wurde für die sechs zu variierenden Parameter ein *vollständiges faktorielles Design* mit zwei Faktor-Leveln ausgewählt. Für die Einstellung „high“ (+) bzw. „low“ (-) wurden folgende Werte gewählt:

	Parameter	-	+
A	Number of ants	5	20
B	Initial pheromon amount	0.1	0.2
C	Evaporation	0.1	0.3
D	Pheromon amount	10	100
E	alpha	1	5
F	beta	1	5

Bei der Inaugenscheinnahme der Ergebnisse dieses Designs stellte sich heraus, dass offenbar der Parameter `beta` einen sehr entscheidenden Einfluss hatte. Die high/low Einstellung dieses Parameters teilte die Ergebnismenge in zwei disjunkte Teile, wobei festzustellen war, dass der höhere Wert deutlich bessere Ergebnisse erzeugte (siehe dazu Abb. 4.1 oben). Die Abbildungen 4.4 und 4.5 zeigen die beiden Teile der Ergebnismenge. Daher wurde zunächst nach einem guten Einstellungsbereich für diesen Parameter gesucht.

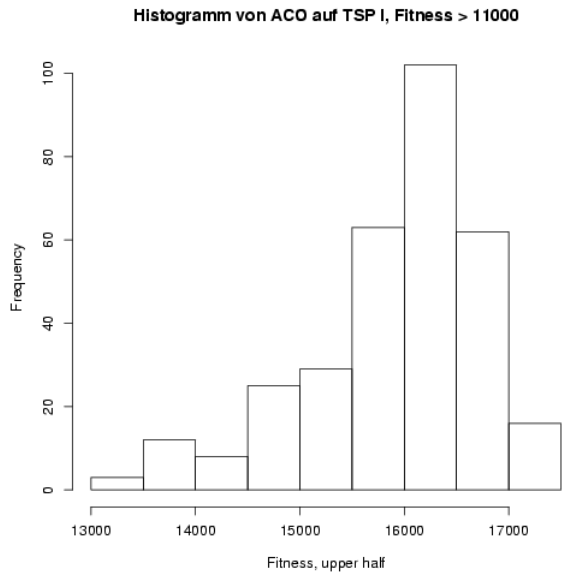

Abbildung 4.4: Teilhistogramm, zeigt den Teilbereich höherer Fitnesswerte.

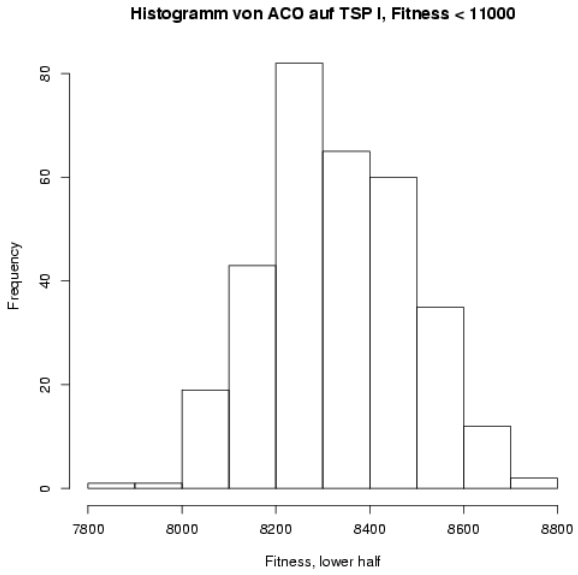

Abbildung 4.5: Teilhistogramm, zeigt den Teilbereich kleinerer Fitnesswerte.

Suche nach einem guten Bereich für `beta`

Da der Einfluss der anderen Faktoren gegenüber `beta` augenscheinlich vernachlässigbar klein ist, wurden diese auf einer beliebigen Einstellung aus dem obigen DoE festgesetzt. Folgende Durchläufe wurden mit jeweils fünf Iterationen durchgeführt:

A	**B**	**C**	**D**	**E**	**F**	**Y**
20	0.2	0.3	100.0	5.0	5.0	8.314.965
20	0.2	0.3	100.0	5.0	2.0	11.972.559
20	0.2	0.3	100.0	5.0	3.0	9.594.376
20	0.2	0.3	100.0	5.0	4.0	8.674.383
20	0.2	0.3	100.0	5.0	10.0	77.039.736
20	0.2	0.3	100.0	5.0	20.0	80.083.354
20	0.2	0.3	100.0	5.0	40.0	80.933.535
20	0.2	0.3	100.0	5.0	30.0	80.933.535
20	0.2	0.3	100.0	5.0	100.0	81.821.924

Die Ergebnisse wurden in einem Boxplot visualisiert, siehe Abb. 4.2 oben. Bei der Betrachtung haben wir festgestellt, dass ein Einstellbereich zwischen 10 und 20 für diesen Parameter für weitere Versuche angemessen erscheint. Daraus wurde ein neues faktorielles Design erstellt.

Design of Experiments (DoE) — Schritt 2: Neues Design

	Parameter	**-**	+
A	Number of ants	5	20
B	Initial pheromon amount	0.1	0.2
C	Evaporation	0.1	0.3
D	Pheromon amount	10	100
E	alpha	1	5
F	beta	10	20

Die Ergebnisse der Durchführung dieses neuen Designs wurden mit Hilfe der *Regressionsanalyse mit linearem Modell* und *Regressionsbäumen* untersucht. Dies sind zunächst die Ergebnisse der linearen Regressionsanalyse:

```
Residuals:
    Min      1Q  Median      3Q     Max
-415.32  -56.48   11.83   67.08  201.50

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 7912.21646   18.85586 419.616  < 2e-16 ***
F              3.16898    0.75208   4.214 2.88e-05 ***
E              6.94646    1.89711   3.662 0.000272 ***
C            211.20208   37.75883   5.593 3.31e-08 ***
B              0.76913   77.24019   0.010 0.992058
D             -0.03532    0.08396  -0.421 0.674151
A             -6.90226    0.49987 -13.808  < 2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

An den Koeffizienten kann man den Einfluss der Faktoren und die Richtung des Einflusses ablesen. Die letzte Spalte bzw. die Sternchen geben die Signifikanz des Ergebnisses für jeden Faktor an. Für die Faktoren mit geringer Signifikanz lassen sich keine oder nur sehr unsichere Aussagen machen. Daher wurden für die weitere Suche nach Parametereinstellungen die Faktoren mit hoher Relevanz und großem Koeffizienten ausgesucht.

Da die Annahme einer Normalverteilung der Ergebnisse nicht zu 100% gegeben war, wurde noch mit Hilfe der `rpart-Library` in `R` ein Regressionsbaum aufgestellt (4.6). Die Analyse durch Regressionsbäume basiert auf keinerlei Verteilungsannahmen und ist daher auch für nicht normalverteilte Daten einsetzbar.

An diesem konkreten Regressionsbaum lässt sich z. B. erkennen, dass der Parameter A (die Anzahl der Ameisen) den stärksten Einfluss auf den Ergebniswert hat und deswegen für die weitere Suche vorrangig betrachtet werden sollte. Außerdem enthält der Baum noch die Angabe, in welchem Bereich der Parameter A gewählt werden sollte, nämlich ≥ 12.5. Auch für die nächsten Parameter (von oben nach unten betrachtet) lassen sich Hinweise für verbesserte Einstellungen entnehmen. Allerdings nimmt die Signifikanz nach unten hin ab.

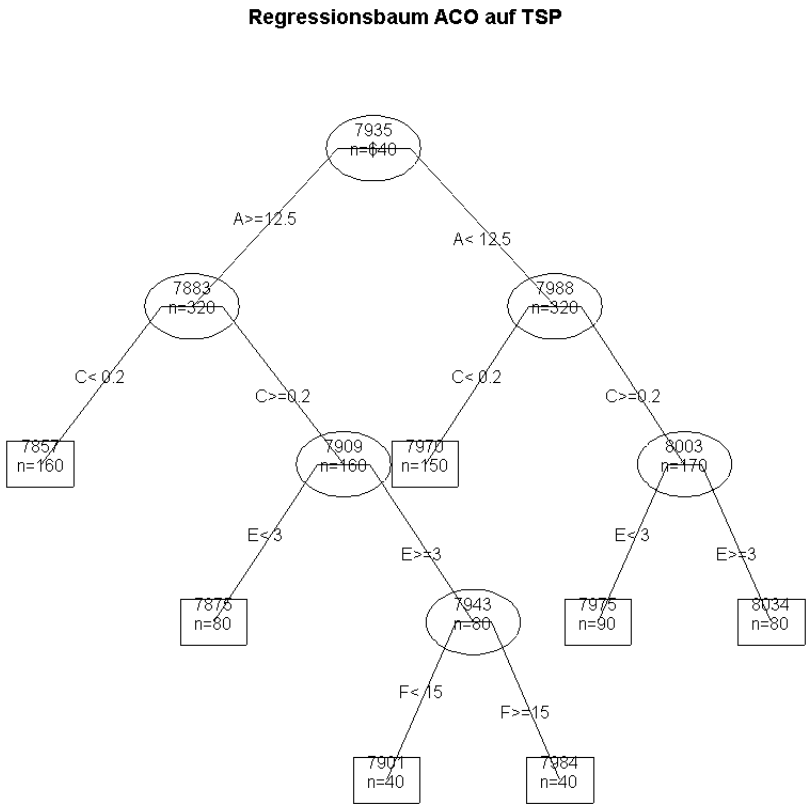

Abbildung 4.6: Regressionsbaum zu ACO.

4.3.3 Particle Swarm Optimization (PSO) — Mold Temperature Control

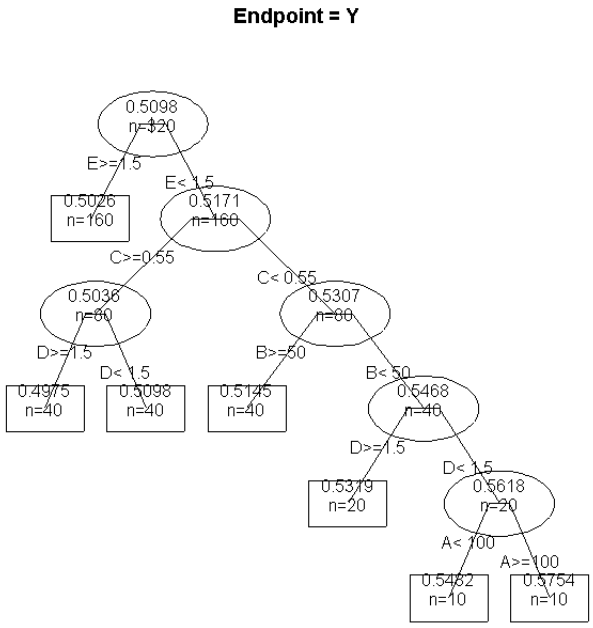

Abbildung 4.7: Regressionsbaum zu PSO.

Auch für diese Versuche wurde ein vollständiges faktorielles Design ausgewählt. Hierbei wurde für das Temperierbohrungsproblem die Anzahl der Stützpunkte für den Kühlpolygonzug auf 3 gesetzt.

	Parameter	-	+
A	Velocity bound	50	150
B	Population size	25	75
C	Inertia weight	0.45	0.65
D	Cognitive term (Phi 1)	1	2
E	Social term (Phi 2)	1	2

Leider lassen sich die hieraus resultierenden Daten nicht in ein lineares Modell „fitten“, da die Voraussetzung einer Normalverteilung der Daten nicht gegeben ist. Daher haben wir auch an dieser Stelle einen Regressionsbaum errechnet (siehe 4.7), da dieser auf keinerlei Verteilungsannahmen beruht. Wie bei der Analyse von ACO auf TSP schon beschrieben, lässt sich hieraus wieder der Einfluss der (obersten) Parameter ablesen.

4.3.4 Great Deluge (GD) — Fahrstuhlproblem

Ähnlich der Analyse des PSO auf dem Mold Temperature Control Problem ist es auch hier leider nicht möglich gewesen, aus den Ergebnisdaten des folgenden DoE-Entwurfs ein lineares Modell zu erstellen.

	Parameter	**-**	+
A	Start temperature	70	100
B	Reduction factor (temp)	0.75	0.85
C	Water factor (reduction)	0.5	0.9
D	Maximum try	50	100

Hier wählten wir für die Analyse ebenfalls den Ansatz der Regressionsbäume, siehe Abbildung 4.3 oben.

Kapitel 5

Seminare

Im folgenden Kapitel werden die von uns behandelten Seminarthemen vorgestellt. Es wird zunächst ein kurzer Überblick über die Seminare des ersten Semesters gegeben. Im Anschluss folgen die erarbeiteten Seminare des zweiten Semesters.

5.1 Seminare des ersten Semesters

Am Anfang unserer gemeinsamen Arbeit im ersten Semester gab es ein Kompaktseminar im Universitätskolleg Bommerholz. Dort wurden die beim ersten Treffen vergebenen (wissenschaftlichen) Seminare und (technischen) Tutorien vorgetragen. Mit diesen Vorträgen sollte ein einheitlicher Wissensstand in den verschiedenen Themengebieten erreicht werden. Dabei lag ein Schwerpunkt auf den unterschiedlichen Ansätzen für Heuristiken und ein weiterer auf drei ausgewählten Problemen. Außerdem wurden in einem Vortrag Testfunktionen für Heuristiken vorgestellt.

Gewonnen wurden nicht nur neue Erkenntnisse von heuristischen Methoden und interessanten Problemen, sondern auch Erfahrungen mit dem Vortragen von komplexen Themen vor einem kritischen Publikum.

