technische universität
dortmund
Fachbereich Informatik

Endbericht
der Projektgruppe 527

Visualisierung und Simulation von Räuber-Beute-Systemen in der Mehrkriteriellen Optimierung

Projektgruppe 527

Prof. Dr.-Ing. Uwe Schwiegelshohn,
Institut für Roboterforschung

März 2009

Inhaltsverzeichnis

1 Einleitung

Viele Probleme, die in der Wirtschaft gelöst werden müssen, umfassen mehrere Ziele, die sich nicht gleichermaßen verfolgen lassen, sondern einander widersprechen. Diese Probleme werden unter dem Begriff mehrkriterielle Optimierung zusammengefasst.
Ein typisches mehrkriterielles Problem zum Beispiel ist die Entwicklung eines Autos. Offensichtliche Ziele, die sich der Entwickler setzt, sind der Preis des Autos, die Geschwindigkeit, die Größe, der Spritverbrauch, die Zielgruppe, die Sicherheit und der Komfort, wobei sich die Liste der Ziele beliebig erweitern lässt.

Zur Untersuchung und Lösung gibt es verschiedene Algorithmen und Modelle, die darauf basieren, alle Ziele zeitgleich zu bewerten. Diese Form der Algorithmen hat sich in den letzten Jahren zwar durchgesetzt, bringt aber einige Probleme mit sich. Zum einen können die Funktionen sehr komplex werden, wenn man alle Kriterien gleichzeitig betrachtet und in Relation setzt, zum anderen kann die Approximation der Lösungen beliebig schlecht sein.
Parallel zu diesen Algorithmen gab es ebenfalls Entwicklungen an anderen Modellformen, die sich bis jetzt aber noch nicht durchgesetzt haben. Eine solche Modellform ist das Räuber-Beute-Modell, welches die Grundlage für die Arbeit dieser Projektgruppe darstellt. Es unterscheidet sich von den oben erwähnten Verfahren im Wesentlichen dadurch, dass nicht alle Ziele gleichzeitig optimiert, sondern die Ziele unabhängig voneinander betrachtet werden.

Das Räuber-Beute-Modell

Die Motivation dieses Verfahrens entspringt aus der Biologie. Räuber-Beute-Modelle beschreiben die Wechselwirkung zwischen Räubern und Beute in der Natur. Räuber haben unterschiedliche Verhaltensweisen in der Jagd nach Beute, die sich immer wieder dem neuen Jagdverhalten ihrer Jäger anpassen muss, um zu überleben. Nur die Beute, die am besten angepasst ist, hat eine Chance den Angriffen der Räuber zu entkommen. Dieses Modell von Jägern und Gejagten lässt sich auf die mehrkriterielle Optimierung übertragen. Die Lösungsmöglichkeiten werden durch die Beute, die Ziele durch die Räuber repräsentiert. Mathematisch lässt sich diese Wechselwirkung und Systemdynamik durch Rekursionsgleichungen (Lotka und Volterra Regeln) beschreiben.
Die Lösungsmöglichkeiten werden dem Selektionsdruck der Ziele ausgesetzt. Schlechte Lösungen, die nur ein Ziel oder gar kein Ziel „gut“ lösen, werden eliminiert, gute Lösungen werden zur „Fortpflanzung“ vorgeschlagen. Durch Variation, das Rekombinieren mehrerer oder das Mutieren einzelner Beuteindividuen, werden neue Lösungen dem Suchraum hinzugefügt. Dieses Schema, das in der Natur den Lebenszyklus der Individuen, Beute sowie Räuber, widerspiegelt, stellt im algorithmischen Bereich einen evolutionären Algorithmus dar.

Das Räuber-Beute-Modell hat im Vergleich zu den anderen Modellformen den Vorteil, dass die Ziele/Kriterien wesentlich einfacher zu skalieren sind, da durch das Hinzufügen oder Herausnehmen von Räubern neue Ziele berücksichtigt oder bestehende Ziele vernachlässigt werden. Zudem ist das Formulieren einer Bewertungsfunktion für ein einzelnes Kriterium einfacher als bei Betrachtung der Gesamtkomplexität. Des Weiteren ist hier die Kenntnis der optimalen Lösungsmenge nicht erforderlich, um gute approximierte Lösungen zu erhalten. Im Vergleich zu den anderen Algorithmen ist das Räuber-Beute-Modell sehr leicht zu implementieren und eignet sich auch zur Parallelisierung. Neben den genannten Vorteilen hat dieses Modell aber auch einen erheblichen Nachteil. Im Gegensatz zu den anderen Verfahren ist das Verhalten des Modells noch kaum erforscht und somit eignet sich das Räuber-Beute-Modell nicht für den praktischen Einsatz. Es kann sowohl die Diversität der Lösungen nicht garantiert werden, als auch eine gute Approximation der Lösungsmenge. Das heißt, dass die Lösungsmenge unter Umständen nur gute Kompromisslösungen oder gute Lösungen für einzelne Kriterien umfasst und nicht alle Lösungsmöglichkeiten gleichmäßig abdeckt.

Um gezielte Untersuchungen der Komponenten und Zusammenhänge des komplexen Modells zu ermöglichen, hat sich die Projektgruppe 527 mit der Erstellung eines flexiblen Frameworks für das

Räuber-Beute-Modell beschäftigt. Der Schwerpunkt lag im Wesentlichen im Softwaredesign und in der Softwareentwicklung. Der Prozess der Softwareentwicklung war in zwei Bereiche geteilt. Zum einen in die Ausarbeitung der Ziele und Anforderungen, die das Framework erfüllen sollte (siehe Kapitel 2), zum anderen in das eigentliche Implementieren und Testen des Frameworks (siehe Kapitel 4 und 5).

2 Zielsetzung

Die Zielsetzung der Projektgruppe setzt sich im Wesentlichen aus zwei Teilaspekten zusammen. Zum einen aus den Anforderungen der Betreuer, die das minimale Ziel der Projektgruppe definieren; zum anderen aus den selbst gesetzten Zielen der Projektgruppe.

2.1 Anforderungen und Ziele im Einzelnen

Die Projektgruppe soll im wesentlichen drei Ziele verfolgen: Entwicklung des Frameworks, Simulation der Systemdynamik und Parallelisierung des Systems.

2.1.1 Entwicklung des Frameworks

Es soll ein Baukastensystem entwickelt werden, mit welchem flexibel Instanzen des Räuber-Beute-Modells simuliert werden können. Zunächst sollen einfache Instanzen behandelt werden, das heißt aktive Räuber, passive Beute und eine einfache mehrkriterielle Funktion wie zum Beispiel Multi-Sphere (Kapitel 4.3.1). Das Framework soll aber erweiterbar sein, so dass problemlos komplexere Instanzen simuliert werden können, wie zum Beispiel sich bewegende Beute oder Beute die ebenfalls jagen kann. Konzepte und Lösungen sind im Kapitel 4 beschrieben.

2.1.2 Simulation der Systemdynamik

Des Weiteren soll eine Simulation entwickelt und implementiert bzw. integriert werden, die Einblicke in die Systemdynamik und Evolution erlaubt. Diese Visualisierung soll das System des Räuber-Beute-Modells verdeutlichen. Dabei ist die Struktur (Räumliche Struktur der Population) des Modells zu berücksichtigen. Es soll zum einen möglich sein globale Veränderungen im System zu erkennen, zum anderen aber auch lokale Besonderheiten herauszukristallisieren.
In der Projektgruppe sind verschiedene Konzepte für die Ausgabe und Simulation erstellt worden. Zum einen gibt es einen Ausgabemanager, der über Events über Veränderungen im System benachrichtigt wird. Diese Informationen werden in Dateien geschrieben, welche anschließend mit Programmen wie MatLab oder R analysiert werden können. Desweiteren gibt es auch Metriken, die während des Durchlaufs Plots erzeugen, die die Veränderungen und Verbesserungen beschreiben (siehe Kapitel 4.7).

2.1.3 Parallelisierung des Systems

Das Framework soll parallelisisert werden. Es gibt im Wesentlichen zwei Arten der Parallelisierung. Zum einen ist es in der Natur nicht nur so, dass Räuber im Räuber-Beute-Modell unabhängig von einander agieren, sondern auch zeitgleich. Dieses Verhalten soll durch die Parallelisierung der Räuber betrachtet werden. Es ist unklar wie diese Parallelisierung sich auf das Verhalten des Räuber-Beute-Modells auswirkt. Die Konzepte und Lösungen zur Unterstützung dieses Ansatzes sind in Kapitel 4.6 beschrieben.
Die andere Möglichkeit der Paralellisierung besteht darin, zeitaufwändige Berechnungen, wie die Berechnung der Fitness, zu parallelisieren. Diese Art der Parallelisierung verkürzt lediglich die Rechenzeit, wirkt sich aber nicht auf das Verhalten des Räuber-Beute-Modells aus. Details hierzu sind in Kapitel 4.5 zu finden.

Um das minimale Ziel der Projektgruppe zu erfüllen, sollte eine verteilte Architektur für das Räuber-Beute-Modell konzipiert und implementiert, Simulations- und Visualisierungsmethoden konzipiert und integriert sowie eine ausführliche Dokumentation der Architektur und der Methoden verfasst werden.

3 Organisation der Projektgruppe

Im folgenden Kapitel wird die Organisation der Projektgruppe beschrieben. Dazu gehört zum einen die Organisationsform der Gruppe und das Projektmanagement, zum anderen aber auch der zeitliche und thematische Ablauf der Projektarbeit.

3.1 Zeitlicher und thematischer Ablauf

Die Arbeit der Projektgruppe ist im Wesentlichen in vier Phasen unterteilt:

- Seminarfahrt
- Sammeln und Testen der Designvorschläge
- Design und Implementierung des Frameworks
- Testen des Frameworks und Analyse der Ergebnisse

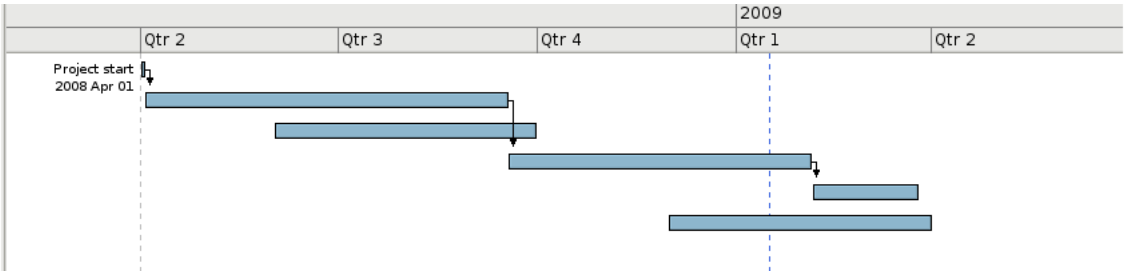

Abbildung 1: Ganttdiagramm zum Ablauf der Phasen der Projektgruppe

Tasks

WBS	Name	Start	Finish	Work
1	Seminarfahrt	Apr 1	Apr 2	2d
2	Designvorschläge sammeln	Apr 3	Sep 17	120d
3	Zwischenbericht	Jun 2	Sep 30	87d
4	Design & Implementierung	Sep 18	Feb 4	100d
5	Testen & Analyse	Feb 5	Mar 25	35d
6	Endbericht	Dec 1	Mar 31	87d

Abbildung 2: Ganttdiagramm: Auflistung der Aufgaben

Die Ziele und Ergebnisse der vier Phasen werden hier kurz vorgestellt.

3.1.1 Phase 1 - Die Seminarfahrt

Mit der Seminarfahrt zu Beginn der Projektgruppe wurden im Wesentlichen zwei Ziele verfolgt. Zum einen sollten die Teammitglieder und Betreuer sich untereinander kennen lernen, zum anderen sollten alle auf einen Wissensstand gebracht werden. Dazu wurden die relevanten Themen der Projektgruppe wie Parallelisierung, Visualisierung, Softwareentwicklung und Evolutionären Algorithmen von den Teammitgliedern erarbeitet und vorgestellt. Folgende Vorträge sind auf der Fahrt gehalten und diskutiert worden:

Evolutionäre Algorithmen: Grundlagen und Konzepte
Als Grundlage für die Betrachtung erweiterter Modelle der evolutionären Algorithmen, wie dem Räuber-Beute-Modell, müssen zuerst die grundlegenden Komponenten und Verfahren betrachtet werden. Dazu gehören die Verfahren der evolutionären Programmierung, der genetischen Algorithmen, der Evolutionsstrategien und der genetischen Programmierung. Für das Projekt sind vor allem die Konzepte der Evolutionsstrategien und der generischen Algorithmen interessant, da diese speziell für die Optimierung entworfen wurden.

Mehrkriterielle Optimierung: Grundlagen und Verfahren
Im Rahmen der Projektgruppe spielt mehrkriterielle Optimierung eine wesentliche Rolle - sind es doch eben diese Probleme, die mit dem Räuber-Beute-Modell angegangen werden sollen. Mehrkriterielle Probleme haben keine eindeutige optimale Lösung, sondern vielmehr eine Menge von Lösungsalternativen. Von einem Verfahren zur Lösung mehrkriterieller Optimierungsprobleme wird erwartet, dass es eine Approximation der Paretofront liefert und dabei sowohl Konvergenz als auch Diversität sicherstellt. Es sind Metriken entwickelt worden, die helfen, diese Anforderungen quantitativ zu erfassen.

Das Räuber-Beute-Modell
Um Optimierungsprobleme mit mehreren Kriterien zu lösen, wird oft auf Lösungsansätze der evolutionären Algorithmen zurückgegriffen. Einer dieser Ansätze wird durch das Räuber-Beute-Modell beschrieben. In der Biologie besteht eine Wechselwirkung zwischen Räubern (welche die Ziele verkörpern) und ihrer Beute (welche die Lösungsmöglichkeiten darstellen). Die Abbildung der Natur auf das Optimierungsproblem wurde schon in einigen wissenschaftlichen Arbeiten beschrieben. Das genaue Verhalten des Systems ist aber noch nicht erforscht.

Visualisierung von Ergebnissen und Metriken zur Bewertung
Das Ziel der Visualisierung von Ergebnissen ist es eine Kommunikationsgrundlage für die Präsentation und Analyse der Datenmenge zu schaffen. Die fertigen Darstellungen müssen dafür effektiv, wirtschaftlich und expressiv gestaltet werden. Außerdem ist es wichtig, das Bearbeitungsziel und die Art der Bewertung der Variablen zu kennen. Die einfachste Möglichkeit Variablen zu visualisieren, ist sie auf einfache visuelle Komponenten abzubilden. Diese werden schließlich zu mächtigeren Visualisierungstechniken kombiniert. In der mehrkriteriellen Optimierung wird speziell auf Multiparameter-Daten gearbeitet. Visualisierungstechniken für solche Daten sind Streckenzüge und ikonenbasierte Techniken.