Die folgende Übersicht enthält die Seminarvortragsthemen, die in Bommerholz behandelt wurden:

- Motivation
- Heuristiken
  - Evolutionsstrategien
  - Genetische Algorithmen
  - Simulated Annealing und Tabu Search
  - Memetische Algorithmen
  - Particle Swarm Optimization
  - Ant Colony Optimization
- Probleme
  - Testfunktionen
  - Fahrstuhlproblem
  - Routingproblem
  - Temperierbohrungsproblem
- OO-Programmentwurf von EA

Zu den einzelnen Seminaren sei auf [BBB+03] verwiesen.

5.2 Seminare des zweiten Semesters

Im Gegensatz zu den Seminaren des ersten Semesters, die den Beginn der PG markierten und Grundlage unserer Arbeit waren, hatten die Vorträge im zweiten Semester begleitenden und vertiefenden Charakter. Zielsetzung war einerseits, die bereits gewonnenen Ergebnisse zu erweitern, andererseits sollten anfallende Aufgaben wissenschaftlich vorbereitet werden.

Die Vorträge fanden in den regulären PG-Sitzungen statt. Mögliche Themen wurden in der vorlesungsfreien Zeit identifiziert und nach und nach gewählt bzw. zugewiesen. Als Vorgabe wurde eine Dauer von 15-20 Minuten angegeben. Im Gegensatz zur ersten Seminarphase war vor den Vorträgen

keine schriftliche Ausarbeitung anzufertigen; die nachfolgend stehenden Texte wurden für diesen Bericht erstellt.

Grundlage für die Identifizierung möglicher Themen war ein inhaltliches Konzept, das zunächst mögliche Themenbereiche und Ziele bzw. Fragestellungen der Vorträge klarstellte. Diesen Bereichen wurden erst anschließend konkrete Vortragsthemen zugeordnet. Die von uns behandelten Bereiche waren:

- **Implementationsaspekte der Heuristiken**
  Hier sollten die Heuristiken, die bereits vorgestellt und zur Implementierung vorgesehen waren, eingehender betrachtet werden. Es sollte v. a. die Art der Implementierung beleuchtet werden. Wichtige Fragestellungen waren etwa, wie kompliziert die Implementierung sein würde, welche Laufzeit die benötigten Operatoren haben, usw. Entsprechende Vorträge wurden zu ES und GA einerseits und zu ACO und PSO andererseits gehalten.

- **Vorstellung neuer Metaheuristiken**
  Zusätzlich zu den im ersten Semester vorgestellten wurden weitere Heuristiken eingeführt. Funktionsweise und Konzept dieser Heuristiken sollten ebenso beleuchtet werden wie Ergebnisse, die mit ihnen erzielt wurden. Außerdem sollte erörtert werden, ob die entsprechende Heuristik von uns überhaupt implementiert werden sollte. Auf diesem Themenbereich lag der Schwerpunkt der Vortragsreihe. Vorgestellt wurden „Ruin and Recreate“, „Sintflutalgorithmus“, „Räuber-Beute-System“, „Population Based Incremental Learning“, „Variable Neighbourhood Search“ und „Scatter Search“. Nur der Sintflutalgorithmus wurde von uns später tatsächlich implementiert.

- **Neue Probleme**
  Analog zu neuen Metaheuristiken sollten auch neue Anwendungsprobleme vorgestellt und diskutiert werden, ob sie sich zur Integration in MooN anböten. Exemplarisch wurde ein Problem aus der Graphentheorie behandelt.

- **Allgemeine Theorie**
  Um die Arbeit der PG besser in einen größeren wissenschaftlichen Rahmen einzuordnen, sollten relevante Forschungsergebnisse der Informatik vermittelt werden. Aus eher theoretischer Sicht wurde das NFL-Theorem vorgestellt. Ein mehr praktisch orientierter Vortrag beschäftigte sich mit Softwareüberprüfungsmethoden.

- **Konkurrenz zu MooN**
  Hier sollte MooN ähnliche oder gar in direkter Konkurrenz stehende Software vorgestellt werden. Thema waren Funktionsumfang und Einsatzgebiete der Produkte im Allgemeinen sowie Unterschiede zu und Gemeinsamkeiten mit MooN im Besonderen. Ergebnis hieraus war ein Vortrag zu „HotFrame“.

Die folgenden Abschnitte enthalten Zusammenfassungen der Vorträge. Sie stehen in der Reihenfolge, in der sie während des Semesters gehalten wurden.

5.2.1 Ruin And Recreate

Stefan Walter

Einführung und Strategie

Ruin & Recreate (R&R) ist eine Optimierungsstrategie für Graphenprobleme, wie das Travelling Salesperson Problem oder andere Routing Probleme. Da durch die zufallsgesteuerte Konstruktion von Lösungen mit großer Wahrscheinlichkeit schlechte Lösungen entstehen, ist die Idee von R&R das teilweise Zerstören von Lösungen und ein geschickter Wiederaufbau, vergleiche [Pöp00]. Die Zerstörung geschieht ähnlich dem Abwurf von Bomben: In einem kreisförmigen Zielgebiet werden alle Städte bzw. Knoten zerstört.

Wiederaufbau

Die Wiederaufbaustrategie ist ein zentraler Teil von R&R und wird als `best insertion` bezeichnet. Dabei werden die Städte als Fixpunkte und deren Verbindungen durch die aktuelle Route als „Gummibänder“ betrachtet. Städte, die durch den Abwurf einer Bombe zerstört worden sind, werden so wieder in die Route eingefügt, dass die Gummibänder möglichst wenig gedehnt werden müssen, d. h. die Route am wenigsten verlängert wird. Man kann diesen Umstand auch physikalisch auffassen: Die Spannung der Gummibänder stellt ein Energieniveau dar, das es zu minimieren gilt.

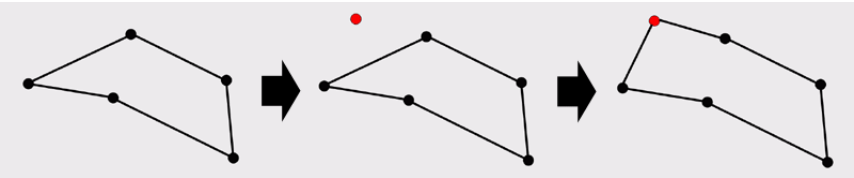

Abbildung 5.1: Einfügen eines Punktes.

Erweiterungen und Erkenntnisse

Als Erweiterungen des Konzeptes ist der Einsatz von „Streubomben“ möglich. Diese haben keinen radialen Zerstörungsradius, sondern löschen zufällig Kno-

ten aus dem Netzwerk. Bei Versuchen hat sich herausgestellt, dass ein Zerstörungsgrad von bis 30% bei Radial-Bomben, bzw. bis zu 50% bei Streubomben die beste Wirkung erzielt. Der Nachteil eines großen Zerstörungsgrades ist der damit verbundene hohe Rechenaufwand für den Wiederaufbau.

Bei Problemen mit hoher Komplexität, wie z. B. dem Vehicle Routing Problem, hat sich die R&R-Methode am besten bewährt. Der Vorteil liegt hier darin, dass beim Wiederaufbau qua Konstruktion nur zulässige Lösungen generiert werden. Zum Beispiel führen Gewichtsschranken bei anderen Verfahren mit kleinen Änderungen oft zu unzulässigen Lösungen. Das ist bei best insertion ausgeschlossen.

Outsourcing / Parallelisierung

Das Verfahren hat gewisse Ähnlichkeiten mit dem „Outsourcing“. Zur Erklärung: Outsourcing ist ein Begriff aus der Betriebswirtschaftslehre. Es handelt sich um eine Strategie bei der Dienstleistungen oder Produktionsleistungen, die ehemals im Unternehmen selbst erzeugt wurden, ausgelagert bzw. von Fremdfirmen eingekauft werden. Der Vorteil liegt in der besseren Berechenbarkeit der Kosten (da klare Angebote ausgehandelt werden) und in der Ausnutzung von Konkurrenzsituationen auf dem freien Markt. Betrachtet man das Beispiel des Vehicle Routing Problems sähe die Übertragung des Verfahrens folgendermaßen aus: Jeder Lastwagen wird als selbständige Geschäftseinheit betrachtet. Für jeden Auftrag, d. h. ein Paket vom Depot zu einem Ort zu bringen, gibt jeder Lastwagen ein Gebot in Höhe des Selbstkostenpreises ab. Da sich anfangs alle Lastwagen im Depot befinden, sind alle Gebote gleich und es wird zufällig ausgewählt. Im Verlauf ergeben sich jedoch für einige Lastwagen Vorteile, wenn ein Auftrag in die Nähe einer Stadt geht, die von diesem Lastwagen sowieso schon angefahren wird. Dann sind die Kosten und damit das Gebot des LKW nur die Kosten des Umwegs zur neuen Zielstadt. Würde jedoch die Gewichtsschranke bei einem Wagen überschritten, so gibt er erst gar kein Gebot ab.

Werden die Aufträge so verteilt, kann es zu ungünstigen Situationen kommen, so dass etwa eine Stadt von mehreren LKW angefahren wird. Dann hilft der „Abwurf einer Bombe“ auf diese Stadt. Sie wird gelöscht und damit auch alle Touren der dorthin fahrenden Lastwagen. Sofort danach wird sie wieder eingefügt und die entsprechenden Aufträge nach dem oben beschriebenen Verfahren vergeben. Dabei erhält mit hoher Wahrscheinlichkeit ein LKW den Zuschlag, der schon eine Stadt in der Nähe anfährt. Damit entzerrt sich das Routengeflecht also.

Diese unabhängige Betrachtung der Lastwagen begünstigt eine Parallelisierung der rechenaufwändigen Routenkonstruktion, Preisberechnungen etc. Dieser Teil des Verfahrens nimmt gewöhnlich bis zu 90% der Zeit in Anspruch.

Effekte der Bomben

Die Bomben erzeugen hauptsächlich aus großen Problemen mittelgroße. Das ist einerseits günstig für die Wiederaufbauverfahren, andererseits werden so Teilprobleme konstruiert, die eventuell exakt lösbar sind. Sie sind also dazu geeignet, Verfahren, die gut auf mittelgroßen Problemen arbeiten, auf große anwendbar zu machen.

Fazit

Der Literatur [Pöp00] ist zu entnehmen, dass in dem beschränkten Anwendungsbereich der Graphenprobleme mit dem R&R-Verfahren gute Ergebnisse erzielt wurden. Bei der Anwendung auf Lastwagenprobleme beispielsweise wurden mit geeignet angepassten R&R-Verfahren bereits im Durchschnitt Lösungen einer Qualität gefunden, die bei anderen Verfahren nur im günstigsten Fall erreicht wurden.

Andererseits gibt es für die Größe der Bomben nur sehr wenige Erfahrungswerte. Deswegen müssen günstige Parameter durch Probieren gefunden werden. Außerdem hat auch dieses Verfahren einen Nachteil mit fast allen anderen Heuristiken gemein: Die Qualität der Lösung ist nicht beweisbar.

5.2.2 Implementationsaspekte von ES und GA

Bianca Selzam

Im Rahmen der Seminarvorträge wurden die Implementierungsdetails der Genetischen Algorithmen und Evolutionsstrategien vorgestellt. Um den Aufbau eines evolutionären Algorithmus allen PG-Mitgliedern wieder ins Gedächtnis zu rufen, wurde zunächst der prinzipielle Aufbau eines solchen wiederholt:

Prinzipieller Aufbau eines EA

Nach der Initialisierung einer Startpopulation werden die Individuen anhand einer Fitnessfunktion bewertet. Eine bestimmte Menge von Eltern wird aus der Population genommen und gekreuzt, um die Chromosomen des Kinderindividuums zu erhalten. Anschließend können durch die Mutation einzelne Chromosomen verändert werden. Nachdem einige Eltern durch ihre besseren Kinder ersetzt wurden, werden die Abbruchbedingungen geprüft, ob die Iteration beendet wird.

Implementierung einer ES in MooN

Aufgrund der modularen Struktur von MooN wurden sowohl die ES als auch der GA in mehrere Klassen aufgeteilt. Für die ES wurden die Klassen `IndividualStandardES`, `PopulationStandardES`, `StandardES` und `StrategyStandardES` implementiert, deren Funktionsweisen im Folgenden erläutert werden sollen.

`IndividualStandardES` repräsentiert ein Individuum einer ES, dessen tatsächliche Geninformationen in einer `ArrayList` gespeichert werden. Die Objektparameter wie Schrittweite und Rotationswinkel werden in der Klasse `StrategyStandardES` gespeichert, ebenfalls in je einer `ArrayList`.

Eine Population dagegen wird durch die Klasse `PopulationStandardES` simuliert. Ihre Individuen werden wieder in einer `ArrayList` gespeichert.

Die tatsächliche Heuristik wurde in der Klasse `StandardES` implementiert. Zu Beginn jeder Heuristik muss die Methode `initialize()` aufgerufen werden, um notwendige Parameter zu errechnen sowie die Startpopulation zu erzeugen. Solange keine Abbruchbedingung erfüllt ist, wird die Methode `nextGeneration()` durchgeführt, welche pro Aufruf eine Generation voll-

zieht. Weiterhin verfügt `StandardES` über get- und set-Methoden, die hier jedoch nicht näher beschrieben werden sollen.

Implementierung eines GA in MooN

Da ein GA keine Strategieparameter verwendet, war hierfür keine eigene Klasse notwendig. Lediglich die Klassen `IndividualGA`, `PopulationGA` und `GeneticAlgorithm` wurden von uns realisiert. Der Aufbau dieser Klassen ist analog zu denen der ES, da sie dieselben Interfaces `Population` und `Individual` implementieren.

Vorstellung von eaLib

eaLib [Rum03] ist eine Java-Klassenbibliothek für die Implementierung evolutionärer Algorithmen. Sie wurde von Andreas Rummler an der TU Ilmenau entwickelt, um anderen Programmierern von Heuristiken eine reichhaltige Auswahl an genetischen Operatoren, Abbruchbedingungen und Hilfsklassen zu bieten. Da ihre Entwicklung von der Deutschen Forschungsgemeinschaft gefördert wurde, hatte sich Herr Rummler für die kostenlose Bereitstellung an die Öffentlichkeit entschieden. Allerdings darf sie nur in nichtkommerziellen Produkten eingesetzt werden.

Überlegungen zur Einbindung in MooN

Nach der Vorstellung der eaLib in der Gruppe entschieden wir uns dafür, die durch eaLib bereitgestellten Operatoren lediglich als Inspiration für unsere eigenen Implementierungen zu nutzen. Da wir das erste Semester der Projektgruppe mit der sorgfältigen Modellierung von MooN verbracht hatten, hätte die Integration der eaLib unser Konzept komplett durcheinander gebracht. Weiterhin bietet die eaLib nur einen Teil der Funktionalitäten, die wir in MooN verwirklicht haben, weshalb wir uns letztendlich gegen die Integration der eaLib in MooN entschieden.

5.2.3 Sintflutalgorithmus

Thomas Tometzki

Der Sintflutalgorithmus (siehe [DSW93]) wurde 1993 von Gunter Dueck und Tobias Scheuer bei IBM in Heidelberg entwickelt. Der Sintflutalgorithmus ist eine Weiterentwicklung und Vereinfachung des Simulated Annealing-Verfahrens.

Die Vorgehensweise ist denkbar einfach: Ein „Wanderer“ wird an einer beliebigen Stelle einer Funktion „ausgesetzt“ und soll einen möglichst hohen Gipfel finden. Der höchste Gipfel der Funktion steht hier also für das zu suchende Optimum. Der Wanderer ist ständig in Bewegung. An einer Position angekommen, wählt er eine zufällige Richtung und macht einen weiteren Schritt. Während er ziellos umherirrt, fängt es an zu regnen — die Sintflut hat begonnen. Während der Wasserstand unaufhörlich steigt, muss der Wanderer lediglich darauf achten, dass er keine nassen Füße bekommt. Hat er eine Richtung gewählt, bei der er beim nächsten Schritt ins Wasser treten würde, muss er eine andere Richtung wählen. Er merkt sich ständig den höchsten Punkt, den er schon erreicht hat. Ist er nur noch von Wasser umgeben und kann keinen Schritt mehr machen, bei dem er trocken bleibt, endet das Verfahren. Der Wanderer hat einen evtl. ausreichend hohen Gipfel gefunden (vgl. [Ull02]).

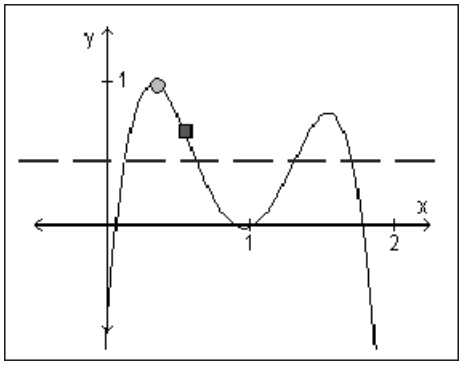

Abbildung 5.2: Funktionsweise des Sintflutalgorithmus. Aus [Mal00].

Abbildung 5.2 zeigt einen solchen Durchlauf des Verfahrens mit einer Funktion in einem zweidimensionalen Suchraum. Das Quadrat symbolisiert den

Wanderer, der Kreis den bislang höchsten Ort, den er erreicht hat. Die gestrichelte Linie deutet an, wie hoch das Wasser schon gestiegen ist.

Diese umgangssprachliche Beschreibung des Verfahrens lässt sich auch formal in einem Flussdiagramm darstellen (siehe Abb. 5.3)

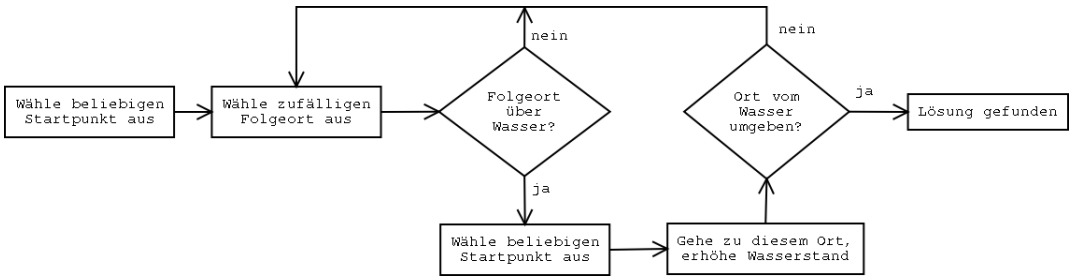

Abbildung 5.3: Flussdiagramm zum Sintflutalgorithmus. Nach [Mal00].

Durch die Einfachheit des Algorithmus und die wenigen Parameter, die gehandhabt werden müssen, lässt sich der Sintflutalgorithmus leicht in einer höheren Programmiersprache umsetzten.

Die Stärken des Algorithmus liegen auf der Hand: Im Gegensatz zu heuristischen Verfahren basiert der Sintflutalgorithmus auf einer „trial and error"-Methode, die eine optimale Lösung in einer während des Verfahrens zufällig entstehenden, abgegrenzten Umgebung ermittelt.

Die einzigen Parameter, die gewählt oder geändert werden müssen sind

- der Startwasserstand,
- die Regenmenge pro Schritt und
- die Schrittweite des Wanderers.

Es werden nur elementare Rechnungen benutzt.

Allerdings hat der Sintflutalgorithmus auch Nachteile. Man benötigt mehr Schritte, im Vergleich zum Simulated Annealing etwa doppelt so viele, um zur Lösung zu gelangen. Die Lösung ist stark abhängig von der Schrittweite des Wanderers. Ist die Schrittweite zu klein gewählt, erreicht er durch die Zufälligkeit seiner Schritte nur einen kleinen Teil der Funktion, sofern das Wasser schnell genug steigt. Wird die Schrittweite zu groß gewählt, kann es passieren, dass er mit einem einzigen Schritt über einen Gipfel hinwegsteigt.

5.2.4 Softwareüberprüfungsmethoden

Selcuk Balci

Die folgende Darstellung der Thematik folgt [Rie97]. Allgemeines Ziel des Testens von Programmen ist es, die Qualität von Softwaresystemen durch systematisches Ausführen der Software zu steigern. Wir teilen Testmethoden in zwei große Gruppen ein, statische und dynamische. Sie unterscheiden sich durch ihre Vorgehensweise. Beim statischen Testen überprüft man die Funktionalität in der Reihenfolge, in der sie entwickelt wurde, beim dynamischen Testen hingegen in der Reihenfolge die bei Ausführung des Programms auftritt.

In beiden Fällen kann informell oder formal getestet werden. Ein Test ist informell, wenn er unter menschlicher Begutachtung durchgeführt wird. Das formale Testen läuft unter dem Einsatz mathematischer und logischer Verfahren ab.

Statische formale Verfahren

Hier unterscheidet man:

- **Formale Verifikation:**
  Die Korrektheit des Programms wird mit Hilfe einer formalen Spezifikation gezeigt. Zuerst wird bewiesen, dass das Programm richtige Ergebnisse liefert, sofern es stoppt. Für einen totalen Korrektheitsbeweis ist anschließend noch zu zeigen, dass das Programm nicht in eine Endlosschleife gerät.

- **Syntaxanalyse:**
  Diese wird heutzutage vom Compiler übernommen.

- **Datenflussanalyse:**
  Diese beinhaltet die Überprüfung des Datenflusses eines Programms oder dessen Spezifikation.

- **Kontrollflussanalyse:**
  Es wird der Kontrollfluss eines Programms oder dessen Spezifikation überprüft. Hierbei werden unendliche Schleifen oder nichterreichbare Codestücke aufgedeckt.

Statische informelle Verfahren

Hier ist vor allem die Inspektion zu nennen. Eine Inspektion wird von höchstens vier Personen durchgeführt. Eine Sitzung sollte dabei nicht länger als zwei Stunden dauern, wobei zwei Sitzungen pro Tag akzeptabel sind. Bei nicht formalen Dokumenten ist eine Inspektion notwendig, bei formalen Dokumenten kann sie sinnvoll sein.

Bei einer Inspektion gibt es vier Rollen: den Moderator, den Entwerfer, den Codierer und den Tester. Eine Inspektion umfasst sieben Schritte:

1. Überblick,
2. Vorbereitung,
3. Eigentliche Inspektion,
4. Überarbeitung/Korrektur,
5. Verfolgung/Überprüfung,
6. Auswertung/Nachbereitung und
7. Inspektion als Einschub beim dynamischen Testen.

Dynamische formale Verfahren

Testen im eigentlichen Sinne bezeichnet eine dynamische Prüfung der Software durch einen experimentellen Ablauf in der realen Umgebung. Man unterscheidet:

- **Idealer Test:**
  Ein Test, der genau dann ein Fehlverhalten aufzeigt, wenn die Software tatsächlich einen Fehler enthält. Ein solcher Test existiert zwar, kann aber nur durch Zufall gefunden werden.

- **Erschöpfender Test:**
  Ein theoretisch möglicher, erschöpfender Test aller Eingabekombinationen ist praktisch meist undurchführbar. Ein Test bei drei Eingaben zu je 16 Bit und einer Testgeschwindigkeit von 0,0001 Sekunde pro Testdatum würde beispielsweise 900 Jahre dauern.

- **Stichproben-Test:**
  „Stichprobe“ ist historisch gesehen ein Begriff aus dem Hüttenwesen und bezeichnet einen Abstich aus dem flüssigen Eisenerz im Hochofen. Dies reicht dort als Qualitätsüberprüfung, da die Materialeigenschaften im gesamten Hochofen nur gering variieren.

Die Teststrategien und Testansätze werden nach den Kriterien gruppiert, nach denen sie ihre Testwerte aussuchen, sie können spezifikationsorientiert oder implementationsorientiert sein.

Im Gegensatz zum Testen im eigentlichen Sinne wird bei der „symbolischen Ausführung“ mit symbolischen Werten getestet, die als Eingabevariablen verwendet werden.

Dynamische informelle Verfahren

Im Unterschied zur eigentlichen Inspektion, gibt bei einem Walkthrough ein Gruppenmitglied einfache Testdaten vor und leitet die restlichen Gruppenmitglieder an, das Programmstück mit den Daten zu testen. Hierbei wird gegenüber dem vollständigen statischen informellen Verfahren einerseits Aufwand gespart, andererseits können nicht alle Fehler aufgedeckt werden, da nicht alle Programmzweige ausgeführt werden.

5.2.5 Das NFL-Theorem

Sören Blom

Im Rahmen der Seminare des zweiten Semesters wurde das „(Almost) No Free Lunch Theorem“ ((A)NFL-Theorem) vorgestellt.

Das NFL-Theorem

Der Auspruch „There ain't no such thing as a free lunch“ bedeutet sinngemäß so viel, wie „Im Leben gibt es nichts umsonst“. Bezogen auf die Informatik stellt sich die Frage, ob es irgendeine Klasse von (Optimierungs-)algorithmen gibt, die auf allen Instanzen einer Problemklasse besser arbeitet, als andere. Dies bezieht sich etwa auf Laufzeit oder Resourcenverbrauch. Dies wurde früher z. B. für Evolutionäre Algorithmen behauptet.

Diese Annahme wurde aber in [WM95] und [WM97] widerlegt. Der Kern dieser Arbeit stellt das NFL-Theorem dar, das sich vereinfacht wie folgt darstellen lässt (nach [DJW99]):

Gegeben sei eine Menge $F = f : X \to Y$, wobei X und Y endliche Mengen sind und Y vollständig geordnet ist. Für jede zufällig ausgewählte Funktion aus F haben alle Optimierverfahren dasselbe durchschnittliche Verhalten.

Demnach ist es zwar möglich, einen Algorithmus zu finden, der auf bestimmten Problemklassen gute Resultate liefert, jedoch wird er diesen Vorteil in anderen Bereich wieder verlieren. Vergleicht man zwei Heuristiken miteinander über allen Problemen, so wird man feststellen, dass sie durchschnittlich gleich gut arbeiten.

A free appetizer

Demnach könnte man der Auffassung sein, dass es wenig Sinn hat, Energie in die Erforschung bestimmter Heuristiken zu investieren, wenn sie doch, bezogen auf alle Probleme, keinen Vorteil bieten. Dieser Ansicht widersprechen nicht nur die Erfahrungen in der Praxis, in der sich Evolutionäre Algorithmen für eine Vielzahl praktischer Probleme bewährt haben. Droste, Jansen und Wegener argumentieren in [DJW02] und [DJW99], dass das NFL-Theorem zwar mathematisch korrekt ist, aber niemals „alle Funktionen“ berücksichtigt werden bzw. werden können. Dies kann entweder daran liegen, dass die Menge möglicher Funktionen zu groß ist, oder dass deren Auswertung nicht effizient möglich ist.