Werkzeuge zur Visualisierung
Bei der Visualisierung von Räuber-Beute-Modellen können verschiedene Werkzeuge eingesetzt werden, die sowohl das direkte Visualisieren, als auch die statistische Auswertung vereinfachen. Hierzu zählt unter anderem OpenGL. Durch den Einsatz dieser Softwareschnittstelle kann unter anderem die Beute-Population und deren Veränderung visualisiert werden. Programme und Skriptsprachen wie Mathematica, Maple und MatLab können dann bei der statistischen Auswertung behilflich sein. Diese statistische Auswertung kann in Grafiken durch Panelmatrizen, Streckenzüge oder ikonenbasierte Techniken dargestellt werden.

Simulation und parallele Systeme
Unter Simulation wird die Nachahmung der Operationen eines realen Prozesses oder Systems über einer gewissen Zeitdauer verstanden. Ein großer Anteil der Simulationen wird heutzutage auf einem Rechner durchgeführt, diese Art der Simulation wird daher Rechnersimulation genannt. Auch das Verhalten des Räuber-Beute-Modells kann rechnergestützt simuliert werden, um einen Einblick in die Systemdynamik und das Verhalten zu erlangen.

Konzepte paralleler Rechensysteme
Da das Framework parallelisiert werden soll, wurden Grundlagen und Konzepte paralleler Rechensysteme vorgestellt. Die Parallelisierung des Frameworks ist zum einen durch den hohen Rechenaufwand der Funktionsauswertungen bei mehrkriteriellen Optimierungsproblemen motiviert, zum

anderen aber auch, da in der Natur Individuen unabhängig und zeitgleich agieren, was in dem Räuber-Beute-Modell ebenfalls dargestellt und simuliert werden soll.
Es gibt verschiedene Modelle paralleler Rechensysteme, zu nennen sind dabei unter anderem die Modelle mit verteiltem Speicher und gemeinsamen Speicher.

Frameworks zur Entwicklung paralleler Anwendungen
Im Bereich der Entwicklung paralleler Anwendungen sind eine Vielzahl von Problemen zu bewältigen, die sich nicht mit der eigentlichen Aufgabe befassen. Daher hat sich die Nutzung von Bibliotheken und Frameworks durchgesetzt. Durch sie hat der Entwickler die Möglichkeit sich auf das Wesentliche der Aufgabe zu beschränken. Dabei sind Modelle für die Kommunikation in verteilten Systemen sowie für verteilte Objekte zu nennen.

Objekt- und Aspektorientierte Programmierung
Es gibt Methoden der objekt- und aspektorientierten Programmierung, die helfen sollen die Softwareentwicklung zu verbessern. Dabei werden alle eigenständigen Anforderungen der Anwendung als Aspekt bezeichnet. Diese Aspekte sollen getrennt entwickelt, gestestet und zum Schluß zusammengefügt werden. Die aspektorientierte Programmierung erweitert die klassische objektorientierte Programmierung dahingehend, dass die Aspekte gekapselt und modulübergreifend definiert werden können. Das geschieht zum Beispiel durch Concerns, Crosscutting Concerns, Joinpoints, usw.

Softwarequalitätsmanagement
Die Motivation qualitativ hochwertige Software zu entwickeln existiert nicht erst nach Vorfällen wie dem der Ariane 5, deren Absturz auf einen Pufferüberlauf also auf einen Softwarefehler zurückgeführt werden kann. Dies führt zu der Frage, wie hohe Qualität in Software erreicht werden kann. In der Tat besteht bis heute keine Möglichkeit mittels Tools oder Vorgehensweisen sicherzustellen, dass Software frei von Fehlern ist. Allerdings wird versucht durch die Organisation und die Strukturierung des Entwicklungsprozesses die bestmöglichen Rahmenbedingungen zu schaffen, was automatisch zu einer Verringerung von Fehlern führt und sich somit positiv auf die Qualität auswirkt.

Projektmanagement
Die Erfolgsquote bei Projekten ist trotz vielfältiger zur Verfügung stehender Methodenund Hilfsmittel besonders im Softwarebereich nicht groß. Untersuchungen hierzu stellen immer wieder Fehler im Projektmanagement als Auslöser fest. Daher ist es von Bedeutung Hilfsmittel zur Strukturierung und Planung der Arbeit einzusetzen. Die Planung des Projektes, vor allem die Zeitplanung, ist daher sehr wichtig. Diese kann durch verschiedene Tools unterstützt werden, so können Ablaufpläne durch Netzpläne und Gantt-Diagramme visualisiert werden.

Softwareentwicklung und -entwurf in kleinen Teams
Extreme Programming ist eine agile Methode zum Entwurf und zur Erstellung von Software und soll in kleinen Teams angewandt werden. Kerngedanke ist der Fokus auf das Programmieren in Paaren und die Einhaltung der Regel, dass keine Methode vor ihrer Testklasse implementiert wird. Extreme Programming fördert sowohl die interne Kommunikation als auch die externe mit dem Kunden. Dieser stellt während des gesamten Prozesses der Softwareerstellung als ein Teil des Teams Feedback bereit.

3.1.2 Phase 2 - Sammeln und Testen der Designvorschläge

Die zweite Phase umfasst neben der Festlegung der Organisationsstruktur die Einarbeitung in das Thema. Auf Grund mangelnder Erfahrung im Bereich der Entwicklung eines Frameworks wurden hier Designkonzepte und Ideen gesammelt und in kleineren Prototypen gestestet. Einige dieser Designkonzepte sind auch in der endgültigen Version des Frameworks übernommen worden, andere wurden stark überarbeitet, da die Prototypen deutliche schwächen der Konzepte aufgezeigt haben. In der Beschreibung des endgültigen Designs wird auf den Entwicklungs und- Designprozess näher eingegangen. Die entwickelten Prototypen sollen hier aber nicht detalliert beschrieben werden.

3.1.3 Phase 3 - Design und Implementierung des Frameworks

Die dritte Phase beinhaltet das Design und die Implementierung der Release-Version des Frameworks. Dazu gehört die Festlegung der Schnittstellen und deren Dokumentation. Nachdem das Zusammenspiel und die Abhängigkeiten der Schnittstellen geklärt waren, sind die Schnittstellen und Komponenten implementiert worden. Viele Möglichkeiten, die das Framework noch bietet und zulässt sind nicht implementiert worden, können aber durch den Benutzer des Frameworks hinzugefügt werden.

3.1.4 Phase 4 - Testen des Frameworks und Analyse der Ergebnisse

Um diverse Testprobleme zu berechnen und das Verhalten des Frameworks sowie die Ergebnisse zu analysieren, wurden in dieser Phase auch noch weitere Komponenten für das Framework implementiert. Die Schnittstellen und Komponenten sind durch Unittests verifiziert worden, so das sichergestellt ist, dass die Komponenten keine fehlerhaften Ergebnisse berechnen.
Des Weiteren wurden die Ergebnisse der Testfunktionen analysiert. Zum einen wurde die Paretofront und Paretomenge berechnet und geplottet, zum anderen wurde die Bewegung der Räuber beobachtet. Mit Hilfe von Metriken kann ebenfalls die Diversität und die Entwicklung der Population beobachtet werden. Im Kapitel **??** sind die Beobachtungen und Ergebnisse beschrieben.

3.2 Organisation der Teamarbeit

Zur Abstimmung der Arbeiten im Team gibt es wöchentliche treffen. Dabei dient ein Treffen zur Abstimmung und Verteilung der nächsten Aufgaben, sowie zur Vorstellung erarbeiteter Ergebnisse. Die weiteren Treffen dienen zum gemeinsamen Arbeiten. Durch das Designen und Implementieren in einem Raum werden Kommunikationswege einfacher. So kann bei Problemen und Fragen direkt mit den anderen gesprochen werden. Auch neue Aufgaben, die erst während der Implementierung auftreten, können direkt verteilt werden.
Diese Treffen sollen so effizient wie möglich sein, daher muss zumindest einer diese vorbereiten und die Aufgaben, die noch anstehen, im Blick haben. Die Organisation der Teamarbeit hat sich erst mit der Zeit entwickelt und kann nicht exakt mit einer standadisierten Organisationsform beschrieben werden. Zu Beginn wurden diverse Organisationsformen ausprobiert, wobei für die folgenden Phasen keine Organisationsform direkt übernommen wurde, sondern jeweils die Teile, die als gut empfunden wurden.
Von der anfänglichen Organisationsform, in der alle Teammitglieder alles gemeinsam gemacht haben, ist schnell Abstand genommen worden. Durch diese Form ist die Arbeit sehr unproduktiv gewesen und durch viele Diskussionen gebremst worden.
Die zweite Organisationsstruktur ist unter SCRUM bekannt. Ein Teammitglied wird zum Scrummaster ernannt und hat sich zusätzlich zu den Teamaufgaben um die Organisation und die Zeitplanung zu kümmern. Nachdem alle Aufgaben definiert worden sind, werden Prioritäten verteilt und die Aufgaben in der dadurch vorgegebenen Reihenfolge abgearbeitet. Es wird zu Beginn jeder Phase festgelegt, welche Aufgaben im nächsten Zeitabschnitt erledigt werden und wer im Team für die Aufgabe verantwortlich ist. Die Deadline wird nie verschoben, es kann bei Verzögerungen lediglich passieren, dass Aufgaben aus der Liste entfernt werden, oder eventuell weitere hinzu genommen werden.
Diese Organisationsstruktur wurde im Laufe der Projektgruppe so modifiziert, dass sie den Anforderungen des Teams entsprach. Es gibt ein Teammitglied, das sich um den zeitlichen Ablauf, Deadlines und die Aufgabenverteilung kümmert. Dieses deligiert auch weitere größere Aufgabenbereiche an andere Teammitglieder, die sich dann nur um den Fortschritt dieses Themenbereichs bemühen. So wurden zum Beispiel der Zwischenbericht und der Endbericht aus der Organisationsaufgabe des „Masters“ genommen und an ein anderes Teammitglied deligiert. Die anstehenden Aufgaben werden nicht mehr als „Productbacklog“ und „Sprintbacklog“ festgehalten. Auch der „Daily Scrum“ ist entfallen. Es wird lediglich noch ein Statusbericht beim Meeting gegeben. Da die

Aufgabenbereiche recht überschaulich waren und die Deadlines ebenfalls, wäre das beibehalten der Organisationsstruktur wie bei SCRUM vorgesehen, ein zu hoher Aufwand gewesen.

4 Design des Frameworks

Im folgenden Kapitel wird das Design des Frameworks genauer erklärt. Hierbei werden insbesondere Strukturen und Konzepte erläutert und die vom Framework bereitgestellten Funktionen vorgestellt. In diesem Zusammenhang werden die schon implementierten Testfunktionen beschrieben.

4.1 Struktur und Modellierung

Bei der Modellierung des Frameworks wurde darauf geachtet, die aus dem abstrakten Räuber-Beute-Modellbekannten Komponenten in genau ein entsprechendes Interface beziehungsweise in genau eine entsprechede Klasse umzusetzen. Diese Interfaces und Klassen, sowie Interfaces und Klassen, die ein Mindestmaß an Zusammenspiel regeln, bilden das API-Paket. Konkrete, abgeschlossene Implementierungen finden sich im IMPL-Paket, während das Server-Paket, die Klassen und XML-Dateien beinhaltet, die Testläufe anstossen und spezifizieren. Schließlich gibt es noch zwei Pakete, die als Stützen bei der Erzeugung (ConfigUtil) und Implementierungen (Util) dienen.

4.1.1 Das API-Paket

Die Basis des Frameworks bilden die Individuen (Individual), die Operatoren in einer Welt bestehend aus miteinander verbundenen Positionen (Position) ausführen. Die Welt (World) ist dabei ein Singleton und hat Zugriff sowohl auf die aktiven Komponenten (Individuen, Umgebungen) als auch die Positionen. Die Individuen werden nicht durch Klassentyp in Räuber und Beute unterteilt, sondern lediglich durch ihr Verhalten und ihre Attribute. Das Verhalten wird durch eine eigene Klasse (Behaviour) repräsentiert, die jeweilige als nächste durchzuführende Aktion (Action) besimmt. Die drei klassischen Operatoren des Räuber-Beute-ModellBewegung (Movement), Selektion (Consumption) und Recombination (Reproduction) sind dabei spezielle Aktionen. Die Repräsentation des Problems erfolgt mit Hilfe der Chromosomen (Chromsome) der Individuen und der Modellierung der Fitnessfunktionen (TestProblem und FitnessFunction). Ferner gibt es ein optionales Speziesattribut (Species) von Individuen. Die einzige Klasse, die weder arbeitsablauforganisierende Funktion haben noch einer Komponente des klassischen Räuber-Beute-Modells sind, sind die Umgebungen (Environments), die Positionen zusammenfassen und Operationen auf diesen und darauf befindlichen Individuen ausführen.

4.2 Benutzung des Frameworks

Das bereits beschriebene API-Paket umfasst den Kern des Framworks und enthält alle Schnittstellen. Somit definiert die API das komplette Zusammenspiel aller Klassen. Das Implementieren sämtlicher Schnittstellen ist jedoch sehr aufwändig und nur in den wenigsten Fällen notwendig. Vielmehr möchte der Benutzer anhand eines Beispiels in das Framework eingeführt werden. So kann das System verstanden und an den gewünschten Stellen angepasst werden. Damit dieses Vorgehen möglich ist sind die Pakete IMPL und SERVER vorhanden.

Das IMPL-Paket stellt dem Benutzer Implementierungen der verschiedenen API-Komponenten zur Verfügung. Somit kann der Benutzer gleich zu Anfang aus verschiedenen Bausteinen für sein eigenes Modell auswählen und ist nicht gezwungen übliche Komponenten, wie z.B. eine Bewegungsfunktion für einen Räuber, selbst zu implementieren.

Das SERVER-Paket hingegen bietet eine zusätzliche Funktion. Es umfasst eine Implementierung der zentralen Schnittstelle *PPOModell* und ermöglicht so dem Benutzer sein Modell mit Hilfe von

XML-Konfigurationen zusammenzustellen. Daher müssen nicht sämtliche Komponenten des Frameworks von Hand erzeugt werden. Vielmehr erwartet das Programm lediglich die Angabe von Konfigurationsdateien. So wird dem Anwender nicht nur Programmierarbeit erspart. Es wird auch wesentlich einfacher eigenen Komponenten in das Framework einzubringen.