Dieses „NFL-Szenario“ genannte, „alle Funktionen“ einschließende, Szenario ist in den Augen der Autoren unrealistisch. Sie zeigen daher einige realistischere Szenarien auf, bei denen nur eine Teilmenge aller Funktionen betrachtet wird.

- **Zeitbeschränkt**
  Es werden nur solche Funktionen betrachtet, die sich in einer beschränkten Zeit auswerten lassen.
- **Größenbeschränkt**
  Es werden nur solche Funktionen betrachtet, die sich durch eine begrenzte Anzahl von Schaltkreisen darstellen lässt.
- **Kolmogoroff-beschränkt**
  Es werden nur Funktionen betrachtet, deren Kolmogoroff-Komplexität beschränkt ist.

Innerhalb solcher Szenarien lassen sich einige Aussagen über die Leistungsfähigkeit verschiedener Verfahren treffen, bei denen Unterschiede sichtbar werden können. Allerdings fügen die Autoren hinzu, dass der erwartete Nutzen gering sein wird, da Aussagen über Funktionsklassen dieser Art sehr abstrakt sein werden.

In [DJW99] (S. 12f) wird die Reichweite des „free appetizer“ in der Zusammenfassung bewertet: „The NFL theorem is a simple theorem ruling out statements that some search heuristics have some advantage on the average of ‚all‘ functions. However the NFL scenario is not a realistic one. For realistic black box scenarios, in particular those defined by some restrictions on the complexity of the considered functions, NFL theorems will not hold, but at least a free appetizer is possible in some situations. *The ANFL theorem proves, that one cannot expect much by well-chosen heuristics in complexity restricted black box scenarios.*“ (Hervorhebung durch mich).

Konsequenzen für die PG

Für die Projektgruppe ergeben sich aus der Kenntnis dieser Diskussion keinerlei direkte Konsequenzen, da sie theoretischer Art sind und der Fokus des Projektes auf anwendungsnahe Fragestellungen gerichtet ist. Andererseits sollte die Debatte um das NFL-Theorem jedem als Hintergrundwissen bekannt sein, der sich mit Optimierungsverfahren beschäftigt.

5.2.6 Hotframe

Ulf Schneider

Einführung

Hotframe ist ein Framework zur Implementierung diverser Meta-Heuristiken und Nutzung dieser in einem zu schreibenden Decision Support System (Entscheidungsfindungs-System, kurz DSS). Es entstand im Rahmen der Dissertation [Fin00] von Andreas Fink am Institut für Wirtschaftsinformatik der Universität Hamburg unter dem Titel „Software-Wiederverwendung bei der Lösung von Planungsproblemen mittels Meta-Heuristiken“. Es geht in dieser Dissertation um die Realisierung von wieder verwendbaren Metaheuristiken für den Einsatz in Entscheidungsfindungs-Systemen. Im Rahmen von DSS ist der Einsatz von Metaheuristiken momentan nicht üblich, dies hofft Fink allerdings durch seinen Ansatz zu ändern.

Das Framework wird nicht komplett, sondern lediglich das Grundkonzept vorgestellt und mit MooN verglichen. Die Teilnehmer der Projektgruppe sollten nachfolgend überlegen, ob und inwiefern sie Teile dieses Konzeptes in MooN übernehmen wollten.

Grundgerüst

HotFrame selbst besteht im Kern aus

- C++ - Templates,
- beispielhaft realisierten Metaheuristiken:
  General Simulated Annealing, Tabu-Search und Iterated Local Search;
  ein EA wird hingegen lediglich vorgestellt,
- Lösungsraumspezifischen Datenklassen,
- Observer-Klassen (Logger) und einem Interface für Abbruchbedingungen, auf die hier nicht weiter eingegangen wird.

Unterschiede zu MooN

MooN kann Probleme und Heuristiken beliebig kombinieren und diese Zusammenstellung jederzeit ändern. Bei HotFrame wird zu Beginn ein Problem

bestimmt und bleibt im Weiteren fest. Die Heuristiken werden an die Datentypen des Problems angeglichen, können aber danach beliebig ausgetauscht werden. Das Problem ist Teil des Suchraumes und wird dort lediglich als Funktion realisiert. Der Suchraum ist zentraler Anschlusspunkt für alle Teile des Frameworks.

Der entscheidende Unterschied ist, dass HotFrame nur Teil eines zu entwickelnden übergeordneten Programms und nicht wie MooN ein eigenständiges Programm ist.

Lösungsraum und Nachbarschaft

Das zentrale Konstrukt von HotFrame ist der Suchraum. Er ist nicht konkret implementiert, sondern ein *Interface*, in dem Methoden zur Durchsuchung des Lösungsraums vorgegeben sind. Der eigentliche Suchraum ist problemspezifisch vom Entwickler des Problems zu realisieren.

Es gibt eine Lösung `S`, die Ausgangspunkt für eine Nachbarschaftssuche ist. Diese kann auf Gültigkeit überprüft werden (`evaluate()`), gemäß des Problems ausgewertet (`f()`) oder verschoben (`move()`) werden. Zudem kann getestet werden, wie gut ein Verschieben der Lösung an einen bestimmten Ort wäre (`computeEvaluation`). Das Interface `S` ist in 5.4 abgebildet.

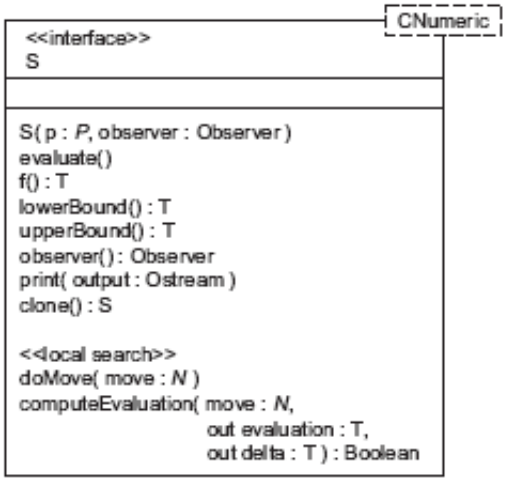

Abbildung 5.4: *Interface* `S`. Aus [Fin00].

Es gibt eine Nachbarschaft `NeighborhoodPosition` zum Suchpunkt `S`, in der mittels der Klasse `N` ein anderer Lösungspunkt in der Nachbarschaft gesucht werden kann (siehe 5.5). Es gibt Methoden in `N`, um einen nächsten Nachbarn zu suchen (`operator++()`) und die Güte dieses Nachfolgers zu bestimmen (`operator*()`).

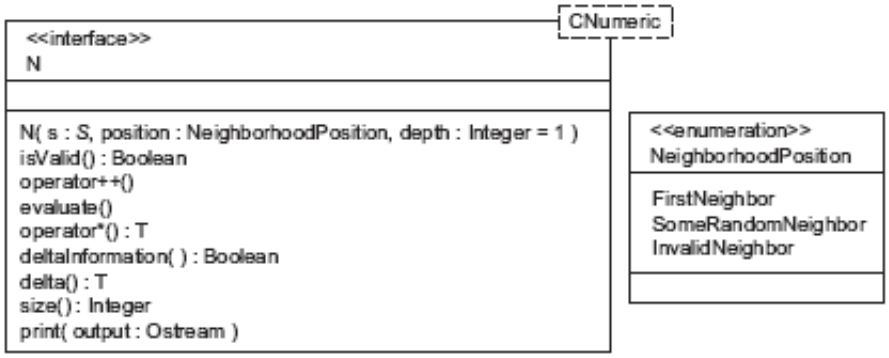

Abbildung 5.5: *Interface* `N` und *Enumeration* `NeighbourhoodPosition`. Aus [Fin00].

Gesteuert wird dieser Suchvorgang von einer Heuristik, die im nächsten Abschnitt näher vorgestellt wird. Prinzipiell wird ein beliebiger, gültiger Suchpunkt `S` erzeugt und die Nachbarschaft in `NeighbourhoodPosition` gespeichert. Dann wird mittels `N` diese `NeighbourhoodPosition` nach einem neuen guten Nachbarn abgesucht, der Punkt `S` versetzt und die `NeighbourhoodPosition` und `N` aktualisiert.

Modulare Heuristiken

Eine hervorragende Idee in der Dissertation ist die Modularisierung der Heuristiken. Fink hat ausgehend von einer iterierten lokalen Suche eine Möglichkeit gefunden, ähnliche Heuristiken auf einen Grundalgorithmus abzubilden, sodass leicht eine Abänderung des Funktionsverhaltens der Heuristik zu realisieren ist. Der Grundalgorithmus ist in 5.6 dargestellt.

Beispiele für Realisierungen von anderen Heuristiken mittels dieses Grundalgorithmus sind in 5.7 angeführt.

Abschließende Bemerkung

Grundlage dieses Seminarbeitrages ist die Dissertation von Andreas Fink [Fin00]. [FV02] ist eine englische Übersetzung eines Teiles der Dissertation. Sie enthält alle hier vorgestellten Informationen.

Algorithmus 2 IteratedLocalSearch.

IteratedLocalSearch
$< S, N, NeighborSelection, Diversification >$
$(s, T_{\max} = \infty, I_{\max} = \infty, R_{\max} = 1, \omega = \text{false}, returnBest = \text{true})$:
$\Omega : (t \geq T_{\max})$ or (ω)

$s_{\text{best}} = s;$
for $r = 1$ to $R_{\max}$
  if $r > 1$
    $Diversification(s);$
  $i = 0;$
  do
    $i = i + 1;$
    $s' = NeighborSelection< S, N >(s);$
    if s' is valid
      $s = s';$
      if $f(s) < f(s_{\text{best}})$
        $s_{\text{best}} = s;$
  while (s' is valid) and ($i < I_{\max}$);
if *returnBest*
  $s = s_{\text{best}};$

Abbildung 5.6: Grundalgorithmus „Iterated Local Search“. Aus: [Fin00].

Algorithmus 7 SteepestDescent.

SteepestDescent $< S, N > (s, T_{\max}, I_{\max}, \omega)$:

IteratedLocalSearch $< S, N,$ BestPositiveNeighbor, $\phi >$
$(s, T_{\max}, I_{\max}, 1, \omega,$ true$)$;

Algorithmus 8 FirstDescent.

FirstDescent $< S, N > (s, T_{\max}, I_{\max}, \omega)$:

IteratedLocalSearch $< S, N,$ FirstPositiveNeighbor, $\phi >$
$(s, T_{\max}, I_{\max}, 1, \omega,$ true$)$;

Algorithmus 9 IteratedSteepestDescent.

IteratedSteepestDescent $< S, N, \mathit{Diversification} > (s, T_{\max}, I_{\max}, R_{\max}, \omega)$:

IteratedLocalSearch $< S, N,$ BestPositiveNeighbor, $\mathit{Diversification} >$
$(s, T_{\max}, I_{\max}, R_{\max}, \omega,$ true$)$;

Algorithmus 10 RandomWalk.

RandomWalk $< S, N > (s, T_{\max}, I_{\max}, \omega, \mathit{returnBest})$:

IteratedLocalSearch $< S, N,$ RandomNeighbor, $\phi >$
$(s, T_{\max}, I_{\max}, 1, \omega, \mathit{returnBest})$;

Abbildung 5.7: Abgeleitete Algorithmen. Aus: [Fin00].

5.2.7 Ramseyzahlen

Dirk Hoppe

Überblick

Die Motivation zur Beschäftigung mit dem relativ außergewöhnlichen Bereich der Führung konstruktiver Existenzbeweise in der Graphentheorie und hier speziell der Ramseytheorie geht auf das Interesse eines PG-Teilnehmers zurück, Genetische Algorithmen und verwandte Verfahren dort einzusetzen. Zunächst wird ein kurzer Überblick über die Problemstellung gegeben. Die Eignung dieses Problems als Testproblem in MooN, sowie die Eignung MooNs als Arbeitsumgebung für dieses Problem soll abschließend motiviert werden.

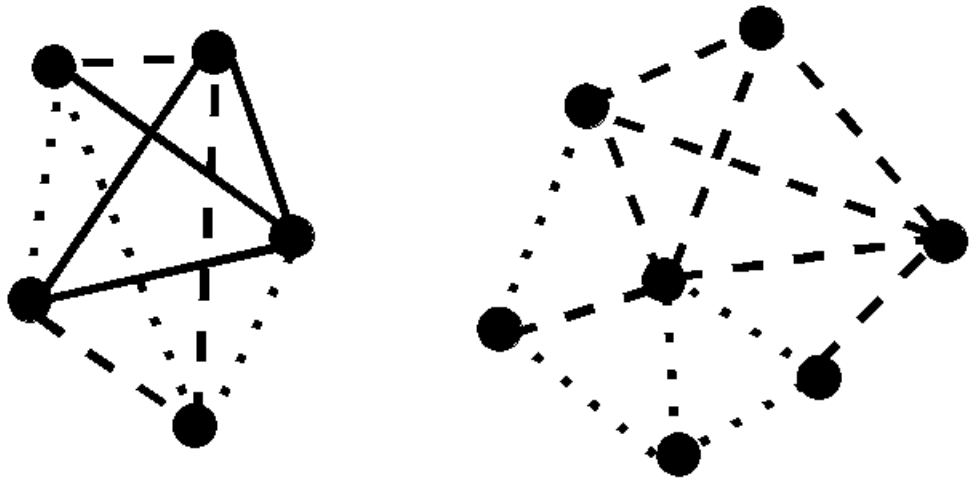

Abbildung 5.8: Beispielgraphen. Die Linienstile repräsentieren verschiedene Farben. Links: Ein 3-gefärbter K_5. Rechts: Ein Graph mit einer gepunkteten 3-Clique und einer gestrichelten 4-Clique.

Mathematische Grundlagen

Für eine eingehendere Beschreibung von graphentheoretischen Grundbegriffen sei auf [Die00] verwiesen. Hier sollen nur kurz drei Begriffe erklärt werden:

- Ein *vollständiger Graph* ist ein Graph, bei dem jede mögliche Kante auch einmal vorhanden ist. Der vollständige Graph mit n Ecken wird auch K_n genannt.
- Eine *Kantenfärbung* ist eine Abbildung, die jeder Kante eines Graphen eine Farbe zuordnet. Ein Graph, dessen Kanten auf k Farben abgebildet werden, heißt auch *k-kantengefärbt*.
- Eine *Clique* ist ein vollständiger Teilgraph, eine Clique mit k Ecken heißt auch *k-Clique*. Bei einem kantengefärbten Graphen ist eine *einfarbige Clique* ein solche, bei der alle Kanten die gleiche Farbe haben.

Abbildung 5.8 illustriert diese Begriffe.

Der Satz von Ramsey

Ausgangspunkt der Überlegungen ist der Satz von Ramsey (1930) (in [Ram30]). Dieser ist ein wichtiges Ergebnis der Graphentheorie und kann auch als kombinatorische Aussage aufgefasst werden. Eine mögliche Formulierung des Satzes ist (vergleiche z. B. [Rad94]):

> Seien i, j ganze Zahlen mit $i \geq 2$, $j \geq 2$ gegeben. Dann existiert $R = R(i, j)$, sodass:
>
> Jeder {rot, blau}-kantengefärbte vollständige Graph G mit mindestens R Ecken enthält einen rote i-Clique oder eine blaue j-Clique.

Die Zahl $R(i, j)$ heißt Ramseyzahl von i und j. Es sind nur wenige Ramseyzahlen bekannt, die folgende Tabelle gibt einen Überblick:

R(i,j)	**3**	**4**	**5**	**6**	**7**	**8**	**9**
3	6	9	14	18	23	28	36
4		18	25	[35; 41]	[49; 61]	[56; 84]	[69; 115]
5			[43; 49]	[58; 87]	...	...	...
6				...	...	...	...

Die Werte entstammen [Rad94]. Eine Zahl bedeutet, dass die Ramseyzahl bekannt ist. Ansonsten ist die beste bekannte unteren und obere Schranke angegeben.

Aus der Theorie ergeben sich jedoch obere bzw. untere Schranken. Während die oberen Schranken eher theoretischer Natur sind, erhält man die unteren durch Konstruktionen von Graphen. Das Auffinden von Graphen einer bestimmten Größe ohne einfarbige Cliquen zu gegebenen Werten von i und j ist das eigentliche Problem, das zur Debatte gestellt wird. Es kann als Optimierungsproblem formuliert werden, indem man die Zahl der verbotenen, einfarbigen Cliquen als Fitnesswert identifiziert und diesen zu minimieren versucht.

Bisherige Ansätze dieses Problem zu lösen beinhalten:

- Konstruktionen aus Teilgraphen mittels kleinerer Ramseyzahlen
- Mathematische Konstruktionen
- Klassische lokale Suchverfahren
- Genetische Algorithmen (siehe [KN03])

Formal ist das Auffinden eines Graphen mit Fitnesswert 0 ein konstruktiver Existenzbeweis. Die im nächsten Abschnitt vorgebrachten Argumente für einen Einsatz von MooN auf dieser Problemstellung lassen sich auf viele weitere Probleme der diskreten Mathematik übertragen, für die eine solche Beweisführung möglich ist. Exemplarisch seien hier nur die Schurzahlen [AH72] genannt.

Eigenschaften des Optimierungsproblems

Es können leicht einige wesentliche Eigenschaften des Problems ausgemacht werden. Zunächst sollte festgestellt werden, dass eine vernünftige Problemgröße Graphen mit $30 - 100$ Ecken umfasst. Diese Werte werden durch die bekannten Grenzen für kleine Ramseyzahlen einerseits und andererseits durch die Tatsache, dass die Komplexität des Problems exponentiell anwächst, motiviert.

Es gibt — ohne Rücksicht auf Isomorphie — $2^{\frac{n(n-1)}{2}}$ Graphen mit n Ecken. Bei $n = 20$ sind das ca. 10^{57}, bei $n = 100$ ca. 10^{1490} Graphen. Dieser sehr große Suchraum ist allerdings sehr flach. Fast alle Kanten eines gegebenen Graphen können umgefärbt werden, ohne die Fitness zu ändern. Die komplette Berechnung der Fitness eines gegebenen Graphen erfordert $\binom{\binom{n}{2}}{max(i,j)}$ Schritte, das sind zuviele, um häufig angewendet zu werden. Eine mögliche Lösung ist es, eine Heuristik zu verwenden, die die Fitness nicht komplett neu berechnet,

sondern gemäß der Veränderungen aktualisiert. Beim Umfärben einer Kante, was ein Standardoperator auf kantengefärbten Graphen ist, muss nur ein Bruchteil der möglichen Cliquen tatsächlich überprüft werden, was viel Zeit spart.

Ein Einsatzgebiet für MooN?

Es können einige Argumente dafür angeführt werden, ein Problem-Plug-In basierend auf obigen Überlegungen in MooN zu integrieren. Als Vorteile für MooN lassen sich identifizieren:

- Die Integration eines nichtalltäglichen, mathematischen Problems.
- Das Problem ist ein schwieriges kombinatorisches Problem und kann als Referenzproblem für die Heuristiken genutzt werden, da Ergebnisse sehr gut vergleichbar sind.
- Ein Hybrid-Ansatz, der in MooN realisierbar ist, wäre vielversprechend.
- Es könnte ein praktischer Nutzen entstehen, falls durch den Einsatz von MooN, vorbereitend oder in der Optimierung, neue theoretische Ergebnisse bekannt werden.
- Es kann ein Wissenstransfer zwischen Informatik und Mathematik angeregt werden, da Informatiker geeignete Verfahren entwickeln, diese aber i. d. R. nicht auf mathematische Probleme anwenden und Mathematiker nichtdeterministischen Verfahren in der Optimierung oft skeptisch gegenüber stehen.

Etwas kritischer muss das Urteil aus Problemsicht ausfallen. Ein Hauptvorteil von MooN ist die Möglichkeit, beliebige neue Heuristiken, auch Hybride, auf dem Problem zu testen. Zudem erlaubt die Ablaufsteuerung, insbesondere die GUI, die einfache Handhabung von Tests. Als gravierender Nachteil von MooN ist jedoch anzusehen, dass keine aktualisierende Fitnessauswertung möglich ist. Für den ernsthaften Optimiereinsatz wird MooN auf diesem Problem deshalb zu langsam sein.

Als Fazit bleibt das mögliche Ziel, mit Hilfe von MooN geeignete Verfahren zu identifizieren und zu vergleichen, also eine Suche „in die Breite“ der Heuristiken vorzunehmen. Die eigentliche, intensive Optimierungsarbeit können dann optimierte, evtl. in einer schnellen Programmiersprache geschriebene Programme übernehmen. MooN wäre somit nur die erste Phase in einem Optimierungsprozess.

5.2.8 Implementationaspekte von ACO und PSO

Daniel Blum

Implementierungsaspekte von Ant Colony Optimization (ACO)

Die zentrale Informationseinheit bei ACO ist die Pheromonmatrix. In ihr wird das Wissen über erzeugte und bewertete Lösungen durch die Anhäufung von Pheromon gesammelt. Da ACO auf Graphen der Dimension n arbeitet und für jede Kante im Graph ein Pheromonwert benötigt wird, ist die Größe der Pheromonmatrix bei gerichteten Graphen $n * n$. Für ungerichtete Graphen reicht eine entlang der Diagonale halbierte Matrix aus. Als Datentyp empfiehlt sich hier ein Array aus Elementen des primitiven Typs double, um eine hohe Genauigkeit zu erzielen, aber gleichzeitig schnelle Zugriffe zu ermöglichen.

Die Ameisen, die nun Lösungen erzeugen, lassen sich gut als eigene Klasse realisieren. Zum einen wird hier die Funktionalität untergebracht, mit der Lösungen erzeugt werden können, zum anderen kann hier der gefundene Weg als Information gespeichert werden. Auch das Verteilen des Pheromons nach Ermittlung der Fitness kann hier angesiedelt werden. Es bietet sich an, gewisse Informationen bezüglich des Verhaltens der Ameisen, z. B. die Pheromonmatrix, in einer Klasse zur Verfügung zu stellen, der Kolonie. Die Ameisen haben Zugriff auf ihre jeweilige Kolonie und können die Informationen dort abrufen.

Ein entscheidender Faktor für die Laufzeit von ACO ist die Anzahl der Ameisen, da für jede Ameise fast der komplette ACO-Algorithmus durchlaufen wird. Lediglich die Evaporation der Pheromonmatrix ist von der Ameisenzahl unabhängig, wird jedoch in der Laufzeit von der Lösungsermittlung dominiert. Weiterhin wirkt sich die Dimensionsgröße n quadratisch aus. Denn jede Ameise besucht zur Lösungserzeugung n Knoten im Graph, und trifft an jedem dieser Knoten eine Entscheidung, bei der sie $(n-1)$ Knoten betrachtet.

Zur Entscheidungsfindung, welcher Knoten als nächstes besucht wird, kann die Rouletteradtechnik eingesetzt werden. Jeder der Nachbarknoten x_i soll mit einer bestimmten Wahrscheinlichkeit $p(x_i)$ gewählt werden. Dazu werden alle x_i als Kreisabschnitte auf einem imaginären Rouletterad interpretiert, sodass die Größe jedes Abschnitts proportional zu $p(x_i)$ ist, und alle Abschnitte zusammen einen kompletten Kreis ergeben. Es wird nun eine

zufällige Position auf dem Rouletterad erzeugt, und der zugehörige Abschnitt gewählt. Die Darstellung des Rouletterades kann mit Hilfe eines Arrays simuliert werden, in dessen Feldern die Größe des jeweiligen Abschnitts gespeichert ist.

Schließlich ist noch zu bemerken, dass ACO nur Lösungen für Permutationsprobleme erzeugen kann.

Implementierungsaspekte von Particle Swarm Optimization (PSO)

Bei PSO ist das Wissen über bisherige Lösungen in den jeweiligen Individuen gespeichert. Diese sind am besten als eigene Klasse zu implementieren, da sie zum einen die Lösung tragen, und zum anderen weitere Informationen bezüglich ihrer Geschwindigkeit im Lösungsraum. Die Hauptklasse kümmert sich um die Verwaltung der Individuen und verändert sie von Generation zu Generation.

Die Schwarmgröße bestimmt die Laufzeit, da die Berechnung der Individuen der Hauptbestandteil des Algorithmus ist. Bei der Problemdimension n müssen für jedes Individuum $2n$ Werte neu berechnet werden, n Lösungsvektoreinträge und n Geschwindigkeitsvektoreinträge.

5.2.9 Räuber-Beute-Systeme

Igor Vatolkin

In diesem Vortrag wurde die mehrkriterielle Optimierung am Beispiel eines Räuber-Beute-Modells vorgestellt.

Multikriterielle Optimierung

Einführung

Viele praxisrelevante Optimierungsprobleme haben mehrere, in Konflikt stehende Optimierungskriterien. Je nachdem, welches Kriterium man stärker gewichtet, erscheint die eine oder andere Lösung vorteilhafter. Da eine Gewichtung meist nicht vorab angegeben werden kann, versucht man bei der multikriteriellen Optimierung, gleich alle Lösungen zu finden, die bei irgendeiner denkbaren Gewichtung der Kriterien optimal wären. Diese werden auch als Pareto-optimale Lösungen bezeichnet. Der Entscheidungsträger kann dann nach der Optimierung aus dieser Menge von Alternativen die für seine Gewichtung beste Lösung auswählen.

Das im Vortrag vorgestellte Beispiel aus [Deb01] handelte vom Kauf eines Autos, wobei die Kosten sowie die Qualität wichtig sind. Die Kosten sollten minimiert und die Qualität maximiert werden. Um das Optimum zu ermitteln, benötigt man weiterhin eine Informationsbasis. Dazu könnte z. B. der Kontostand des Käufers, die tägliche Fahrleistung usw. gehören.

Man kann im Allgemeinen auf zwei verschiedene Weisen vorgehen, um multikriterielle Probleme mit Optimierungsverfahren zu lösen. Die erste Vorgehensweise ist in 5.9 und die zweite in 5.10 illustriert.

Eigenschaften der multikriteriellen Optimierung

Das Ziel multikriterieller Optimierung ist es, Lösungen zu finden, die nah an der Pareto-optimalen Front liegen und diskret sind. Es werden zwei mehrdimensionale Suchräume betrachtet: der Parameterraum, der die Eingabewerte für die Optimierung definiert, sowie der Zielraum. Dabei müssen die Nachbarn im Parameterraum nicht unbedingt die Nachbarn im Zielraum sein. Evolutionäre Algorithmen sind gut geeignet, um solche multikriterielle Probleme zu lösen.