Mit Hilfe der XML-Konfigurationen (SERVER-Paket) und den Implementierungen (IMPL-Paket) werden dem Benutzer komplette Modelle für einige allgemein bekannte Probleme der mehrkriteriellen Optimierung zur Verfügung zu stellen. Folgende solcher Standardprobleme sind bereits vorimplementiert:

Problem:	**Konfigurationsdatei:**
Multi-Sphere	multiSphere.xml
Multi-Sphere	multiSphereWithEnvironment.xml
Kursawe	kursawe.xml
P*	pStar.xml

(Für das Problem Multisphere bestehen zwei Konfigurationsdateien. Die zweite Konfiguration veranschaulicht die Benutzung von Environments.)

Die Konfiguration des SERVER-Paketes muss mit Hilfe von zwei Dateien vorgenommen werden. Zum einen muss das eigentliche Räuber-Beute-Modell beschrieben werden und zum anderen werden die Systemeinstellungen des Frameworks festgelegt.
Das Modell enthält die eigentliche Problembeschreibung des Räuber-Beute-Systems, das man erstellen will. Hier wird die Umgebung (im klassischen Fall ein Torus) und ihre Größe definiert. Außerdem muss angegeben werden, welche Art und wieviele Beute- und Räuberindividuen in dem Modell enthalten sein sollen. Dies umfasst z.B. auch das Jagd- und Bewegungsverhalten der Räuber oder die Gentypen der Beute.
Die Systemeinstellungen beinhalten alle Paramter für das Framework. Insbesondere muss hier ein Stopkriterium angegeben werden, durch das die Terminierung des Systems gewährleistet wird. Zusätzlich muss an dieser Stelle angegeben werden, welche Ausgabe- und Metrikstrategien verwendet werden sollen. Metriken sind Ausgaben, die während der Laufzeit dargestellt werden. Ausgabestrategien geben an, auf welche Weise die Ergebnisse des Frameworks gespeichert werden sollen. In beiden Fällen geht es also darum die Ergebnisse geeignet anzugeben.

Durch die Veränderung der Konfigurationsdateien wird natürlich kein neues Verhalten in das Framework integriert. Um eine neue Funktionalität hinzuzufügen, muss eine neue Javaklasse erstellt werden, die eines der Interfaces des API-Paketes implementiert. Wenn aber eine solche neue Klasse in das Framework eingebunden wurde, kann diese Klasse über die Konfiguration angesprochen werden.
Eine genauere Beschreibung, wie das Framework mit Eclipse erweitert werden kann, befindet sich im Anhang. C

4.3 Testfunktionen

Da multikriterielle Probleme in der Praxis sehr komplex und die Paretofront und Paretomenge nicht exakt berechenbar sind, sind Testsuiten und Testfunktionen entwickelt worden, um Algorithmen für multikriterielle Probleme auf ihre Funktion und Güte zu testen. Durch die Testfunktionen ist es aber auch möglich Algorithmen und Parametereinstellungen eines Algorithmus zu vergleichen.
Die Testfunktionen sind so konzipiert, dass die Paretomenge und Paretofront berechenbar sind oder zumindest das Aussehen der Funktion bekannt ist. Es gibt unterschiedliche Testsuiten in welchen Funktionen ähnlichen Schwierigkeitsgrades und mit ähnlichem Aussehen zusammengefasst sind. Jede Testfunktion hebt unterschiedliche mathematische Eigenschaften hervor. Einige bestehen zum Beispiel nur aus konvexen, andere nur aus konkaven Teilen, die etwas komplexeren Funktionen

haben sowohl konkave als auch konvexe Teile. In der Praxis haben die multikriteriellen Probleme mehr als eine mathematische Eigenschaft, mit der der Algorithmus umgehen können muss, manche sind auch nur mit schwierigen Simulationen zu berechnen. Die Testfunktionen beschränken sich, je nach Schwierigkeitsgrad, auf wenige mathematische Eigenschaften, die bekannt sind.
Im Folgenden werden die implementierten Testfunktionen Multi-Sphere, Kursawe, P^* und Binh vorgestellt.
Zu den etwas künstlichen Testfunktionen sind ebenfalls Testfunktionen implementiert worden, die aus der Praxis motiviert sind. Zu sochen Problemen gehört das Scheduling. Im Folgenden wird ebenfalls eine Variante vorgestellt, die das Scheduling-Problem mehrkriteriell löst.

4.3.1 Multi-Sphere Testproblem

Die Multi-Sphere-Testfunktion ist eine konvexes Testproblem und wird für zwei Kriterien durch folgende Formel beschrieben.

$$\begin{aligned} f_1(\vec{x}) &= x_1^2 + x_2^2 \\ f_2(\vec{x}) &= (x_1 - 2)^2 + x_2^2 \qquad x_1, x_2 \in [-10; 10]^2 \in \mathbb{R}^2 \end{aligned} \tag{1}$$

Konvex bedeutet, dass für die Funktion in einem bestimmten Wertebereich $f''(x) \geq 0$ gilt.
Diese Testfunktion ist eine der einfachsten Testfunktionen für multikriterielle Probleme. Die Paretofront besteht aus einem konvexen Teil und ist zusammenhängend. Die Werte für beide Funktionen (f_1 und f_2) liegen im Intervall $[0, 4]^2$. Die Paretomenge, also Werte, die die Variable x_1 annimmt, liegen zwischen 0 und 2. Die Variable x_2 ist immer 0. Die Paretomenge und Paretofront sind in Abbildung 3 zu sehen.

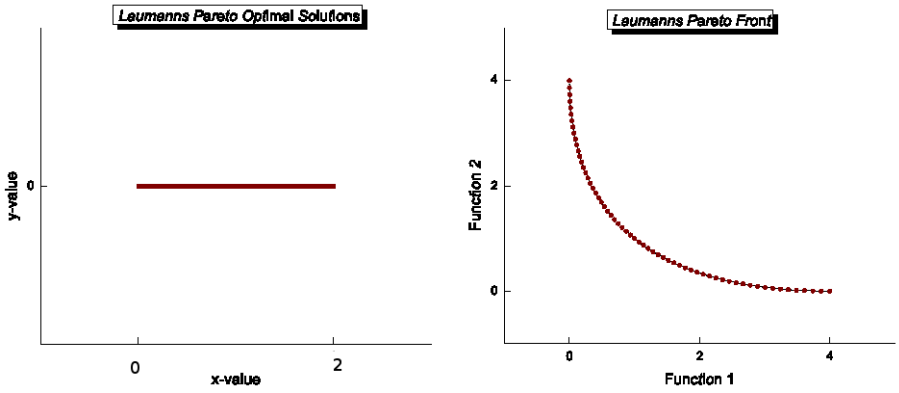

Abbildung 3: Paretomenge (links) und Paretofront (rechts) des Multi-Sphere Testproblem

4.3.2 Kursawe Testproblem

Das Kursawe Testproblem ist wesentlich schwieriger zu lösen als das Multi-Sphere Testproblem. Zum einen liegt es daran, dass das Testproblem sowohl aus konkaven als auch konvexen Teilen besteht, zum anderen, da weder Paretomenge noch Paretofront zusammenhängend sind (siehe Abbildung 4).
Die Schwierigkeit in der Approximation der Paretomenge liegt darin, dass diese aus zwei nicht zusammenhängenden Teilen besteht, wobei ein Teil konvex und ein Teil konkav ist. Je nach verwendeter Rekombination und Mutation wird entweder der konkave oder der konvexe Teil besser gefunden. Es stellt sich als besonders schwierig heraus, die Einstellungen der Parameter im Räuber-Beute-Modell so vorzunehmen, dass beide Teile der Paretomenge und Paretofront gleichermaßen gut approximiert werden.

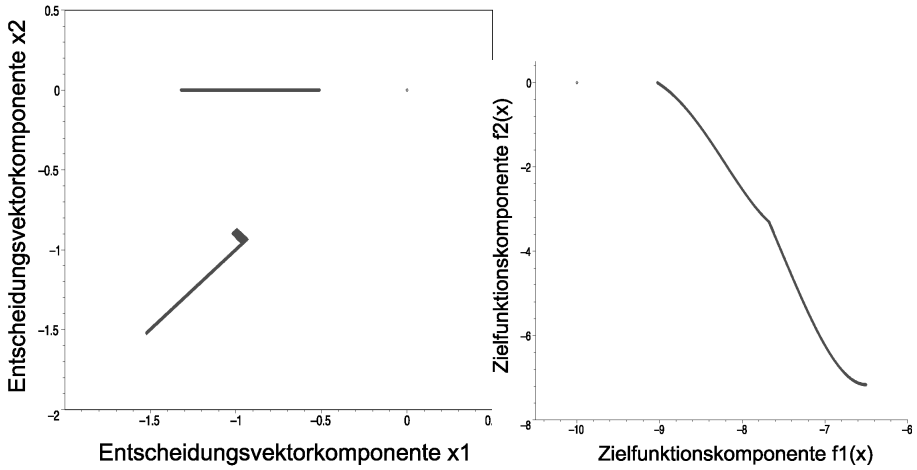

Abbildung 4: Paretomenge (links) und Paretofront (rechts) des Kursawe-Testproblems für den zweidimensionalen Bereich

Das Kursawe Testproblem ist durch folgende Gleichungen mathematisch beschrieben.

$$
\begin{aligned}
f_1(\vec{x}) &= \sum_{i=1}^{n-1}(-10 \cdot \exp(-0.2 \cdot \sqrt{x_i^2 + x_{i+1}^2})) \\
f_2(\vec{x}) &= \sum_{i=1}^{n}(\mid x_i \mid^a + 5\sin(x_i^b)) \\
\text{Wobei } \vec{x} &= \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{R}^n, a = 0.6 \text{ und } b = 3 \text{ gelten.}
\end{aligned}
$$

4.3.3 P^* Testproblem

Das Testproblem P^* approximiert die konvexe Hülle der Punkte, die durch die Menge der Zielkriterien gegeben sind. Der Schwierigkeitsgrad dieses Testproblems ist ähnlich dem der Multi-Sphere Testproblem, hier können aber einfacher Kriterien hinzugefügt werden, da sich die Anzahl der Kriterien nicht auf die Dimension auswirkt. In diesem Problem wird die Dimension nur über die Dimension der Vektoren beeinflusst, nicht über die Anzahl der Kriterien.
Die Population repräsentiert die Punkte im Vektorraum, wobei die Gene der Individuen die Koordinaten speichern und der Fitnesswert der Individuen die euklidische Distanz des Punktes zu dem Punkt des Zielkriteriums. Je weiter der Punkt von dem Punkt des Zielkriteriums entfernt ist, desto schlechter ist der Fitnesswert.

Das Testproblem P^* wird durch folgende Formel berechnet:

$$\begin{aligned} f(\vec{x},\vec{y}) &= \mid \vec{x}-\vec{y} \mid = \sqrt{(\vec{x}-\vec{y})\cdot(\vec{x}-\vec{y})} \\ \text{Wobei } \vec{x} &= \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{R}^n(\ Individuum) \\ \text{Und } \vec{y} &= \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \in \mathbb{R}^n(\ Zielkriterium) \end{aligned}$$

4.3.4 Binh Testproblem

Das Binh-Testproblem wurde 1996 von T. Binh und U. Korn publiziert und besteht aus zwei Gleichungen, die zeitgleich minimiert werden. Das Optimum der einen Gleichung liegt im Punkt $(0,0)$, das Optimum der zweiten Gleichung im Punkt $(5,5)$. Die Pareto-optimalen Lösungen sind gut zu analysieren und in Abbildung **??** zu sehen. Das Testproblem ist ein Spezialfall des allgemeinen Multi-Sphere Testproblems.
Im Folgenden sind die zwei Gleichungen und die Intervalle der Variablen aufgelistet:

$$\begin{aligned} f_1(x,y) &= x^2+y^2 \\ f_2(x,y) &= (x-5)^2+(y-5)^2 \\ \text{Wobei} &\quad -5 \leq x,y \leq 10 \end{aligned} \tag{2}$$

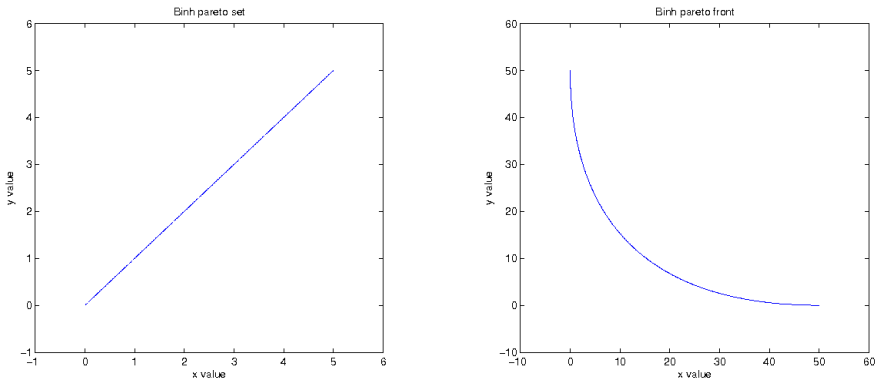

Abbildung 5: Paretomenge (links) und Paretofront (rechts) des Binh-Testproblems

4.3.5 Mehrkriterielle Schedulingprobleme

Die meisten Schedulingalgorithmen beschräen sich darauf, einen Auftragsplan so zu erstellen, dass dieser bezüglich eines Kriteriums (Auslastung der Maschinen, Durchsatz der Jobs) optimal ist. Jedoch lassen sich auch mehrere Ziele in einem Schedule verfolgen. Im Folgenden wird ein solches Problem vorgestellt. In Kapitel 5.6 werden dann Lösungen, die durch das Framework ermittelt wurden, präsentiert.

Minimierung der Total Weighted Completion Time (TWCT) und der Maximum Lateness (ML) eines Schedules

Sowohl das TWCT Problem als auch das ML Problem lassen sich offline auf einer Maschine in polynomieller Zeit optimal lösen. Im allgemeinen Fall ist das entsprechende bikriterielle Problem

jedoch NP-schwer. In diesem speziellen Modell (das aber immer noch NP-schwer ist) wird vereinfacht davon ausgegangen, dass keine Bedingungen an die Bearbeitungsreihenfolge gestellt werden, also kein Job soll erst dann in die Maschine eingegeben werden können, wenn ein anderer bereits bearbeitet wurde. Für jeden Job $j \in J$ ist die Bearbeitungszeit (processing time) p_i, das Duedate d_i und die Kosten (penalty) für eine verzögerte Fertigstellung w_i bekannt. Die Reihenfolge, mit der die Jobs in die Maschine eingeben werden ist ein Schedule. Für jeden Schedule S gelte im Folgenden:

1. Der Zeitpunkt der Fertigstellung (completion time) eines Jobs sei $C_{s,j}$.
2. Die Verspätung $l_{s,j}$ eines Jobs j sei
   - 0, wenn $C_{S,j} \leq d_i$
   - $C_{s,i} - d_i$, sonst

Mit diesen Definition lassen sich TWCT und ML auf einem Schedule s wie folgt beschreiben:

1. $twct(S) = \sum_{j \in J} C_{S,j} \cdot w_i$
2. $ml(S) = \sum_{j \in J} l_{S,j}$

Beispiel

Gegeben sind 3 Jobs mit Parametern die der Tabelle zu entnehmen sind.

Job	Processing Time	Deadline	Penalty
j_1	1	12	10
j_2	3	6	5
j_3	5	5	1

Mit earliest-deadline-first (EDF) lässt sich ein Schedule ermitteln, der die geringste maximum lateness hat, in diesem Fall also die Bearbeitungsreihenfolge (j_3, j_2, j_1) mit $ML((j_3, j_2, j_1)) = 2$. Ein für TWCT optimaler Schedule lässt sich anlog ermitteln, indem man den Job mit der größten penalty zuerst bearbeitet (**G**reatest **Penalty F**irst). Dieser Ansatz liefert die Auftragsreihenfolge $(j_1, j2, j_3)$. Insgesamt sind in diesem gewählten Beispiel alle möglichen Auftragsreihenfolgen pareto-optimal bezüglich des bikriteriellen Problems.

Methodik

In diesem Abschnitt werden die allgemeinen Operatoren vorgestellt, die zur Simulation des weiter oben vorgestellten mehrkriteriellen Schedulingproblem verwendet wurden. Die Komponenten und Operatoren, deren Einsatz eine unmittelbare Auswirkung auf das Ergebnis eines Simulationslaufs hatten, werden jedoch später unter dem Abschnitt Ergebnisse erläutert.
Die Repräsentation eines Schedules lässt sich auch als Permutation über die zu Grunde liegenden Jobs auffassen. Für die konkrete Darstellung einer solchen Permutation gibt es eine Reihe von Variationen, deren Unterschiede aber kaum Auswirkungen auf die Testläufe hatten.
Die Reproduction erfolgte durch

1. Mutation der Chromosome der umliegenden Beute (Vertauschen, Invertierung, randomisiertes Einfüegen),
2. und Anwendung von Crossoveroperatoren für Permutationen auf die Chromosome der umliegenden Beute.

Denkbar waren auch Operatoren, die gezielt Verbesserungen bezüglich eines Kriteriums vorgenommen haben. Diese haben jedoch im Vergleich schlechter Abgeschnitten und werden im Folgenden nicht weiter betrachtet. Auf höherer Ebene wurde das einfache von Laumanns-Modell simuliert.

Die Individuen wurden auf einem zweidimensionalen Torus platziert, wobei nur die Räuber die aktiven Komponenten des Modells ansteuerten. Jeder Räuber verfolgte eines der beiden Kriterium und wählte aus seiner unmittelbaren Nachbarschaft eine Individuum aus, das bezüglich seines Kriteriums das schlechteste war. Unmittelbar nach der Selektion erfolgte auch sogleich die Rekombination.

4.4 Funktionen und Features des Frameworks

Im Folgenden werden einige Funktionen und Features des Frameworks genauer vorgestellt. Im einzelnen sollen hier die Individuen (Räuber und Beute) näher betrachtet werden, sowie die Fitnessauswertung zur Selektion der Beute und das Konzept der Variationsoperatoren zur Erzeugung neuer Beute. Des Weiteren wird die Engine und ihre Funktionsweise erläutert.

4.4.1 Individuen

Individuen sind im von-Laumanns-Modell neben den Positionen der Welt die wesentlichen Elemente des Räuber-Beute-Modells. Das Framework abstrahiert an dieser Stelle die klassische Auffassung von einer passiven Beute, die lediglich einen Wert repräsentiert, und einem aktiven Räuber, der in der Lage ist, sich zu bewegen, zu konsumieren und die Reproduktion startet. Es existiert zunächst eine Oberklasse `Individual`, welche zum einen gemeinsame Eigenschaften von Räubern und Beute zusammenfasst und zum anderen die vom Framework zur Verwaltung eines Individuums notwendigen Elemente liefert. Das Verhalten eines Individuums kann auf zwei unterschiedlichen Wegen realisiert werden, wobei beide Varianten garantieren, dass auch Implementierungen von Systemen, die zum Zeitpunkt des Entwurfs noch nicht ersichtlich waren, realisiert werden können.
Indirekt wird einem Individuum zum Zeitpunkt der Kompilierung über eine vorher definierte XML-Datei das Verhalten (`Behaviour`) im Sinne von Aktivitäten, Ziele (`Objective`) und Chromosome (`Chromosome`) übergeben. Diese Elemente sind nicht zwingend vollständig notwendig. Im klassischen Modell wird ein Räuber ein Bewegungs- und Konsumverhalten, eine Zielfunktion, aber kein Chromosom besitzen, während ein Beuteindividuum ein Chromosom benötigt, um reproduziert werden zu können. Auf diese Weise ist es beispielsweise auf eine einfache Art möglich, ein sich bewegendes Beuteindividuum zu erstellen, da an dieser Stelle eine Zuweisung einer Aktivität ausreicht. Die einzelnen Aktivitäten werden wie Perlen auf einer Kette zusammengeschaltet und ermöglichen so komplexe, aufeinander Abgestimmte Verhaltensweisen.
Speziellere Anpassungen benötigen das direkte Anpassen und damit Ableiten der Klasse. So lassen sich beispielsweise Arten erstellen, bei denen die Individuen einem Alterungsprozess ausgesetzt sind oder Individuen eines gleichen Stammes ihr Verhalten aufeinander abstimmen.
Dabei ist das ebenfalls in der XML-Datei definierte Attribut `species` stets ein Indikator, zu welcher Art ein Individuum gehört. Dieses Attribut wird unter anderem bei der Reproduktion benötigt.
Individuen residieren auf Positionen und interagieren über diese mit der Welt. Die Startposition wird beim Erzeugen eines Individuums gesetzt und sollte zur korrekten Ausführung stets gesetzt bleiben.

4.4.2 Fitnessauswertung

Die Fitnessevaluation in unserem Framework wurde gegenüber dem herkömmlichen Gedanken multikriterieller Berechnungen dahingehend erweitert, dass die Trennung von einem eineindeutigen Mapping von Kriterien zu deren Berechnungen aufgelöst wurde. Dafür wurde der gesamte Berechnungsprozess von dem Algorithmus getrennt betrachtet und eine feste Schnittstelle als Verbindung der beiden Komponenten zur Verfügung gestellt. Somit kann nun eine Konfiguration der Fitnessevaluation komplett unabhängig von der Umgebung oder den verwendeten Individuen und deren Verhalten erfolgen.
Die Klasse `Evaluator` stellt eben diese Schnittstelle dar, die Fitnessberechnungen sammelt, startet und beobachtet. Jeder Räuber und die Welt bekommen bei der Initialisierung eine Instanz dieser

Klasse zugewiesen. Eine Fitnessevaluation wird in dem Framework insbesondere dann angestoßen, wenn ein Räuber für seine Menge an Kriterien, die von diesem optimiert werden sollen, eine Menge an Individuen, Beute in seiner Umgebung, ausgewertet haben möchte. Dafür muss der Räuber in seiner Selektionsphase den Evaluator anstoßen. Der Evaluator ist dabei eine Framework betreibende Klasse, die eine feste Schnittstelle darstellt. Für eine konventionelle Nutzung des Frameworks sollte es nicht mehr nötig sein diese durch eine eigene Implementierung zu erweitern.
Das Optimierungsproblem selbst wird komplett über eine von der abstrakten Klasse `TestProblem` abgeleiteten Klasse definiert. Der Evaluator kennt dabei die Referenz auf das verwendete Problem. Es ist entscheidend, dass nur eine Instanz der Klasse erzeugt wird, da damit, wie weiter unten erwähnt, redundante Berechnungen verhindert und umgangen werden.
In dieser Klasse, die das zu optimierende Problem beschreibt, werden, wenn eine Fitnessevaluation erfolgen soll, Objekte der Klasse `Computation` erzeugt. Diese stellen einzelne Berechnungen von Kriterien dar und stoßen einzelne Berechnungen von Fitnessfunktionswerten an. Sie sind insbesondere aus der Anforderung der Parallelisierbarkeit und der Flexibilität der Berechnungen enstanden. Deshalb ist es möglich sowohl Berechnungen für ein Individuum zu parallelisieren, wie auch dieselbe Berechnung für mehrere Individuen. Damit können sowohl ein einzelner Fitnesswert, wie auch mehrere Fitnesswerte in einer Computation berechnet werden. Außerdem ist es insbesondere einfacher externe Berechnungen, wie etwa Simulationen, durchzuführen und dort mehrere Aspekte einer Simulation als Fitnesswerte in das Framework miteinzubinden. Zum Beispiel wird bei einem multikriteriellen Problem wie Multisphere eine Computation für die Berechnung eines Fitnesswertes verantwortlich sein. Wird dagegen zum Beipiel ein Schedulingalgorithmus simuliert, so kann eine einzige Computation Fitnesswerte für die Laufzeit insgesamt, die Zeit die der Prozessor untätig ist oder die verpassten Deadlines gleichzeitig berechnen.
Das wurde auch durch die Benutzung der Klasse `Objective` erreicht. Durch diese werden die einzelnen Kriterien, die im klassischen Sinne Fitnesswerte darstellen, identifiziert. Mit Hilfe eines Mapping von einer Computation auf eine Menge von Objectives, können dann die verschiedenen Kriterien und ihre Berechnungen miteinander verbunden werden. Solche Objectives werden den Individuen, die Fitnesswerte evaluieren können sollen, im allgemeinen Fall sind das die Räuber, mitgegeben. Die Implementierung des Testproblems garantiert dabei, dass eine effiziente Menge an Berechnungen für einen nachfragenden Räuber erzeugt wird. Der Evaluator stößt dann für seinen Räuber die nötigen Berechnungen an. Dabei wird in der Computation Klasse garantiert, dass alle berechneten Fitnesswerte in der Beute gespeichert werden, so dass einmal berechnete Werte für alle Räuber zur Verfügung stehen.
Um Berechnungen auf dem Cluster ausführen zu können, gibt es die abstrakte Klasse `DRMAAComputation`, die von der üblichen Computation erbt. Diese wird in dem Kapitel über die Parallelisierung des Frameworks noch genauer erläutert.

4.4.3 Variationsoperatoren

Variationsoperatoren sind Teil einer Reproduktion und arbeiten auf Mengen von Individuen. Im Rahmen dieser Aktionen könnnen Variationsoperatoren zum Beispiel für Mutation oder Rekombination benutzt werden, doch auch andere Reproduktionsformen lassen sich darauf abbilden.
Das Konzept der Variationsoperatoren unterstützt den Aufbau des Modells aus leicht austauschbaren Teilen. Es erlaubt eine Trennung von Rekombinations- beziehungsweise Mutationsverfahren und der Aktion, wodurch ein Austausch der Verfahren bei Experimenten deutlich erleichtert wird. Der Ansatz dazu entstand aus der Beobachtung, dass die einzelnen Teilschritte einer Reproduktion in der Regel weitgehend identisch sind und sich nur in kleinen Punkten unterscheiden. Ein Großteil der Unterschiede fiel in die Auswahl der Eltern einer Rekombination und die Berechnung variierter Werte eines Nachkommens. An dieser Stelle setzen ein oder mehrere Variationsoperatoren an, um diese Details der Reproduktion zu füllen. Die Operatoren werden hierbei als Liste nacheinander abgearbeitet, an deren Ende üblicherweise ein oder mehrere neu erzeugte Individuen anfallen. Ein Nutzer hat so die Wahl zwischen verschiedenen Variationsoperatoren und kann mit geringem Pro-

grammieraufwand direkt auf das Ergebnis Einfluss nehmen.

Als unterstützende Klassen existieren im Framework zum einen die abstrakte `VariationOperatorReproduction`, welche einen Rahmen für eine Reproduktion mit Variationsoperatoren zur Verfügung stellt und leicht für einen konkreten Anwendungsfall zu erweitern ist, und zum anderen fertig implementierte Reproduktionsklassen wie `AutonomousReproduction` und `ControlledReproduction`, die direkt verwendet oder zuvor noch über eine Reihe von Parametern beeinflusst werden können.

Je nach Umfang der erwünschten Unterschiede zwischen Variationsoperatoren ist es auch möglich, bestehende Implementierungen von Variationsoperatoren wie `CreatingMutationVO`, `ModifyingMutationVO`, `RecombinationOfBestVO` oder `SpeciesRecombinationVO` zu verwenden, für die zu diesem Zweck die Interfaces `ChromosomeMutation` und `ChromosomeRecombination` entwickelt wurden. Der Zweck dieser Interfaces ist erneut, das Verhalten aus modularen Teilen zusammenstellen zu können. Für einige der in Testfunktionen verwendeten Chromosomenstrukturen existieren bereits direkt verwendbare Versionen einer Mutation oder Rekombination, die sich in einen der vier genannten Variationsoperatoren einfuegen und sofort nutzen lassen. Andere Chromosomen oder andere Mutations- und Rekombinationsansätze lassen sich mit minimalem Aufwand umsetzen, wenn sie in Verbindung mit den zur Verfügung gestellten Variationsoperatoren den Vorstellungen entsprechen.

Der Zusammenhang zwischen den einzelnen Komponenten ist in der folgenden Grafik anhand eines Beispiels veranschaulicht. Es stellt eine Reproduktion dar, bei der jedes Individuum eigenständig Nachkommen in seiner Umgebung erzeugen soll, sofern freie Stellen existieren.