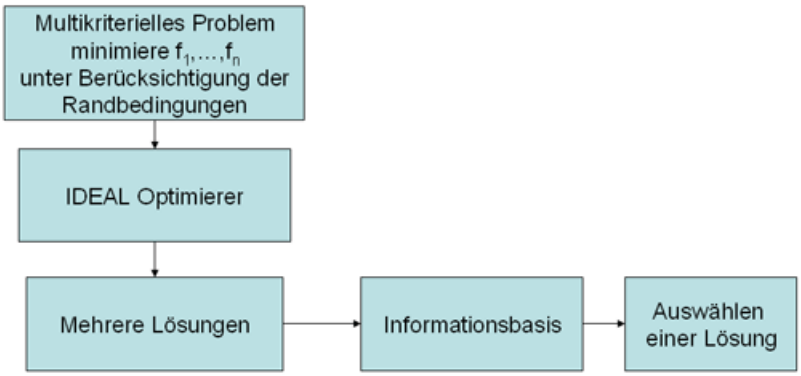

Abbildung 5.9: Erste Vorgehensweise bei der multikriteriellen Optimierung.

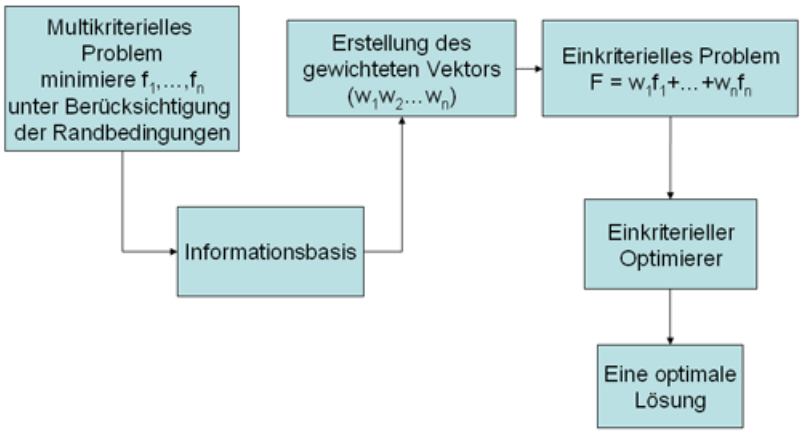

Abbildung 5.10: Zweite Vorgehensweise bei der multikriteriellen Optimierung.

Räuber-Beute-Modell

Einführung

Das vorgestellte Modell basiert auf der Evolutionstheorie, insbesondere auf dem Verhältnis zwischen Räuber- und Beutepopulationen. Die Beute repräsentiert die potenziellen Lösungen und wird auf einem 2D-Gitter abgebildet. Die Räuber werden mit den Zielfunktionen assoziiert. Sie bewegen sich horizontal und vertikal im Suchraum und töten die schwächsten Beuteindividuen.

Parameter

Folgende Parameter sind zu beachten:

- Beutemenge,
- Menge der Selektionskriterien auf der Beutemenge,
- Räubermenge,
- Nachbarschaftsgraph (ungerichtet, verbunden),
- Crossoveroperator,
- Mutationsoperator und
- Räuberwanderungsfunktion.

Weitere Parameter sowie eine genaue Beschreibung sind in [Lau99] dargestellt.

Beispiel

In 5.11 optimiert der Räuber 6 die Funktion f_1, Räuber 15 optimiert f_2. Im nächsten Zug eliminiert f_1 Beuteindividuum 7, f_2 eliminiert 14.

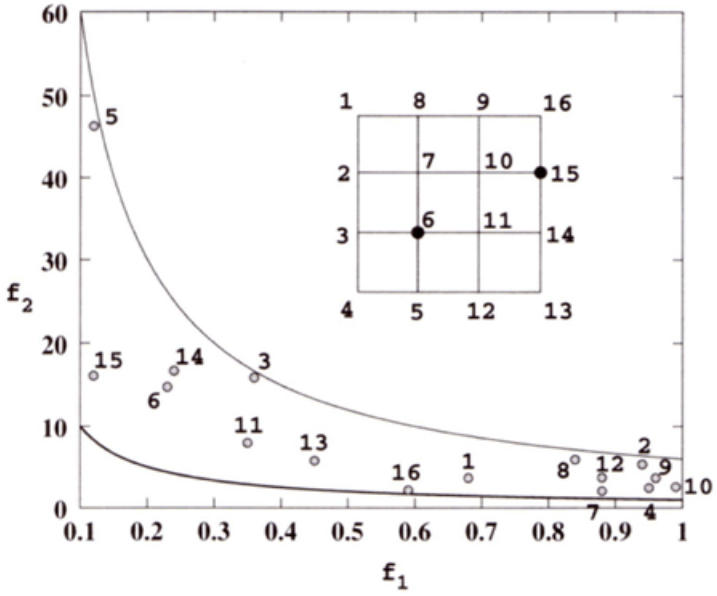

Abbildung 5.11: Aus [Deb01].

Weitere Überlegungen zum Algorithmus

Als Vorteil des Algorithmus kann man seine Einfachheit nennen und die Tatsache, dass nichtdominante Lösungen nicht bevorzugt werden. Ein Nachteil ist, dass immer die schlechtesten Individuen entfernt werden und sich der Algorithmus nicht um Lösungen mittlerer Güte kümmert.

Der Algorithmus kann durch Variieren der Parameter abgeändert werden. Außerdem gibt es erweiterte Räuber-Beute-Modelle, in denen ein Räuber nicht nur mit einer Funktion, sondern mit einem gewichteten Vektor aus den Funktionswerten assoziert wird.

5.2.10 Population Based Incremental Learning

Marko Tosic

Population Based Incremental Learning (PBIL) ist eine Heuristik, die Baluja in [BC95] vorgestellt hat. Sie ist von Genetischen Algorithmen (GA) abgeleitet. Diesen ähnlich wird in jedem Iterationsschritt eine Selektion vorgenommen, und es finden gelegentlich Mutationen statt. Einen Crossover-Operator gibt es jedoch nicht. An seine Stelle tritt ein reellwertiger Wahrscheinlichkeitsvektor. In diesem ist für jedes Bit die Wahrscheinlichkeit gespeichert, mit welcher es den Wert 1 annimmt.

Anstatt die Population explizit zu speichern wird in jedem Iterationsschritt anhand des Wahrscheinlichkeitvektors eine Menge von Kandidaten erzeugt. Von diesen Kandidaten werden in der Selektionsphase die besten ausgewählt. Anhand dieser Auswahl wird nun der Wahrscheinlichkeitsvektor aktualisiert.

Aktualisieren des Wahrscheinlichkeitsvektors

Die Art und Weise, in der von den selektierten Individuen gelernt wird, hat ihren Ursprung im Competitive Learning (CL). Es werden nur die besten Kandidaten ausgewertet. Dazu werden der Reihe nach die in dem Vektor gespeicherten Wahrscheinlichkeiten in Richtung des Wertes des Bits in dem Kandidaten hin angepasst. Die Geschwindigkeit dieser Anpassung kann über einen Parameter, die so genannte „Learning Rate“, eingestellt werden. Als Formel lässt sich dies darstellen als

$$P_{neu}(i) = P_{alt}(i) * (1 - R_L) + B(i) * R_L$$

Dabei wird der neue Wert P_{neu} der Wahrscheinlichkeit, dass das i-te Bits auf 1 gesetzt wird, berechnet aus dem vorherigen Wert P_{alt}, der Learning Rate R_L, sowie dem Wert des i-ten Bits des besten Vektors $B(i)$. Des Weiteren kann nicht nur von den besten Individuen gelernt werden, sondern auch von den schlechtesten, indem der Wahrscheinlichkeitsvektor in die entgegengesetzte Richtung verändert wird. Dies hat sich in Experimenten auch als sinnvoll erwiesen.

Mutation

Das Problem in einem lokalen Optimum mit der Suche „hängen“ zu bleiben ist bei PBIL genauso gegeben wie bei GA. Daher wurde der Mutationsoperator übernommen. Es gibt bei PBIL mehrere Möglichkeiten, ihn zu realisieren. Man kann beispielsweise die Mutation aus einem GA übernehmen und auf die erzeugten Kandidaten anwenden. Eine andere Möglichkeit wäre, die Mutation direkt auf dem Wahrscheinlichkeitsvektor durchzuführen. Hierbei kann die Auswirkung der Mutation über einen Parameter, den so genannten „Mutation Shift“, festgelegt werden. Dieser gibt für einen Vektor den Betrag an, um den er seinen Wert ändert.

Leistung von PBIL

Baluja hat in [BC95] PBIL vorgestellt und auch in Experimenten nachgewiesen, dass es besser optimiert als der Standard-GA. Untersucht wurden hierbei jeweils mehrere Instanzen von Problemen, die typischerweise gut mit GA bearbeitet werden können, wie etwa die wohl bekannten Probleme TSP, Jobshop Scheduling, Knapsack, Bin Packing und ANN. Lediglich bei fünf der insgesamt 27 Instanzen wurde nicht von PBIL am besten optimiert.

Varianten

Es sind einige Varianten des PBIL denkbar und zum Teil auch schon untersucht worden. Wie bereits erwähnt kann statt nur von den besten Lösungen auch von den schlechtesten gelernt werden. Eine ebenfalls vielversprechende Variante ist das inkrementelle Aktualisieren des Wahrscheinlichkeitsvektors mit jeder neu erzeugten Lösung. Die Veränderung des Vektors wird dann gemäß des Fitnesswertes der Lösung vorgenommen. Naheliegend ist die Variante, mehrere PBIL parallel laufen zu lassen, wie dies auch bei GA praktiziert wird, um der genetischen Drift entgegenzuwirken.

Fazit

PBIL ist ein vielversprechendes Optimierungsschema, das seine Leistungsfähigkeit im Experiment bereits unter Beweis gestellt hat. Es ist daher in jedem Fall erstrebenswert, bei künftigen Problemstellungen auch PBIL in Betracht zu ziehen.

5.2.11 Variable Neighbourhood Search

Djamila Lindemann

Variable Neighbourhood Search (VNS, siehe [Thi03]) wurde 1995 von Hansen und Mladenović entwickelt. Sie ist eine Erweiterung der lokalen Suche.

Bei der lokalen Suche wird von der momentanen Lösung aus in einer Nachbarschaft nach einer besseren Lösung gesucht. Anschließend sucht der Algorithmus in der Nachbarschaft der besseren Lösung nach einer noch besseren Lösung. Dies führt schnell und sicher zu einem lokalen Optimum.

Für die VNS wird eine feste Menge von k_{max} Nachbarschaften definiert und ihnen eine Reihenfolge $N_1, ..., N_{k_{max}}$ gegeben. Nun findet für eine zufällige Lösung x eine lokale Suche in N_1 statt. Falls diese Suche eine bessere Lösung findet, fährt sie für die bessere Lösung in deren Nachbarschaft fort. Falls keine bessere Lösung gefunden wurde, geht der Algortihmus zur nächsthöheren Nachbarschaft über und führt dort eine lokale Suche auf dem momentan besten Element durch.

Nachbarschaften

Eine Nachbarschaft der Lösung x ist eine Menge von Lösungen, die im selben Suchraum liegen wie x. Sie stehen zu x in einer bestimmten Beziehung.

Zum Beispiel können beim TSP die Nachbarschaften zu einer Tour t so definiert werden, dass $N_1(t)$ die Menge der Touren ist, die durch Anwendung eines `2-Opts` auf t entstehen, dass $N_2(t)$ die Touren sind, die durch ein `3-Opt` aus t entstehen und $N_3(t)$ die Touren sind, die durch ein `4-Opt` aus t entstehen.

Ein anderes Beispiel wäre es, einen optimalen Bitstring einer vorgegebenen Länge zu suchen. Dann wären die Nachbarschaften eines Bitstrings b auf folgende Weise wählbar: $N_1(b)$ sind alle Bitstrings, die durch flippen eines Bits der momentanen Lösung b generiert werden. $N_2(b)$ enthält die Bitstrings, die durch Flippen zweier Bits von b entstehen.

Eine Nachbarschaft eines Suchpunktes x kann also durch Operationen definiert werden, durch die von x aus zu den Elementen der Nachbarschaft gelangt werden kann, aber auch durch die Anzahl der zu anderen Elementen unterschiedlichen Bits.

Der Algorithmus

```
initialisiere();
repeat
        k := 1;
        repeat
                Local Search();
                Move();
        until k = k_max;
until stopping condition is met
```

Bei der Initialisierung werden Art und Reihenfolge der Nachbarschaften festgelegt, eine Startlösung erzeugt und die Abbruchbedingung der äußeren Schleife festgelegt.

In der inneren repeat-Schleife werden die Nachbarschaften mit Index 1 bis k_{max} durchlaufen bis die äußere repeat-Schleife dies abbricht. In der Phase $LocalSearch()$ wird der momentane Suchpunkt mit einer lokaler Suche optimiert.

Die Phase $Move()$ entscheidet, in welcher Nachbarschaft die Suche fortgesetzt wird. Falls die lokale Suche die bisherige Lösung nicht verbessern konnte, wird in der Phase $Move()$ der Index k um 1 erhöht. Wenn nach der Erhöhung $k \leq k_{max}$ gilt, geht es in der nächsthöheren Nachbarschaft mit der Phase $LocalSearch()$ weiter, sonst wird k wieder auf 1 gesetzt und die innere Schleife wird weiter durchlaufen. Falls die lokale Suche eine Verbesserung erzielt hat, wird in der Phase $Move()$ der Index auf 1 gesetzt und erneut die innere Schleife durchlaufen.

Vergleiche

Hansen und Mladenović (in [MH97]) haben zur Beurteilung der VNS auch fremde Untersuchungen herangezogen. In diesen wird die VNS z. B. auf TSP, VRP, p-Median-Problem, Weber-Problem, Graphenproblemen, Lastverteilungsproblemen und Problemen der künstlichen Intelligenz mit anderen Heuristiken verglichen. Sie kommen zu dem Schluss, dass die VNS in seiner Leistung vergleichbar ist mit vielen anderen Metaheuristiken, wie Tabu Search, Reactive Local Search, Multistart Local Search und ähnlichen.

5.2.12 Scatter Search

Vedran Divkovic

Einführung

Die Idee zu Scatter Search (SCS) stammt von Fred Glover aus dem Jahre 1968. Aber erst 1997 wurde der Algorithmus offiziell in [GLM03] vorgestellt. Hinter der Idee verbirgt sich ein allgemeines Konzept und kein bestimmtes Lösungsverfahren. Neue Lösungen werden durch die Kombination von Lösungen der Teilmengen einer Referenzmenge erzeugt.

Scatter Search Phasen

Es gibt zwei Phasen:

1. Bildung einer Referenzmenge:
   - Es wird eine Startmenge von Lösungen initialisiert.
   - Diese Menge wird mit Optimierungsverfahren verbessert.
   - Die besten Lösungen bilden die Referenzmenge (Diversität wird berücksichtigt).
2. Scatter Search Evolution:
   - In dieser Phase wird zunächst die Referenzmenge in Untermengen aufgeteilt.
   - Diese Untermengen werden miteinander kombiniert, um neue Lösungen zu erzeugen.
   - Wenn eine bessere Lösung gefunden wird, ersetzt diese die in der Referenzmenge schlechteste Lösung.

Anpassungen

Scatter Search muss für jedes Problem angepasst werden. Da die Architektur des SCS sehr allgemein ist, ist die Anpassung sehr einfach. Es müssen folgende vier Methoden überschrieben werden:

- Methode zur Verbesserung von Lösungen
- Methode zur Aktualisierung der Referenzmenge
- Methode zur Erzeugung von Teilmengen der Referenzmenge
- Methode zur Kombination von Teilmengen von Lösungen

Performance

Im Vergleich mit anderen Algorithmen für die betrachteten Probleme (siehe unten) bringt SCS sehr gute Ergebnisse hervor. Allerdings ist er langsamer, was sein größter Nachteil ist.

SCS vs. GA

SCS und GA sind populationsbasierte Verfahren. Im Gegensatz zu GA hat SCS keinen Mutationsschritt. Vorteil von SCS ist, dass es nicht auf das Kombinieren von nur zwei Lösungen limitiert ist, sondern dass mehrere Lösungen aus der Referenzmenge miteinander kombiniert werden können. Beim Scatter Search nehmen nicht nur die besten Lösungen teil, sondern auch diejenigen, welche die Diversität vergrößern. Je größer die Diversität ist, desto mehr Informationen hat man über die Lage und Qualität der Lösungen.

Anwendungen für SCS

Obwohl es sich um ein junges Verfahren handelt, wurde SCS schon auf viele Probleme angewendet und spezifiert, wie zum Beispiel das „Linear Ordering Problem“ ([RCT99]), das „Maximum Clique Problem“ ([RCT01]) oder „Graph Drawing“ ([LM99]). In allen Beispielen hat sich der Einsatz von SCS bewährt.

Fazit

Scatter Search ist ein sehr junges Verfahren, das noch erforscht werden muss. Bei den betrachteten Problemen wird die Qualität der Ergebnisse beachtlich verbessert. Der einzige Nachteil ist die geringe Geschwindigkeit.

Kapitel 6

Glossar

Abbruchbedingung

Dieses Plug-In legt fest, wann ein Einzellauf abgeschlossen ist. Das könnte z. B. der Fall sein, wenn eine vorgegebene Anzahl von Generationen oder ein bestimmter Zeitpunkt erreicht wurde.

Ablaufsteuerung

Wenn ein Gesamtlauf zusammengestellt wurde, kann er ausgeführt werden. Die Ablaufsteuerung wird vom Nutzer bedient und bietet die Möglichkeit des Startens, des Abbrechens oder zwischenzeitlichen Stoppens eines aktiven Gesamtlaufs, sowie die Zuschaltung bzw. Konfiguration einer Laufzeitvisualisierung.

Algorithmus, evolutionärer

Als Grundlage für einen evolutionären Algorithmus wird eine Population bestimmter Individuen betrachtet, die sich nach Evolutionsprinzipien aus der Biologie entwickelt. Individuen werden mit ihren Eigenschaften, die von einem zum anderen Algorithmus stark variieren können, und ihrem Fitnesswert identifiziert.

Batchmodus

Im Batchmodus werden die Befehle eines Skripts nacheinander durch das Betriebssystem abgearbeitet. Allgemein kann man mit einer Batchdatei eine Folge verschiedener Befehle durch die Eingabe eines Dateinamens ausführen lassen. MooN lässt sich durch Skripte steuern und kann dadurch auch im Batchmodus laufen.

Complete Run

s. Gesamtlauf

Datenausgabe

Die Resultate der Arbeit der Heuristiken lassen sich zur späteren Analyse oder Verwendung in Statistikprogrammen ausgeben. Dafür wurden Kategorien eingeführt. Jede Kategorie ist für einen Ausgabewert zuständig (z. B. das beste Individuum einer Population) und es kann festlegt werden, in welchen Intervallen die Information in welche Dateien geschrieben werden soll.

Defaultparameter

In der Einarbeitungsphase in das MooN-Programm soll sich der Benutzer nicht mit zu vielen Problemdetails beschäftigen müssen. Daher sollen die Parameter jedes Problems und jeder Heuristik Defaultwerte besitzen. Dies sind sinnvolle Werte, die vom Programmierer als Vorschlag festgesetzt wurden. Die Parameterwerte können vom Benutzer geändert und gespeichert werden.

Design, vollständiges faktorielles

Ein mögliches experimentelles Design. Dabei werden für jeden Parameter, der als „Faktor“ bezeichnet wird, genau zwei Einstellungen festgelegt, in denen er im Versuch getestet wird. Werden nun alle möglichen Kombinationen für alle Faktoren betrachtet, so ist dies ein „vollständiges faktorielles Design“. Das ergibt bei k Faktoren 2^k Möglichkeiten.

Design Of Experiments

Dies ist ein festes Schema, das bei den Tests benutzt wird, um gute Parametereinstellungen für Heuristiken zu finden. Es bezeichnet allgemein das Vorgehen, methodisch ein Experiment zu planen, um mit möglichst wenigen Versuchen möglichst genaue Aussagen treffen zu können.

Einzellauf (engl.: Single Run)

Zu einem Einzellauf gehört genau eine Heuristik, die auf genau ein Problem angewandt und von genau einer Abbruchbedingung beendet wird. Der Lauf initialisiert erst das Problem und dann die Heuristik, da diese Problemdaten braucht.

Evolver

Dieses Programm stammt von Thomas Michelitsch vom Institut für Spanende Fertigung der Universität Dortmund zur Bewertung einer Temperierbohrung. Basierend auf einer Menge von Punkten in einem Werkstück werden die Kosten der Bohrung, die Kühlleistung und die Qualität der Temperierbohrung ermittelt. Das Programm wird über eine TCP/IP Schnittstelle angesprochen und ist in MooN als Problem-Plug-In gekapselt.

Fitnessfunktion

Eine Fitnessfunktion berechnet den Fitnesswert eines Individuums. Dabei kann sie z. B. die Position des Individuums im Raum berücksichtigen.

Fitnesswert

Der Fitnesswert ist ein Kennwert, der die Güte einer speziellen Lösung zu einem Optimierungsproblem beschreibt. Ziel der Optimierung ist es, eine Lösung mit einem möglichst guten Fitnesswert zu finden.

Framework

Ein Framework ist eine Menge zusammenhängender Klassen, die einen wiederverwendbaren und erweiterbaren Entwurf für Softwareprodukte eines bestimmtes Typs darstellen. Für die Erweiterbarkeit werden oft Schnittstellen festgelegt.

Gesamtlauf (engl. Complete Run)

Ein Gesamtlauf ist eine Liste, die mehrere Einzelläufe in einer Sequenz zusammenfasst. Hierzu müssen diese zunächst konfiguriert werden. Danach werden sie nacheinander durchgeführt. Zum Verständnis des Begriffs Gesamtlaufs ist die Metapher der „Playlist“ hilfreich, wie sie in vielen Multimedia-Playern existieren. Ein konfigurierter Gesamtlauf kann gespeichert und geladen werden.

GPL (GNU General Public License)

Dies ist eine Open Source Lizenz für MooN, die Dritten erlaubt, den Code weiter zu verwenden und zu ändern, allerdings mit einigen Einschränkungen, z. B. dass weitere Implementierungen auch unter der GPL Lizenz entwickelt werden müssen. Siehe auch [Fre03].

GUI (Graphical User Interface)

Die GUI ist die Benutzeroberfläche von MooN und dient der Ablaufsteuerung.

Heuristik

Heuristiken sind erratische Verfahren zur Problemoptimierung. Im Gegensatz zum Algorithmus terminieren Heuristiken nicht und haben auch die gleiche Formalität. Beispiele sind Evolutionäre Algorithmen, Simulated Annealing oder Tabu Search. In der PG wird der Begriff Heuristik oft synonym mit Metaheurisitik verwendet.

Hotframe

Dies ist ein Framework zur Implementierung diverser Metaheuristiken und zur Nutzung dieser in einem zu schreibenden Decision Support System (Entscheidungsfindungs-System), siehe [Fin00].

Individuum

Gemäß des biologischen Vorbilds werden bei vielen Heuristiken Punkte im Suchraum Individuen genannt. Die zu einem Zeitpunkt aktuellen Individuen bilden die Population, die sich von Generation zu Generation verändert.

Instanziierung

Dies ist die Zuweisung von konkreten Werten zu Heuristiken oder Problemen.

Iterationsschritt

Nachdem eine Heuristik eine mögliche Lösung des Optimierungsproblems gefunden hat, versucht sie in einem Iterationsschritt die Lösung zu verbessern. Dies kann z. B. bei einem evolutionären Algorithmus die Berechnung der nächsten Population sein.

Kennzahl

Aus den exportierten Daten können bestimmte Kennzahlen wie Erwartungswert oder Varianz berechnet werden, die anschließend mit einem Visualisierungstool graphisch dargestellt werden können. Sie dienen zur Bewertung einer Heuristik und liefern Anhaltspunkte dafür, wie erfolgreich der Lauf beendet wurde.

Laufkonfiguration

Mit einer XML-Datei lässt sich die komplette Konfiguration eines Gesamtlaufes speichern. Dazu gehören die Parameterwerte der Heuristiken, Probleme und Abbruchbedingungen aller Einzelläufe.

Laufzeitdaten

Dies sind Daten, die während eines Laufs generiert werden. Sie können auf verschiedene Weise weiter benutzt werden, etwa zur statistischen Analyse. Die Laufzeitdaten werden in einer oder mehreren Dateien gespeichert.

Laufzeitvisualisierung

Sie ermöglicht das Anzeigen von Daten während eines Laufs und gibt dem Benutzer einen Einblick in den Optimierungsablauf. Die Visualisierung stellt die Fitness in einem zweidimensionalen Graphen mit beschrifteten und skalierten Achsen dar.

Logausgabe

Mittels Log4J werden alle Kommandozeilenausgaben von MooN ausgegeben. Man kann dabei Einstellungen vornehmen, die nur Ausgaben bestimmter Wichtigkeit erlauben sowie Ausgaben aus einem bestimmten Teil des Programms separat betrachten lassen.

Metaheuristik

Dies ist eine Strategie, die andere Heuristiken benutzt, um mögliche Lösungen für Optimierungsprobleme zu finden.

Module

Im Kontext des MooN-Projekts ist ein Modul ein nicht ins Hauptprogramm integrierter Bestandteil der Software. In unserem Entwurf sind Probleme, Heuristiken und Abbruchbedingungen Module. Technisch erfolgt die Realisierung des modularen Ansatzes über Plug-Ins.

MooN

MooN ist der Name des Produkts, bedeutet „Metaheuristical optimization for ordinary Needs“.

Operator

Ein Operator modifiziert ein oder mehrere Individuen im Laufe der Iterationsschritte. Beispiele für Operatoren sind Mutation oder Crossover.

Optimierungsproblem

Ein Optimierungsproblem ist eine zu maximierende bzw. minimierende Funktion, die jedoch nicht explizit gegeben sein muss. Üblicherweise sind solche Probleme nicht in polynomieller Rechenzeit lösbar.

Permutation

Eine Permutation der Menge {1,2,...,n} ist eine Anordnung dieser Zahlen ohne Auslassungen oder Wiederholungen.

Plug-In

Plug-Ins sind Programmteile, die durch entsprechende Plug-In-Schnittstellen in das Hauptprogramm integriert werden können. Im MooN-Projekt werden sowohl die Heuristiken als auch die Probleme und Abbruchbedingungen als Plug-Ins implementiert. Somit ist gewährleistet, dass MooN in Zukunft leicht erweiterbar sein wird.

Plug-In-Schnittstelle

Sie legt genau fest, welche Anforderungen ein Plug-In erfüllen muss. Die Schnittstelle definiert Anzahl, Art und Wertebereich aller Parameter, die erforderlich sind, um einen sinnvollen Austausch zwischen Programm und Plug-In zu ermöglichen.