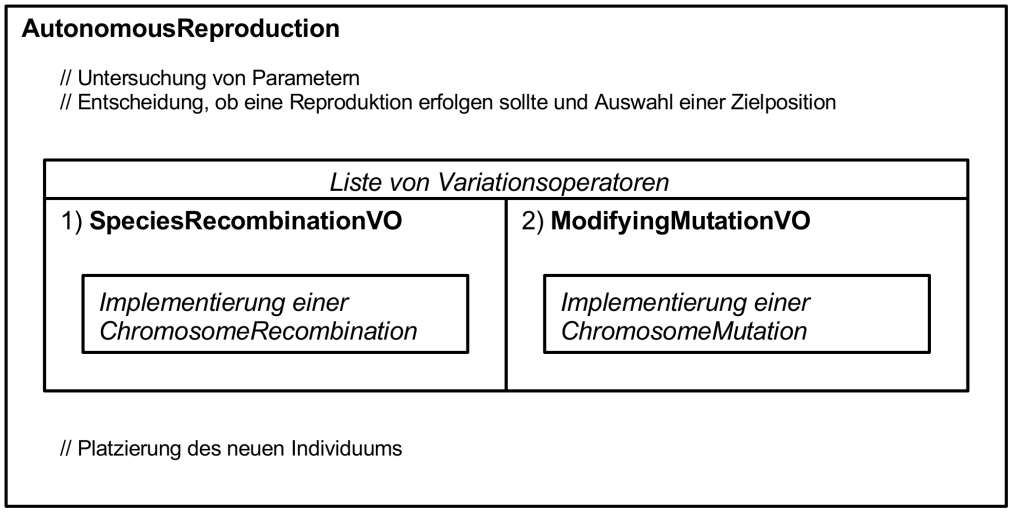

Abbildung 6: Beispiel für eine Reproduktion mit Variationsoperatoren

Hierbei bietet die Klasse `AutonomousReproduction` einen Rahmen, dessen Liste von Variationsoperatoren um zwei für die beabsichtigte Rekombination und Mutation geeignete ergänzt wird. Im Idealfall sind so lediglich noch die an die speziellen Chromosomen angepassten Mutations- und Rekombinationsteile zu implementieren.

4.4.4 Engine

Hinter dem Konzept der Engine verbirgt sich die Möglichkeit Aktionen, die Veränderungen an den Daten des Räuber-Beute-Modell vornehmen, unabhängig auszuführen. Dabei ist es unwichtig, ob

die Aktion eine Handlung eines Individuums oder einen Umwelteinfluß repräsentiert. Die Hauptaufgabe der Engine besteht darin einen Arbeitsthread von dem schon existierenden Steuerthread abzuspalten. Der Arbeitsthread führt eine zuvor definierte Strategie aus, die beschreibt in welcher Reihenfolge und in welcher Art und Weise die Aktionen angestoßen werden. Dabei ist es möglich neben den Grundfunktionen, die den Arbeitsthread starten, stoppen, pausieren bzw. fortsetzen können, beliebig viele Stop-Kriterien zu definieren, die dafür sorgen, dass sobald ein Stop-Kriterium erfüllt ist der Arbeitsthread angehalten wird. Diese Funktionalität wird durch die Engine bereitgestellt und durch den Steuerthread umgesetzt. Da der Steuerthread, nachdem er einen Befehl an den

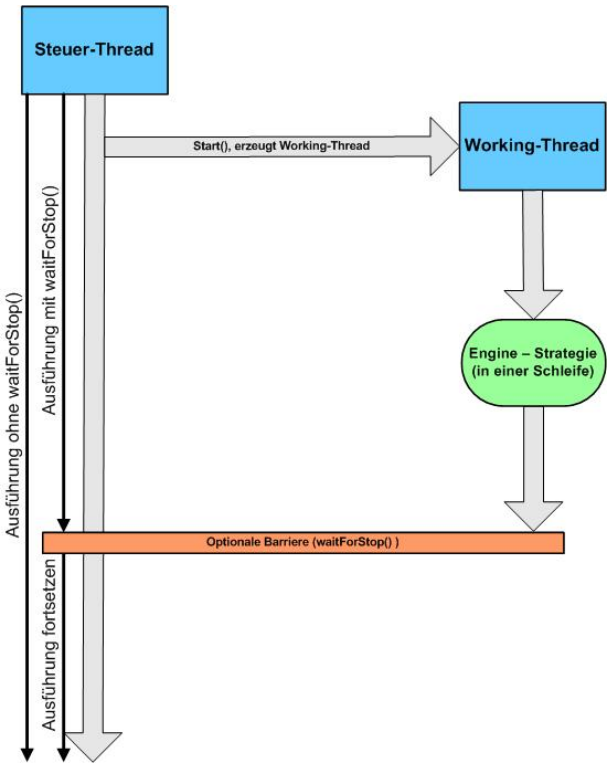

Abbildung 7: Dummy: Funktionsweise der Barriere waitForStop()

Arbeitsthread abgeschickt hat, normalerweise unabhängig vom Zustand des Arbeitsthread weiterläuft, existiert eine Funktion die den Steuerthread zwingt zu pausieren solange der Arbeitsthread noch nicht beendet wurde. So kann man nun zusätzlich sicherstellen, dass Konfigurationen, die erst durch ausgeführte Aktionen zur Laufzeit geschaffen werden, auch wirklich erreicht werden bevor der Steuerthread darauf aufbaut.

4.5 Parallelisierung des Frameworks

Um eine effizientere Nutzung des Räuber-Beute-Modells zu ermöglichen, soll ein Teil des Programms parallelisiert werden. Hierzu sollen die komplexeren Berechnungen auf unterschiedlichen Knoten des IRF-Clusters parallel berechnet werden.
Da der rechenaufwendigste Teil des Programms die Berechnung der Fitnessfunktion darstellt, soll diese parallel auf den verschiedenen Knoten des Clusters ausgewertet werden.
Das IRF-Cluster ist ein Portable-Batch-System von Torque. Somit verwaltet und scheduled das Cluster seine Jobs selber. Das Ansprechen des Clusters aus einem Java-Programm stellt eine der Probleme beim Parallelisieren dar, so dass unterschiedliche Konzepte beim Lösen dieser Problematik verfolgt wurden.

4.5.1 Terracotta

Terracotta ist ein Open-Source-Clustering-Projekt für Java. Es schiebt sich zwischen die Java Virtual Machine und die eigentliche Applikation. Dabei muss die Applikation nicht explizit für eine Parallelisierung angepasst werden, denn Terracotta übernimmt die Synchronisation und das Zwischenspeichern der Objekte. Aus Sicht der Applikation sind alle Objekte immer lokal vorhanden und Terracotta verwaltet unabhängig das Abgleichen der Objekte zwischen den verschiedenen Instanzen der Applikation. Programme, die aus der eigentlichen Applikation und der Terracotta-Zwischenschicht bestehen, können sowohl auf einem Rechner mehrfach, als auch auf verschiedenen Rechnern gestartet werden. Letzteres entspricht dem eigentlichen Gedanken des Parallelisierens.
Die verwendeten Rechner sollten sich dabei in einem einzelnen, lokalen Netzwerk befinden, da für die einwandfreie Verwendung von Terracotta einige Netzwerkports geöffnet werden müssen. Zwischenliegende Router und besonders Firewalls erschweren eine schnelle Umsetzung. Außerdem sollten die Rechner sich dauerhaft im Netzwerk befinden bzw. eine sich nicht änderne IP-Adresse besitzen. Daher ist es schwierig Terracotta zusammen mit dem vorhandenen Cluster mit dem Batch-System zu nutzen, denn es ist vorher nicht bekannt, wann das System die Applikation von welchem Rechner ausgeführt wird.
Eine Möglichkeit besteht darin das Programm von Hand auf z.B. pool-dhcp01 und pool-dhcp02 zu starten. Vorher muss auf einem der beiden Rechner der Terracotta-Server gestartet werden und beide Applikationen müssen so eingestellt werden, dass sie sich mit diesem Server verbinden.
Eine konkrete Einbindung von Terracotta in unsere Applikation wurde allerdings nicht vorgenommen.

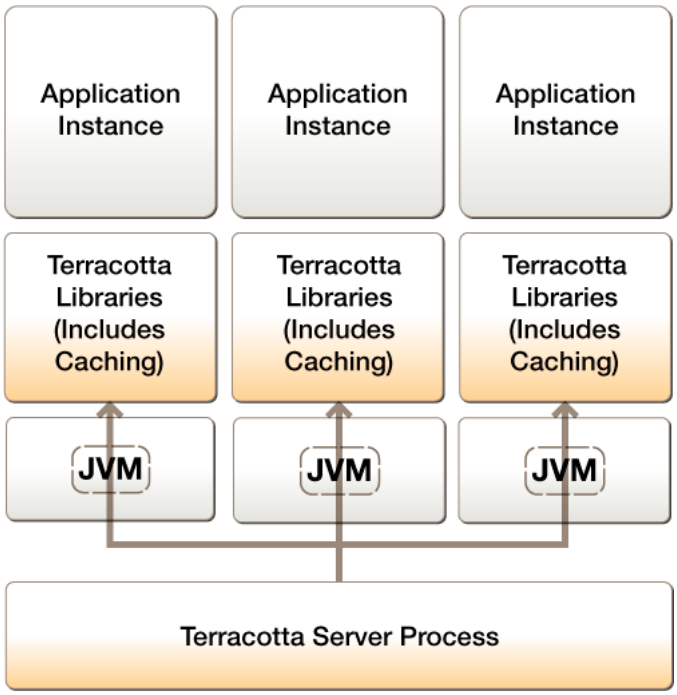

Abbildung 8: Darstellung des Terracotta-Stacks

4.5.2 Java Grid Application Toolkit

Das Java Grid Application Toolkit (Java GAT) ist eine Schnittstelle zu verschiedenen Cluster-Diensten. Durch Java GAT kann vom eigentlichen Cluster abstrahiert werden, so dass eine Anwendung nicht auf die spezifischen Unterschiede verschiedener Cluster eingehen muss.

Jedoch ist das Java Grid Application Toolkit kein bereits komplett abgeschlossendes Projekt. Es ist im Rahmen des GridLab Projektes entwickelt worden und wird vom Albert Einstein/Max-Planck Institut für Gravitationsphysik und der Louisiana State University stetig weiterentwickelt. Dadurch besitzt Java GAT jedoch teilweise noch nicht ausgereifte bzw. nicht implementierte Methoden. Dieses führt bei dem Versuch das Java Grid Application Toolkit zu nutzen zu erheblichen Problemen, da Fehlermeldungen wie „Function not implemented“ auftreten. In Verbindung mit der nicht vorhandenen Java-Doc zum Java Grid Application Toolkit machte dieses das Arbeiten mit Java GAT zu einer so großen Herausforderung, dass die Probleme die Java GAT bereitete, die erhofften Vorteile überstiegen, so dass die Nutzung von Java GAT wieder verworfen wurde.

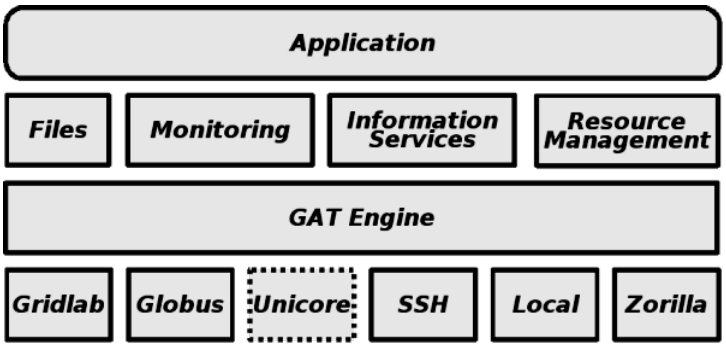

Abbildung 9: Darstellung des typischen GAT-Aufbaus

4.5.3 Clusterzugriff durch Shell-Skripte

Da sich die bisher betrachteten Bibliotheken als nicht geeignet für ein Portable-Batch-System von Torque erwiesen, wurde ein neuer Lösungsansatz erarbeitet, der ohne externe Bibliotheken auskommt und somit unabhängig von fremden Funktionalitäten ist.
Die Idee hinter diesem Ansatz beruht auf dem Zugriff des gemeinsamen Speichers aller Knoten des Clusters. Hierüber können Skripte und Konfigurationen aufgerufen und übergeben werden.
In dem konkreten Fall des Räuber-Beute-Modells bedeutet das, dass sowohl der Prototyp als auch ein externes Programm, welches für die Berechnung der Fitness verantwortlich ist, auf dem Hauptknoten des Clusters liegen. In dem überarbeiteten Beta-Prototypen wird der `Fitnessmanager` nun zu einer abstrakten Klasse, von der die konkreten Klassen `localFitnessmanager` und `parallelFitnessmanager` erben. Der `localFitnessManger` funktioniert dabei analog zum „normalen“ `Fitnessmanager`.
Der `parallelFitnessmanager` geht hingegen für jede zu berechnende Fitness in mehreren Schritten vor:

1. Erzeugen einer eindeutigen ID (Timestamp). Für diese wird die aktuelle Systemzeit genutzt. Existiert bereits eine solche ID (Timestamp), so wird auf eine Systemzeit gewartet, die noch nicht als ID genutzt wird.

2. Erzeugen einer Konfigurationsdatei, welche den Timestamp als eindeutige Kennzeichnung besitzt. Sie beinhaltet die für die Berechnung der Fitness relevanten Werte, wie die Gene der Beute und die Nummer der zu berechnenden Fitness.

3. Erzeugen eines Shell-Skripts, welches den Timestamp als eindeutige Kennzeichnung besitzt. Dies beinhaltet alle zum Aufruf eines `qsub` auf einem Portable-Batch-Systems relevanten Parameter und den Namen des externen Programmes als auszuführenden Job.

4. Ausführen des Shell-Skripts mit dem passenden Timestamp.

5. Warten, bis eine Remote-Datei mit dem passenden Timestamp als Kennzeichnung existiert, die nicht leer ist und auf die zugegriffen werden darf.