Population

Eine Menge von Individuen wird als Population bezeichnet.

Problem

s. Optimierungsproblem

Prototyp

Der im Laufe des ersten PG-Semesters erstellte Prototyp diente der Evaluierung grundlegender Ideen und Vorstellungen, die sich bezüglich des später zu implementierenden Tools ergeben haben. Dabei wurde vor allem die technische Machbarkeit der Plug-Ins überprüft sowie versucht, ein tragfähiges Konzept zur Steuerung der Läufe zu finden.

Regressionsanalyse, lineare

Das ist die statistische Analyse der Abhängigkeiten von Faktoren voneinander unter der Annahme, dass diese linear sind, dass sich also ein Faktor als Linearkombination der anderen darstellen lässt.

Repräsentation

Als Repräsentation bezeichnet man die Codierung einer Lösung eines Optimierungsproblems. Die Einführung einer solchen Codierung erbrachte, dass alle Heuristiken und Probleme, die gleiche Repräsentationen benutzen, miteinander kombinierbar sind. Bei MooN gibt es zwei Arten von Repräsentationen - ein reellwertiger Vektor und die Permutation.

Schnittstelle

s. Plug-In-Schnittstelle

Single Run

s. Einzellauf

Skriptsteuerung

Sie bezeichnet die Steuerung eines Programms auf der Ebene des Betriebssystems mit Hilfe von Dateien, nämlich Skripten in einer Anweisungssprache, die nahe am Betriebssystem angesiedelt ist. MooN lässt sich nicht nur über die GUI steuern, sondern auch über Skripte. Dies ist leicht zu ermöglichen, da die Konfigurationen für Läufe in einem einheitlichen Format gespeichert werden. Skriptsteuerung ermöglicht es, verschiedene Läufe auf verschiedenen Rechnern zu starten, um Rechenzeit zu verteilen bzw. besser nutzen zu können. Auch ein Scheduling ist möglich.

Testfunktion

Als Testfunktionen werden Funktionen bezeichnet, die relativ einfach ausgewertet werden können, evtl. auch einfach strukturiert sind, aber gewisse, leicht ersichtliche Eigenschaften haben, die auch bei Optimierungsproblemen auftreten. Sie dienen zum Testen von Optimierungsalgorithmen und -heuristiken. Ziel ist es, das Verhalten auf Funktionen zu betrachten, die einerseits einfach zu verstehen und schnell auszuwerten sind, die aber andererseits wesentliche Schwierigkeiten der eingesetzten Verfahren bei gewissen strukturellen Grundannahmen aufzeigen können.

TWiki

TWiki ist eine Web-basierte Kollaborationsplattform, die auf der Idee der WikiWiki basiert. Durch eine einfache Syntax ist es den Beteiligten möglich, Einträge vorzunehmen, die beim Betrachten als HTML-Seiten dargestellt werden. Die Projektgruppe nutzte das TWiki zur Koordination ihrer Arbeit, zur Verwaltung der Protokolle und zur Dokumentation der geleisteten Arbeit.

Visualisierung, externe

Am Ende eines Laufs werden die generierten Daten in eine Datei exportiert und dort gespeichert. Mit Hilfe externer Visualisierungsprogramme können diese Daten anschließend graphisch dargestellt werden. Durch diese Art von Visualisierung ist der Verlauf des Optimierungsprozesses jederzeit nachvollziehbar.

Literaturverzeichnis

[AH72] H. L. Abbott and D. Hanson. On a problem of Schur and its Generalizations. *Acta Arithmetica*, 20:175–182, 1972.

[BB04a] Th. Bartz-Beielstein. Experimental Analysis of Search Heuristics – Overview and Comprehensive Introduction. *(Submitted to the EC journal)*, 2004.

[BB04b] Th. Bartz-Beielstein. Tuning Search Algorithms for Real-World Applications: A Regression Tree Based Approach. In *Proceedings of the 2004 Congress on Evolutionary Computation CEC2004*, 2004. In print.

[BBB+03] S. Balci, S. Blom, S. Blum, V. Divkovic, D. Hoppe, D. Lindemann, U. Schneider, B. Selzam, Th. Tometzki, M. Tosic, I. Vatolkin, and S. Walter. PG 431 - Metaheuristiken „Neue Ideen für die Optimierung“. Zwischenbericht Sommersemester, Universität Dortmund, Juli 2003.

[Bäc93] Th. Bäck. *Evolutionary Algorithms in Theory and Practice*. Dissertation, Universität Dortmund, 1993.

[BC95] S. Baluja and R. Caruana. Removing the Genetics from the Standard Genetic Algorithm. In A. Prieditis and S. Russel, editors, *The Int. Conf. on Machine Learning 1995*, pages 38–46, San Mateo, CA, 1995. Morgan Kaufmann Publishers.

[BG98] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[DD99] M. Dorigo and G. Di Caro. The Ant Colony Optimization Meta-Heuristic. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 11–32. McGraw-Hill, London, 1999.

[Deb01] K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley&Sons, Chicester, 2001.

[Die00] R. Diestel. *Graphentheorie*. Springer, Berlin, Heidelberg, New York, 2. edition, 2000.

[DJW99] S. Droste, Th. Jansen, and I. Wegener. Perhaps Not a Free Lunch But At Least a Free Appetizer. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the First Genetic and Evolutionary Computation Conference(GECCO '99)*, pages 833–839, San Francisco, CA, 13–17 1999. Morgan Kaufmann Publishers, Inc.

[DJW02] S. Droste, Th. Jansen, and I. Wegener. Optimization with Randomized Search Heuristics — The (A)NFL Theorem, Realistic Scenarios, and Difficult Functions. *Theoretical Computer Science*, 2002. to appear.

[DSW93] G. Dueck, T. Scheuer, and H.-M. Wallmeier. Toleranzschwelle und Sintflut: Neue Ideen zur Optimierung. *Spektrum der Wissenschaft*, pages 42–51, März 1993.

[EKS01] R. Eberhart, J. Kennedy, and Y. Shi. *Swarm Intelligence*. Academic Press, London, 2001.

[Far02] J. Faraway. Practical Regression and Anova using R. Online PDF document, Juli 2002. `http://cran.r-project.org/doc/contrib/Faraway-PRA.pdf`, Date of visit: 14.02.2004.

[Fin00] A. Fink. *Software-Wiederverwendung bei der Lösung von Planungsproblemen mittels Meta-Heuristiken*. Dissertation, Technische Universität Braunschweig, 2000.

[FP90] C. Floudas and P. Pardalos. A Collection of Test Problems for Constrained Global Optimization. *Lecture Notes in Comput. Science*, 455, 1990.

[Fre02] Free Software Foundation. Categories of free and non-free software. Online HTML document, Free Software Foundation, Inc., Boston, MA, Dezember 2002. `http://www.gnu.org/philosophy/categories.html`, Date of visit: 06.01.2004.

[Fre03] Free Software Foundation. GNU General Public License. Online HTML document, Free Software Foundation, Inc., Boston, MA, Mai 2003. `http://www.gnu.org/licenses/gpl.html`, Date of visit: 06.01.2004.

[FV02] A. Fink and S. Voß. HotFrame: A Heuristic Optimization Framework. In S. Voß and D. L. Woodruff, editors, *Optimization Software Class Libraries*, pages 81–154. Kluwer, 2002.

[Gül03] C. Gülcü. *The complete log4j manual*. QOS.ch, Montreux, Schweiz, 2003.

[GLM03] F. Glover, M. Laguna, and R. Martí. Scatter search. In A. Ghosh and S. Tsutsui, editors, *Advances in Evolutionary Computation: Theory and Applications*. Springer-Verlag, 2003.

[Gol89] D. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley, Bonn, 1989.

[HKS04] T. Husted, T. Kitahata, and J. S. Stevens. The Jakarta Site – The Jakarta Project – Java Related Products. Online HTML document, Apache Software Foundation, 2004. `http://jakarta.apache.org`, Date of visit: 27.01.2004.

[Hu03] X. Hu. PSO Tutorial. Online HTML document, Purdue University, 2003. `http://web.ics.purdue.edu/~hux/tutorials.shtml`, Date of visit: 10.01.2004.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, (220):671–680, 1983.

[KN03] D. Kunkel and P. Ng. Ramsey Numbers: Improving the bounds of R(5,5). In *Proc. 36th Ann. Midwest Instruction and Computing Symposium (MICS 2003)*, 2003.

[Lau99] M. Laumanns. Ein verteiltes Räuber-Beute-Modell zur mehrkriteriellen Optimierung. Diplomarbeit, Universität Dortmund, 1999.

[LK00] A. M. Law and W. D. Kelton. *Simulation modeling and analysis*. McGraw-Hill, third edition, 2000.

[LM99] M. Laguna and R. Martí. GRASP and Path Relinking for 2-Layer Straight Line Crossing Minimization. *INFORMS Journal on Computing*, 11:44–52, 1999.

[LRK01] M. Lvbjerg, T. Rasmussen, and T. Krink. Hybrid particle swarm optimiser with breeding and subpopulations. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*., 2001.

[MAB+01] S. Markon, D. V. Arnold, Th. Bäck, Th. Beielstein, and H.-G. Beyer. Thresholding - a Selection Operator for Noisy ES. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 465–472, Seoul, 2001. IEEE Press.

[Mal00] C. Malerczyk. Visualisierungstechniken für den Sintflutalgorithmus. *Friedberger Hochschulschriften*, (3), April 2000.

[MH97] N. Mladenović and P. Hansen. Variable Neighborhood Search. *Comps. in Opns. Res.*, 24:1097–1100, 1997.

[Mot03] R. Motzer. 2D-Visualisierung der Verfahren Simulated Annealing, Toleranzschwellen-Algorithmus und Sintflut-Algorithmus. Online HTML document, FH Augsburg, März 2003. `http://www.fh-augsburg.de/informatik/projekte/mebib/emiel/entw_inf/or_verf/2d_vis.html`, Date of visit: 03.01.2004.

[MRR+53] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of State Calculations by Fast Computing Machines. *J. Chem. Phys.*, (21, 6):1087–1092, 1953.

[MS96] Z. Michalewicz and M. Schoenauer. Evolutionary Algorithms for Constrained Parameter Optimization Problems. *Evolutionary Computation*, 4(1):1–32, 1996.

[MWM02] J. Mehnen, K. Weinert, and Th. Michelitsch. Evolutionäres Design der Temperierbohrungen für Spritzguss- und Druckgusswerkzeuge. *wt Werkstatttechnik online*, 92(H. 11/12):588–590, 2002.

[Nis97] V. Nissen. *Einführung in Evolutionäre Algorithmen*. Vieweg, Braunschweig, 1997.

[Obj04a] Object Mentor, Inc. JUnit API javadoc. Online HTML document, Object Mentor, Inc., Gurnee, IL, 2004. `http://www.junit.org/junit/javadoc/index.htm`, Date of visit: 26.01.2004.

[Obj04b] Object Mentor, Inc. JUnit, Testing Resources for Extreme Programming. Online HTML document, Object Mentor, Inc., Gurnee, IL, 2004. `http://www.junit.org/index.htm`, Date of visit: 26.01.2004.

[Pöp00] C. Pöppe. Optimieren mit Bomben. *Spektrum der Wissenschaft*, Mai 2000.

[Rad94] S. Radziszowski. Small Ramsey Numbers. *Electronic Journal of Combinatorics*, (DS1):28pp., 1994.

[Ram30] F. P. Ramsey. On a problem of formal logic. *Proc. London Math. Soc.*, 30:264–286, 1930.

[RCT99] C. Rigo, L. Cavique, and I. Themido. Scatter Search for the Linear Ordering Problem. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*. McGraw-Hill, 1999.

[RCT01] C. Rigo, L. Cavique, and I. Themido. A Scatter Search Algorithm for the Maximum Clique Problem. In C. C. Ribeiro, editor, *Essays and Surveys in Metaheuristics*. Kluwer Academic Publishers, 2001.

[Rec73] I. Rechenberg. *Evolutionsstrategie. Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog, Stuttgart, 1973.

[Rie97] E. H. Riedemann. *Testmethoden für sequentielle und nebenläufige Software-Systeme*. Teubner, Stuttgart, 1997.

[Rum03] A. Rummler. eaLib - a Java Evolutionary Computation Toolkit. Online HTML document, TU Ilmenau, 2003. `http://www.evolvica.org/ealib/index.html`, Date of visit: 28.01.2004.

[Sch75] H.-P. Schwefel. *Evolutionsstrategie und numerische Optimierung*. Dissertation, TU Berlin, 1975.

[Thi03] R. Thies. Variable Neighbourhood Search. In *Seminar Metaheuristiken für die Optimierung II*. Universität Dortmund, 2003.

[Ull02] T. Ullrich. Optimierung I, Zusammmenfassung des Tutoriums vom 27.05.2002. Online PDF document, Universität Karlsruhe, Mai 2002. `http://www.stud.uni-karlsruhe.de/~uagg/downloads/tutorium_06.pdf`, Date of visit: 03.01.2004.

[WM95] D. H. Wolpert and W. G. Macready. No Free Lunch Theorems for Search. Technical Report SFI-TR-95-02-010, Santa Fe, NM, 1995.

[WM97] D. H. Wolpert and W. G. Macready. No Free Lunch Theorems for Optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997.