6. Auslesen der Zahl aus der Remote-Datei mit passendem Timestamp als Kennzeichung.

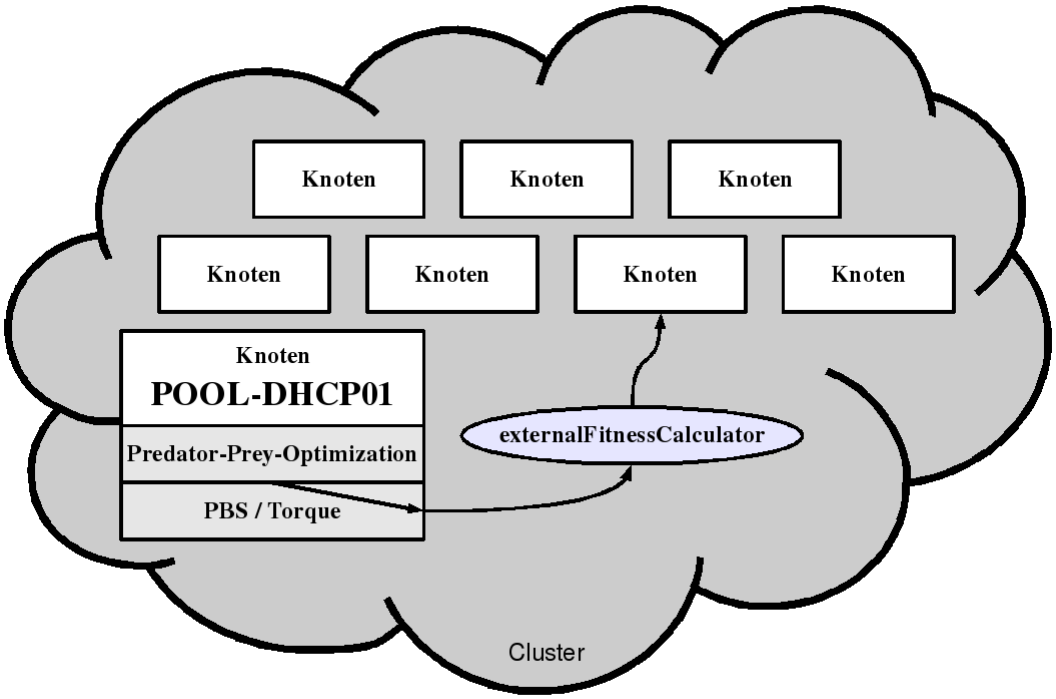

Abbildung 10: Ablauf der Shell-Skript basierten Parallelisierung

Zu betrachten bleibt, welchen Ablauf das externe Programm besitzt. Dieses Programm geht bei jedem Aufruf in folgenden Schritten vor:

1. Erzeugen eines Beuteindividuums und Auslesen der Nummer der Fitnessfunktion aus der Konfigurationsdatei mit dem passenden Timestamp.

2. Berechnen des Fitnesswertes der Beute in Bezug auf die passende Fitnessfunktion.

3. Erzeugen der Remote-Datei mit dem zugehörigen Timestamp als eindeutige Kennzeichnung.

4. Schreiben des errechneten Fitnesswertes als String in die Remote-Datei

- Sonstiger Output wird in die Output-Datei mit dem zugehörigen Timestamp als eindeutige Kennzeichnung geschrieben.

- Fehlermeldungen werden in die Error-Datei mit dem zugehörigen Timestamp als eindeutige Kennzeichnung geschrieben.

4.5.4 DRMAA Java-Binding

Das skriptbasierte Ansprechen des Clusters ohne Benutzung externer Bibliotheken funktioniert gut, jedoch sind die Kosten eine einfache Portierbarkeit des Frameworks gewesen. Das Skript, durch welches das Cluster angesprochen wird, ist auf das Portable-Batch-System von Torque zugeschnitten.

Soll das Framework nun auf anderen Clustern eingesetzt werden, ist dieses mit einem hohen Aufwand verbunden. So muss das Ansprechen dieses Clusters vollkommen neu implementiert werden. Da das nicht mit dem Framework-Gedanken zu verbinden ist, wurde eine neuer Ansatz entwickelt bei dem erneut externe Bibliotheken genutzt werden.
Dieser Ansatz benutzt zum Ansprechen des Clusters die `Distributed Rescource Management API` (DRMAA). Diese Schnittstelle ermöglicht ein äquivalentes Ansprechen aller Cluster unabhängig von deren System. Somit kann das Framework zum Räuber-Beute-Modell auf jedem Cluster ausgeführt werden, für welches ein DRMAA-Binding vorhanden ist, ohne den Code verändern zu müssen.
Bis auf die Art, wie das Cluster angesprochen wird, bleibt dieser Parallelisierungsansatz jedoch sehr ähnlich zu dem skriptbasierten Ansprechen des Clusters. So werden wie bisher sowohl die Konfiguration als auch die Ergebnisse über den gemeinsamen Speicher übergeben. Die eindeutige Zuordnung der Konfigurations- und Ergebnissdateien zu dem jeweiligen Job wird durch eine eindeutige ID erreicht. Der bisherige Rahmen kann daher bei diesem Ansatz wiederverwendet werden.
Zu betrachten bleibt daher die genutze `Distributed Resource Management API`:

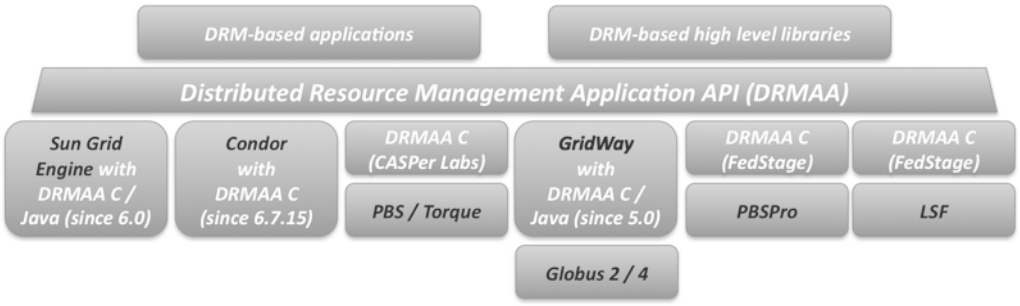

Abbildung 11: Darstellung des DRMAA-Stacks

Wie auf Abbildung 11 zu erkennen ist, gibt es jedoch kein verfügbares DRMAA-Java-Binding für das Portable-Batch-System von Torque. Daher musste dieses DRMAA-Java-Binding für das Portable-Batch-System von Torque selbst implementiert werden.
Der Grundgedanke hierfür war es, einen Adapter mittels `Java Native Interface` (JNI) (siehe Kapitel A) für Java zu schreiben, der auf dem DRMAA-C-Binding von Gridway aufsetzt und analog zum Java-Binding für Globus funktioniert. Jedoch konnte das DRMAA-Java-Binding für Globus nicht direkt übernommen werden, da hier Methoden existierten, die auf Globusattribute zugreifen. Ferner musste das DRMAA-Java-Binding für Torque um Methoden ergänzt werden, die Attribute und Zustände des Portable Batch Systems abfragen und setzen. Diese Implementierung hatte in C zu erfolgen.

4.5.5 Nutzung von DRMAA

Wenn das Räuber-Beute-Modell auf dem Cluster unter Nutzung von DRMAA gestartet werden soll, muss vorher die Systemvariable `LD_LIBRARY_PATH` gesetzt werden. Die Variable muss den Pfad zu der `libdrmaa.so` enthalten. Diese Bibliothek sollte beim Einrichten des PBS/Torque mit DRMAA erstellt worden sein und auf dem IRF-Cluster unter `/export/local/lib` liegen. Außerdem muss die Variable den Pfad zu der `libDrmaaJNI.so` B enthalten die zum oben genannten DRMAA-Java-Binding gehöhrt.

4.6 Parallelisierung des Modells

Neben der Parallelisierung des Systems sollte auch das Modell parallelisiert werden. Es sollte die Möglichkeit geschaffen werden, Aktionen innerhalb des Modells voneinander unabhängig auszu-

führen, um die modellinhaerente Flexibilität bei der Bewertung der einzelnen Fitnesskriterien zur Parallelisierung ausnutzen zu können, sowie Experimente mit größerer Dynamik zu ermöglichen.
Um ein solches Modell zu realisieren, mussten zunächst einige Änderungen an der Infrastruktur vorgenommen werden. Das vorhergehende Design konnte die Anforderungen nicht ohne grundlegende Strukturänderungen erfüllen. Zudem gab es einige weitere notwendige Änderungen und Umstrukturierungen in einigen zentralen Klassen, die hier ebenfalls erläutert werden sollen. Dazu gehören zum einen die Aktionen, die vom Räuber oder auch der Beute ausgeführt werden, aber auch der Evaluator und die Engine. Auch der Outputmanager musste geändert werden, da dieser nicht mehr sequentiell über Änderungen informiert wird, sondern durch die Parallelisierung von mehreren Prozessen zeitgleich Events eintreffen, die abgearbeitet werden müssen.

4.6.1 Infrastruktur zur Parallelisierung

Um die Aktionen im Räuber-Beute-Modell parallel ausführen zu können, müssen einige Komponenten der Infrastruktur angepasst und neue erstellt werden. Da alle Aktionen der Beute und Räuber auf denselben Objekten arbeiten, ist ein neues Konzept für die Zugriffsrechte sehr wichtig. Hierfür wurden Read- und Writelocks verwendet. Des Weiteren besteht die Anforderung an das Modell, dass es deadlockfrei sein sollte. Das bedeutet, dass bei der Vergabe der Schreib- und Leserechte darauf geachtet werden muss, dass sich Aktionen nicht blockieren.

Das Konzept der Read- & Writelocks

Zur Umsetzung der Read- & Writelocks standen mehrere Konzepte zur diskussion. Zunächst einmal musste analysiert werden, welche Objekte alle geschützt werden müssen. Generell gibt es nur zwei verschiedene Objekttypen, die immer wieder verändert werden, das sind zum einen die Positionen, zum anderen die Individuen.
Bei genauerem Betrachten, wird ein Individuum nur dann verändert, wenn auch die Position verändert wird. Somit wird hier auf ein explizites Sperren der Individuen verzichtet, und lediglich ein System zum Sperren der Positionen implementiert. Neben dem Sperren von einzelnen Positionen die verändert werden, sollte das Locksystem nicht nur garantieren, dass das Programm threadsafe, sondern auch, dass es deadlockfrei ist.
Im `TorusMaker`, in welchem Positionen und die Welt (in dem Falle der Torus) erzeugt werden, wird das Locksystem initialisiert. Um im Verlaufe des Algorithmus eine geringe Laufzeit der Aktionen zu erreichen, kann der Benutzer die Anzahl der Level die das Locksystem haben sollte, bestimmen. Minimal werden zwei Level benötigt, das Positionslock, welches eine einzelne Position und das Masterlock, welches alle Positionen, also die gesammte Welt, sperrt. Zwischen diesen Locks können beliebig viele Zwischenstufen definiert werden, die folgendermaßen eingeteilt werden.
Die Locks einer Stufe haben alle die selbe Größe. Zudem sind ebenfalls die Gruppengrößen gleich, das bedeutet, wenn zum Beispiel 5 Positionslocks zu einem Level-1-Zwichenlock zusammengefasst werden, so werden 5 Level-1-Zwischenlocks zu einem Level-2-Zwischenlock zusammengefasst, und so weiter. Dieses wird mit der Torusgröße, der Dimension und der Anzahl der gewünschten Zwischenlockstufen im `Torusmaker` berechnet. Lässt sich die Größe der Locks nicht exakt einteilen, so wird abgerundet.
Beim bestimmen der Anzahl der Zwischenlocks muss der Benutzer darauf achten, dass die Größe sinnvoll gesetzt wird. Wenn beispielsweise die Schrittweite bei 3 liegt, die erste Stufe des Zwischenlocks nur 5 Positionen umfasst, so wird diese nie alle gewünschten Positionen umfassen. Es müsste dann immer erst das Level-2-Lock gesperrt werden, um alle beteiligten Level-1-Locks zu erreichen.
Der Aufbau der Locks ist in Abbildung 4.6.1 visuell dargestellt. Das Locklevel wurde hier auf drei Stufen festgelegt. Das bedeutet, dass es die Positionslocks, das Masterlock und ein Zwischenlock gibt.
Um in der Utility Klasse `LockUtils` standardmäßig auf die Locks zugreifen zu können, wurde in der Klasse `Position` ein Array eingeführt, in dem die Locks gespeichert werden. Dabei ist die erste

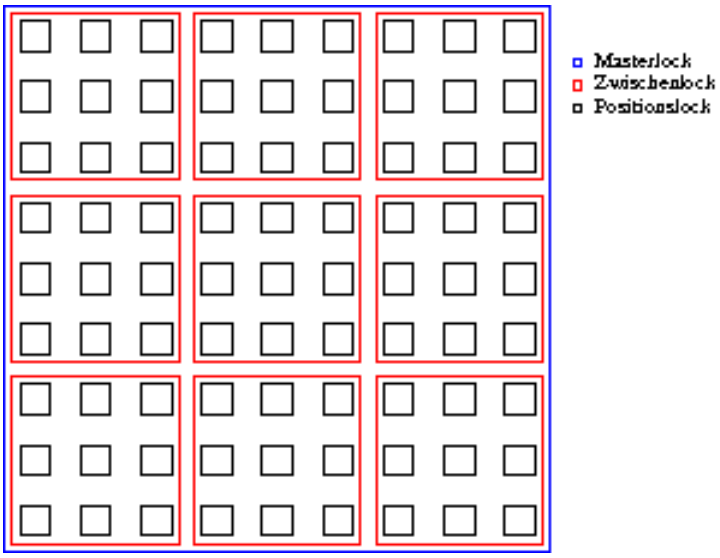

Abbildung 12: Aufteilung der Locklevel auf dem Torus

Stelle des Arrays für das Positionslock und die letzte für das Masterlock reserviert. Dazwischen sind geordnet die anderen Locks hinterlegt.

Die Utility Klasse `LockUtils`

Um nicht in jeder Aktion und in jeder Klasse, in der Positionen und Individuen verändert werden, den Nutzer zu zwingen, sich mit den Lockstrukturen auseinander zu setzen, wurde die Utility Klasse `LockUtils` implementiert. Somit werden die Aufgaben des Sperrens und wieder Freigebens von Positionen zentral geregelt. Die Funktionen müssen nur noch mit den richtigen Parametern aufgerufen werden.

- `gatherPositions(Position position, final int range, final boolean excludeStart, final boolean doWriteLock)`

  Diese Methode setzt die Read- bzw. Writelocks. Wenn `doWriteLock` den Wert `true` hat, so werden Schreibrechte vergeben, an sonsten nur Leserechte. Mit Hilfe der Startposition und der Reichweite, wird die oben beschriebene Methode `gatherPositionLocks` aufgerufen, und somit die größte gemeinsame Sperre gesucht. Diese Positionen werden dann mit den Richtigen Locks versehen. Hier kann kein Deadlock entstehen oder Anfragen unberücksichtigt bleiben, da zum einen die Vergabe fair ist und zum anderen erst das größte gemeinsame Lock gesperrt wird. Des Weiteren wird geprüft, ob die Startposition mit Lock versehen werden soll, oder ausgeschlossen wird. Zum Schluss der Methode werden alle Locks außer die Positionslocks (Level-0-Locks) wieder frei gegeben, so dass andere Aktionen diese wieder anfordern können.

- `releaseLocks(Set<Position> pool, Set<Position> exceptions, boolean writeLock)`

  Diese Methode gibt die Schreib- und Leserechte wieder frei. Dabei können auch Positionen angegeben werden, die noch nicht freigegeben werden sollen, da diese noch weiterhin gebraucht werden. Das verkürzt unter umständen die Wartezeiten für andere Threads. Es werden nur noch die Positionslocks (Level-0-Locks) freigegeben werden.

- `gatherPositionLocks(final Position pos, final int range)`

  Diese Methode gibt eine Liste Locklevel zurück, wobei zu jedem Listenelement eine Menge an Locks gespeichert ist. Das bedeutet, dass an erster Stelle der Liste alle Read- und Writelocks der Positionen sind, also alle „Level 0“- Locks. Am Ende der Liste befindet sich nur noch das Masterlock und daszwischen alle anderen Locks der jeweiligen Zwischenstufen. Zur

Berechnung wird hier eine Startposition benötigt, sowie eine Reichweite, in der die Positionen gesperrt werden sollen. Der Benutzer braucht die Methode nicht aufrufen, diese wird aber von `gatherPositions` benötigt.

Alle anderen Methoden dieser Klasse sind private. In diesen wurden Funktionen, die mehrere Methoden zur Berechnung benötigen, ausgelagert, so dass diese nicht in jeder Methode neu geschrieben werden müssen. Das gestalltet die Utility Klasse übersichtlicher.
Für dieses Locksystem gibt es verschiedene Regeln, die beachtet werden sollten. Diese sind in der Klasse `Aktion.java` dokumentiert:

- Man sollte nur auf Individuen oder Positionen zugreifen, ohne die zugehörige Position vorher zu sperren, wenn man nur die ID, die final ist, betrachten möchte, eine Liste von Positionen mit den zugehörigen Locks ausliest, da diese nicht modifiziert werden kann, oder die Position des aktuell betrachteten Individuums auslesen möchte.
- Aktionen verändern keine Positionen von anderen Individuen. Positionen neuer Individuen werden über den Konstruktor gesetzt. Bei Bewegungsaktionen wird nur das eigene Individuum auf andere Positionen gesetzt, die Positionen sollten hierfür gesperrt sein. Das ist wichtig, da die Individuen nur über das Positionslock geschützt sind.
- Jede Aktion muss mindestens ein Readlock für die Position des eigenen Individuums setzen bevor auf die Position oder das Indiviuum zugegriffen wird (außer in oben beschriebenen Situationen. Sollte das Individuum oder die Position verändert werden, müssen natürlich Writelocks verwendet werden.
- Wenn man mehr als eine Position sperren möchte, müssen erst die Locks auf höheren Leveln gesperrt werden, angefangen bei dem Level, das alle zu sperrenden Positionen umfasst. An sonsten können Deadlocks auftreten.
- Wenn Locks benutzt werden, sollte jeder Writelock so schnell wie möglich wieder freigegeben werden. Nur so können auch andere Aktionen ausgeführt werden und die Performanz des Modells kann verbessert werden.
- Zum Sperren der Positionen sollten die Methoden der `LockUtils` verwendet werden.
- Da Locks für eine Position nie verändert werden können, sollten diese zur Verbesserung der Performanz gecached werden.
- Die Nachbarschaften können sich zur Laufzeit ändern. Damit geht auch eine Änderung der Lockstruktur einher.

4.6.2 Konzept der Aktionen

Wie oben beschrieben wurde die Infrastruktur angepasst, um die Aktionen parallel ausführen zu können. Dazu war das Implementieren der Lese- und Schreibrechte ein notwendig, um zu vermeiden, dass mehrere Aktionen zeitgleich auf das selbe Objekt im Speicher zugreifen, und dieses verändern wollen.
Die Aktionen waren nach ursprünglichem Design so implementiert, dass diese eine Art Pre-Processing beinhalteten, in dem Positionen und Individuen, die betrachtet werden sollten, angefordert wurden. Darauf folgte der vom Nutzer zu implementierende Teil, welcher aber nicht zwangsweise die bereitgestellten Positionen und Individuen nutzen musste, sondern der Nutzer konnte hier Implementieren was er wollte, hauptsache der Rückgabewert stimmte. Nach diesem Teil folgte die Erzeugung des Events für die Ausgabe. Um die Aktion threadsafe zu gestallten, konnten diese drei Teile nicht mehr unabhängig betrachtet werden. Das Pre-Processing sollte nach dem erarbeiteten Konzept das Sperren der Positionen vornehmen, so dass der Nutzer sich zum einen darum nicht mehr kümmern

muss, zum anderen aber eingeschränkt wird in der Implementierung, da dieser nur genau die gesperrten Positionen und keine weiteren betrachten darf. Das Past-Processing nimmt das Senden der Events vor und im Anschluss das Freigeben der gesperrten Positionen. Nicht benötigte Positionen können auch vorher vom Nutzer wieder freigegeben werden.

Durch das System der fairen Locks bekommt jede Aktion ihre Positionen, die sie benötigt. Zum Beispiel: Zwei Aktionen fordern Positionen an. Die eine Aktion ist eine Bewegung eines Individuums und meldet Locks für Position 1, 2, 3, 10, 11, 12, 19, 20 und 21 an. Eine Räuber möchte jagen und meldet für Positionen 3, 4, 5, 12, 13, 14, 21, 22 und 23 eine Consumption an. Nach den bestehenden Zwischenstufen kommt die Bewegungsaktion mit einem Level-1-Lock aus, die Consumption braucht das Lock auf der höheren Stufe, also ein Level-2-Lock. Im folgenden Schritt sperrt die Bewegungsaktion das Level-1-Lock, die Consumption das Level-2-Lock. Da keine einzelne Position gesperrt ist kann die Bewegung die einzelnen Positionen sperren, und das Level-1-Lock wieder frei geben. Die Consumption kann noch nicht direkt die Positionen sperren, sondern braucht erst die Level-1-Locks. Hier ist aber eins noch von der Bewegung gesperrt; durch die Verwendung fairer Locks bekommt die Consumption dieses aber, sobald es frei gegeben wurde. Die fehlenden Positionen bekommt die Consumption ebenfalls, wenn die Bewegung diese nicht mehr benötigt. Erst wenn die Consumption alle Level-0-Locks besitzt, die angefordert wurden, werden die Level-1-Locks und das Level-2-Lock freigegeben. Somit ist garantiert, dass die Aktionen threadsafe sind und deadlockfrei agieren.

Im vorherigen Konzept gab es abstrakte Klassen für die unterschiedlichen Aktionen wie `Movement`, `Reproduction` und `Consumption`. In der überarbeiteten Version gibt es diese abstrakten Klassen nur noch für das `Movement` und die `Consumption`. Die `Reproduktion` ließ sich nicht gut auf einen Nenner zusammenfassen, so dass der Nutzer hier jede Form der `Reproduktion` selbst schreiben muss.
Im Folgenden werden die drei Aktionstypen und ihre Implementierung genauer beschrieben:

- `ThreadsafeMovement`

  In der Methode `perform` wird hier begonnen bei der Startposition die benötigten Positionen zu sperren. Dazu wird die Methode `gatherPositions` der `LockUtils` aufgerufen. Aus diesen Positionen wird die Zielposition ausgewählt. Die dazu gehörige Methode `getDestination` wird vom Nutzer des Frameworks implementiert. Nach den Lockregeln werden danach die nicht mehr benötigten Positionen freigegeben (alle außer Start und Ziel der Bewegung). Die Bewegung wird ausgeführt und das Event für die Ausgabe erzeugt. Zum Schluss werden auch die zwei noch gesperrten Positionen freigegeben.

  Wie aus der Menge der Positionen das Ziel der Bewegung gewählt werden soll, kann der Nutzer bestimmen. Eine uniform zufällige Wahl ist bereits implementiert. Das einzige worauf der Nutzer achten muss, ist nur die gesperrten Positionen zur Wahl heranzuziehen, sonst kann Threadsicherheit und Deadlockfreiheit nicht garantiert werden.

- `ThreadsafeReproduction`

  Die `ThreadsafeReproduction` ist in keiner abstrakten Klasse zusammen gefasst. Es gibt als Beispiel für den Benutzer des Frameworks eine Implementierung für die Selbstmutation. Alle anderen Formen der Reproduktion kann vom Benutzer implementiert und eingebunden werden, jedoch sollten die Regeln der Locks beachtet werden. Neue Individuen werden im Konstruktor beim `IndividualManager` angemeldet, so dass die Aktionen der Individuen auch ausgeführt werden.

- `ThreadsafeConsumption`

  Die `ThreadsafeConsumption` ist ebenfalls in einer abstrakten Klasse zusammengefasst. Hier werden wie beim `ThreadsafeMovement` zunächst die benötigten Positionen gesperrt. Sind in diesem Bereich keine Individuen vorhanden, die gefressen werden können, so werden die

Positionen wieder frei gegeben. Andernfalls evaluiert der Räuber die vorhandene Beute und selektiert ein Individuum, welches getötet werden soll. Die Methoden `select` und `evaluate` werden hierbei vom Nutzer implementiert und müssen sich ähnlich wie beim `ThreadsafeMovement` an die Vorgaben durch die gesperrten Positionen halten. Zum Schluss der Methode werden die gesperrten Positionen wieder frei gegeben, so dass andere Aktionen diese Positionen benutzen und verändern können.

Da das Modell nicht mehr Rundenbasiert ist, wird das Fressen des Räubers nicht mehr direkt durch die Engine kontrolliert. Dadurch das jeder Räuber gleichzeitig sich bewegt und frisst, ist nur noch entscheidend, wie schnell seine Berechnungsergebnisse vorliegen. Ist die Evaluationsfunktion eines Räubers bedeutend einfacher als die eines anderen Räubers, ist ein Kontrollmechanismus als „faire Consumption“ implementiert worden, die verhindert, dass ein Räuber um einen bestimmten Faktor mehr frisst als der andere Räuber.

4.6.3 Konzept der Evaluatoren

An die Evaluatoren stellen sich besondere Anforderungen, weil sie in verschiedenen Bereichen des Frameworks zur Berechnung von Fitnesswerten genutzt werden. Parallelisiert man das Modell, müssen sich diese einzelnen Instanzen bei gleichzeitigen Funktionsaufrufen untereinander koordinieren. Dabei ergeben sich folgende Anforderungen:

- Der Nutzer muss die Anzahl der Threads zur Berechnung der Fitnesswerte festlegen können, damit die Systemlast unabhängig von der Anzahl der gleichzeitig genutzten Evaluatoren handhabbar bleibt und nicht beispielsweise 4000 Threads auf einmal erzeugt werden, nur weil mehrere parallel arbeitende Räuber und eine Auswertungsmetrik zum selben Zeitpunkt Fitnesswerte benötigen.
- Wenn verschiedene Evaluatorinstanzen gleichzeitig aufgefordert werden, für ein Individuum Fitnesswerte auszurechnen, sollen keine Berechnungen für ein bestimmtes Fitnesskriterium doppelt gestartet werden. Stattdessen soll auf den Abschluss der bereits laufenden Berechnung gewartet werden. Der Zugriff auf die hierbei verwendeten Datenstrukturen muss thread-safe erfolgen.
- Die Anzahl der Funktionsauswertungen während des Modellablaufs muss ermittelt werden, um der Engine eine Terminierung zu ermöglichen, wenn ein einstellbares Limit überschritten wird. Diese Funktionalität ist deshalb wünschenswert, weil die benötigten Funktionsauswertungen ein verbreitetes Beurteilungskriterium bei Algorithmen zur mehrkriteriellen Optimierung darstellen.

Die existierenden Implementierungen zu `Evaluator`, `TestProblem` und `Computation` wurden unter Berücksichtigung dieser Punkte überarbeitet. Um die letzte Anforderung erfüllen zu können, wurde ein EvaluatorManager entwickelt, der die entsprechenden Informationen verwaltet.

Die Ausführung der Computations erfolgt in den Evaluatoren nun unter Nutzung eines Threadpools, der über den `ExecutorService` aus dem Concurrency-Paket von Java 5.0 realisiert ist. Damit entfällt die Erzeugung neuer Threads für einzelne Computations. Die Anzahl der Threads in diesem Pool kann vom Benutzer frei gewählt werden, wobei sich für Fitnessauswertungen auf dem lokalen Rechner eine den CPU-Kernen entsprechende Anzahl empfiehlt, während sie bei Nutzung eines Rechenclusters so gewählt werden sollte, wie es dessen Kontrollsystem vorgibt.
Der neue `EvaluatorManager` bietet einen zentralen Punkt, der Informationen über Funktionsauswertungen sammelt. Primär dient dies dazu, einen Terminierungszeitpunkt für den Modellauf vorgeben und Statistiken zum Modellverhalten liefern zu können. Zu diesem Zweck werden Fitnessauswertungen nicht nur erfasst, sondern zusätzlich zwischen tatsächlich ausgeführten Berechnungen

und aus dem Cache beantworteten Ergebnissen unterschieden. Darüber hinaus ist diese Managementklasse als Vermittler zwischen Evaluatoren und Computations vorgesehen, damit im oben erwähnten Fall einer doppelten Evaluationsanforderung keiner der Evaluatoren in einer aktiven Warteschleife bleiben muss, sondern der entsprechende Thread wirklich pausiert und aufgeweckt wird, wenn das Ergebnis nach Beendigung der Berechnung vorliegt.
Um gleichzeitige Anfragen nach denselben Fitnesswerten erkennen zu können, erfassen die Evaluatoren in einer statischen Klassenstruktur Individuen und zugehörige Fitnesskriterien, deren Berechnung derzeit läuft. Diese Informationen stehen somit allen nebenläufig arbeitenden Evaluatoren jederzeit zur Verfügung und die Evaluatoren können vermeiden, eine doppelte Berechnung zu starten. Stattdessen merkt sich der betroffene Evaluator die Individuen, für die bereits Berechnungen laufen, startet bei Bedarf weitere notwendige Berechnungen, und wartet auf die ausstehenden Ergebnisse, bevor die aufgerufene Evaluationsfunktion beendet ist.

4.6.4 Die Engine

Bei der Anforderungsanalyse zur Engine ergaben sich folgende Punkte, die im Kontext der Parallelisierung berücksichtigt und bei der Implementierung schließlich umgesetzt werden mussten:

- Aktionen und Umwelteinflüsse sollen in einer einstellbaren Anzahl von Threads parallel zueinander ausgeführt werden können.
- Um eine effiziente Nutzung der zur Verfügung stehenden Rechenleistung zu gewährleisten, dürfen einzelne Aktionen mit langer Laufzeit den Rest des Systems nicht blockieren. Nur dann kann ein moderner Computer mit vier, acht oder mehr Kernen effizient genutzt werden.
- Es soll möglich sein, die Verteilung der einzelnen Individuen auf Threads in der Engine zu beeinflussen, um beispielsweise Räubern jeweils einen Thread zuzuweisen und die Last so gleichmässig zu verteilen. Ansonsten soll die Engine von sich aus versuchen, eine Fluktuation der Individuenanzahl in den Threads möglichst wieder auszugleichen.
- Ein Nutzer muss auch weiterhin eine sequentielle Ausführung erreichen können, wenn ihm die Dynamik bei paralleler Ausführung zu groß ist oder für bestimmte Experimente reihum agierende Individuen notwendig sind.
- Da bereits umfangreiche Implementierungen von Aktionen existieren, die nicht thread-safe sind und somit nicht parallel zueinander ausgeführt werden dürfen, soll die Engine diese weiterhin verarbeiten und auch zusammen mit parallelisierbaren Aktionen abarbeiten können, wobei dies natürlich nur abwechselnd erfolgen kann, um die Sicherheit des Modells für alte Aktionen zu garantieren.
- Das Modell soll nach einer einstellbaren Anzahl von Funktionsauswertungen angehalten werden, weil die benötigten Funktionsauswertungen ein verbreitetes Beurteilungskriterium bei Algorithmen zur mehrkriteriellen Optimierung darstellen.

Da die bisher zur Verfügung stehende Implementierung einer Engine mehrere dieser Kriterien aufgrund ihres Designs nicht erfüllen konnte, insbesondere nicht den zweiten Punkt, wurde beschlossen, eine komplett neue Version zu entwickeln.

Die neue Implementierung besteht aus zwei Komponenten: einer Kontrollinstanz zur Koordinierung namens `ControllingEngine` und untergeordneten Arbeitsinstanzen der Klasse `SubThreadEngine`. Die Anzahl dieser Arbeitsinstanzen, die jeweils in einzelnen Threads laufen, kann auf verschiedene Weise beeinflusst werden. Zunächst einmal ist es möglich, direkt die gewünschte Anzahl anzugeben. Außerdem kann sie über eine Menge von Spezies festgelegt werden, für deren Individuen beim Start des Modells jeweils ein Thread eingerichtet wird. So etwas kann in Modellexperimenten nützlich

sein, bei denen während des Ablaufs keine neuen Räuber erzeugt werden, aber die Anzahl beim Modellstart variiert wird. In diesem Fall werden die verbliebenen Individuen anderer Spezies anschließend unter diese Threads verteilt. Schließlich ist es noch möglich, für Umwelteinflüsse einen separaten Thread erzeugen zu lassen, damit diese nicht den übrigen Threads zugeordnet sind. Dies kann zum Beispiel ein Modell unterstützen, bei dem die Ausführung eines Umwelteinflusses langwierig ist oder von Zusatzbedingungen wie der Anzahl an bisherigen Funktionsauswertungen abhängt und aufgrund dessen Wartezeiten beinhaltet, so dass der Thread mit Umwelteinflüssen die meiste Zeit inaktiv verbringen soll, ohne die Ausführung von Aktionen der Individuen zu beeinflussen.
Ist eine komplett sequentielle Bearbeitung gewünscht, kann die Anzahl der Threads auf einen reduziert werden. Bei dieser Einstellung werden sämtliche Individuen und Umwelteinflüsse derselben `SubThreadEngine` zugewiesen und von ihr nacheinander abgearbeitet.
Sämtliche Threads nutzen in inaktiven Zeitabräumen eine `Condition` (Java 5.0), um nicht mit aktiven Warteschleifen Rechenleistung zu blockieren.

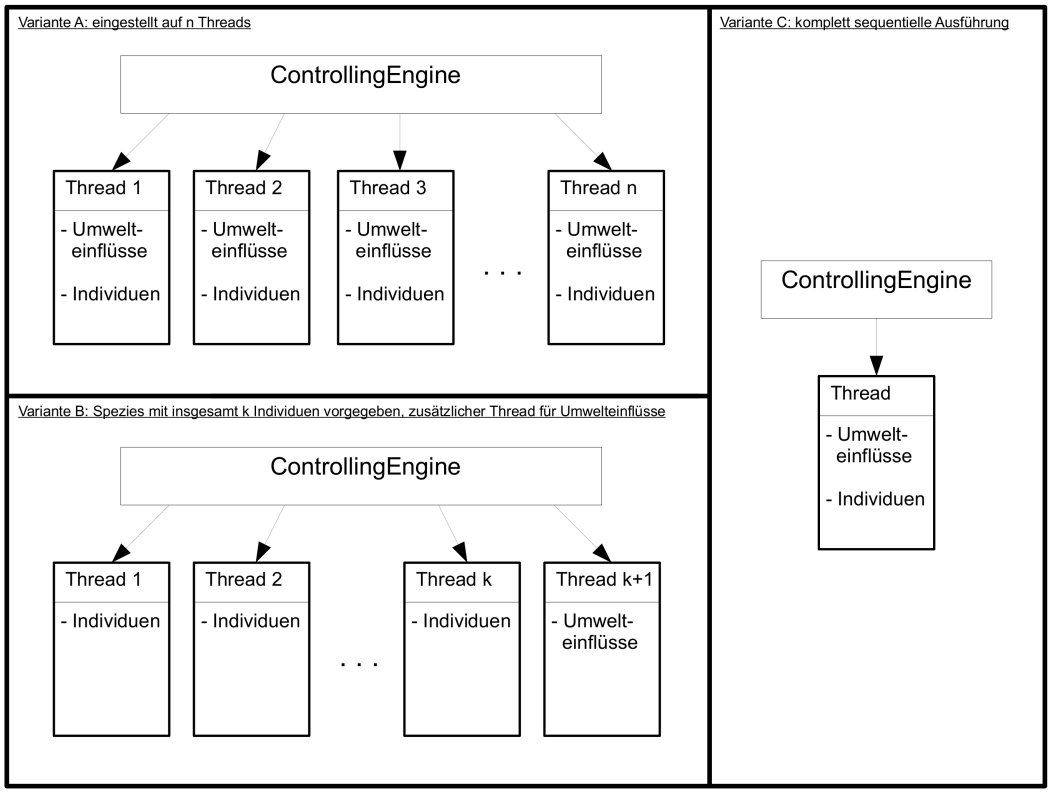

Abbildung 13: Drei Konfigurationsvarianten zur Parallelisierung

Da den `SubThreadEngines` in ihren Threads Individuen und gegebenenfalls Umwelteinflüsse exklusiv zugewiesen werden, steht deren unabhängiger Verarbeitung prinzipiell nichts im Wege. Benötigt eine Aktion in einem der Threads besonders lange, ist nur dieser Thread blockiert (unter der Voraussetzung, dass der Einflussbereich anderer Aktionen nicht mit dem der bereits laufenden Aktion kollidiert, siehe Kapitel 4.6.2).
Einen Sonderfall stellen nicht parallelisierbare Aktionen dar. Dazu zählen insbesondere sämtliche älteren Implementierungen der Projektgruppe, bei deren Entwicklung Nebenläufigkeit nicht berücksichtigt wurde und die aus diesem Grund nicht parallel abgearbeitet werden dürfen. Steht die Ausführung einer solchen an, beantragt die entsprechende `SubThreadEngine` deren exklusive Ausführung bei der `ControllingEngine`. Diese benachrichtigt alle Arbeitsthreads, welche daraufhin

sobald wie möglich pausieren, also in der Regel sobald die derzeit laufenden Aktionen abgeschlossen sind. Werden von mehreren Arbeitsthreads zeitgleich Exklusivrechte beantragt, so werden alle in eine Liste aufgenommen und nacheinander abgearbeitet, damit das Modell nur einmal angehalten werden muss. Dies tritt umso häufiger auf, je mehr nicht parallelisierbare Aktionen und Umwelteinflüsse im Modell verwendet werden. Nachdem alle Threads das Erreichen ihres Haltepunktes bestätigt haben, führt der Kontrollthread die anstehenden exklusiven Funktionen aus und gibt den wartenden Threads anschliessend das Signal, ihre Arbeit wieder aufzunehmen.

Wenn während des Modellablaufs neue Individuen erzeugt werden, die über eigene Aktionen verfügen, so werden diese bei ihrer Erzeugung automatisch vom `IndividualManager` erfasst und von der `ControllingEngine` unter den `SubThreadEngines` verteilt. Dabei wird versucht, die Anzahl der Individuen in den einzelnen Threads soweit möglich gleich groß zu halten. Dies stellt, abgesehen von der Behandlung exklusiver Zugriffe auf das Modell, die Hauptaufgabe der `ControllingEngine` dar. Die einzige verbleibende Aufgabe dieser Klasse ist es, Stopkriterien zu überprüfen. Beliebig viele davon können angemeldet werden und das Modell wird gestoppt, sobald eines der Kriterien erfüllt ist. Zwei Stopkriterien sind bereits fertig implementiert: Eines prüft auf eine festgelegte Anzahl an Fitnessfunktionsberechnungen, das andere auf eine Maximallaufzeit des Modells. Je nach Bedarf kann so eine geeignete Bedingung eingestellt werden. Für simple Funktionstests empfiehlt sich ein kurzer Zeitraum, zur Ermittlung konkreter Ergebnisse eine vorgegebene Berechnungszahl.
Dadurch, dass Aktionen der Individuen nun weitgehend voneinander unabhängig ausgeführt werden, müssen nicht mehr automatisch alle Räuber die gleiche Anzahl an Individuen fressen, auch wenn sie die gleichen Verhaltensmuster nutzen. Dies kann bei nicht sequentieller Bearbeitung zu einem Ungleichgewicht führen, wenn nicht geeignete Aktionsimplementierungen benutzt werden, die ihrerseits eine Fairness garantieren. Details hierzu sind im obenstehenden Abschnitt über Aktionen beschrieben.

4.6.5 Konzept der Events und der Ausgabe

Durch die Parallelisierung der Aktionen von Räuber- und Beuteindividuen entstanden weitere Anforderungen an das Design der Ausgabe und Analyse. Es muss garantiert werden, dass die Events Thread-safe sind und dass die parallel ausgeführten Aktionen Events erzeugen können. Das Auslösen von Events und deren Verarbeitung darf die Engine so wenig wie möglich verzögern. Des Weiteren soll die Auswertung des Modells, mittels Ausgabestrategien und Metriken, den Ablauf des Modells so wenig wie möglich stören, denn eine Auswertung des gesamten Modells erfordert Lesezugriff auf alle Positionen. Damit wäre es nötig alle Enginethreads anzuhalten um dann Lesezugriff auf alle Positionen erhalten zu können.

Die grundlegende Idee ist alle Informationen von Positionen und Individuen indirekt und über den gesamten Modellablauf verfügbar zu machen. Dazu wurde eine zentrale Instanz geschaffen, um Veränderungen des Modells wahrnehmen und Informationsanfragen entgegen nehmen zu können. Zur Steuerung dieser Instanz werden Events verwendet. Jeder Event stellt eine Veränderung im Modell dar. Wenn nun alle Veränderungen im Modell durch Events dargestellt werden und diese Events in der Reihenfolge ihres Auftretens gespeichert werden, dann kann unabhängig vom aktuellen Modellzustand jeder andere vorherige Modellzustand wiederhergestellt werden.
Diese zentrale Instanz wurde im Interface `IOutputEngine` und in ihrer Implementierung `OutputEngineImpl` realisiert. Dieses Interface soll als Erweiterung für den `OutputManager` verstanden werden. Die OutputEngine wird in einem eigenen Thread gestartet und nimmt alle Events an und verarbeitet diese. Alle Events, die das Interface `StorableEvent` implemetieren können durch die OutputEngine verarbeitet werden. Dazu muss der Event alle Informationen der Veränderung, die er darstellt, unabhängig von den Modellinstanzen speichern. Dazu wurden Repräsentationsklassen für Events, Individuen und Positionen erstellt. Ein sogenannter `StatusEvent` stellt eine Veränderung des Modells dar. Dazu werden alle Individuen und Positionen, die durch den Event betroffen sind,

gespeichert. Die Individuen und Positionen werden durch die Klassen `StatusIndividual` und `StatusPosition` dargestellt. Diese Klassen speichern alle Informationen, um daraus wieder Individuen und Positionen rekonstruieren zu können.

Die Klassen wurden EJB3 konform erstellt, um durch ein Object-Relational Mapping (ORM) in einer Datenbank gespeichert werden zu können. Für dieses ORM wurde das Open-Source-Framework Hibernate verwendet. Das Singleton `HibernateUtil` stellt alle wichtigen Methoden zur Verfügung, um einen StatusEvent und alle betroffenen Positionen und Individuen zu persistieren. Diese Informationen sind damit unabhängig vom Modellablauf zugreifbar. Durch die Klassen `PositionFactory` und `IndividualFactory` ist es möglich, StatusPositionen und StatusIndividuen zu Individuen und Positionen umzuwandeln. Mit diesen neu erstellten Individuen ist eine vom Modellablauf unabhängige Visualisierung und Analyse möglich.
Die OutputEngine erstellt zeitgesteuert Snapshots, um diese an die Ausgabe und Analyse zu übergeben. Dazu werden alle bisher verarbeiteten Events zu einem Modellzustand aggregiert. Dieser Modellzustand kann dann wie bisher durch die Ausgabestrategien und Metriken verarbeitet werden.

Durch diese Revisionsverwaltung des Modells wird also der komplette Ablauf gespeichert. Damit entstehen, zusätzlich zum bisherigen Ausgabe- und Analysedesign, Analysemöglichkeiten des gesamten Modellablaufs.

4.7 Visualisierung und Ausgabe

Die Ausgabe gliedert sich im Wesentlichen in zwei Bereiche. Zum einen in die Ausgabe von Textdateien, die später zur Analyse durch andere Programme herangezogen werden können, zum anderen in die Simulation der Systemdynamik mittels Plotts und Metriken, die während des Programmdurchlaufs erstellt und angezeigt werden.

4.7.1 Events als Schnittstelle zwischen Framework und Ausgabe

Die gesamte Ausgabe arbeitet ereignisgesteuert. Dies bedeutet, dass spezielle Ereignisse erzeugt werden, diese werden vom Ausgabemanager aufgefangen und mittels Ausgabestrategien verarbeitet. Das Grundgerüst bildet die abstrakte Klasse `Event`. Sie stellt insbesondere die `send`-Methoden zur Verfügung (das für das entsprechende Ereignis verantwortliche Individuum kann entweder als Parameter mitgegeben werden oder nicht). Diese Methoden benachrichten den Ausgabemanager über die Erzeugung eines neuen Ereignisses. Soll nun ein neues Event erzeugt werden, so geschieht dies mittels

```
new <NameOfEventClass>(<Parameters for Event>).send()
```

bzw.

```
new <NameOfEventClass>(<Parameters for Event>)
    .send(responsablePredator)
```

Die einzige Ausnahme bildet das `CleanEvent`. Dieses wird erzeugt, um alle Ausgabetypen (vgl. übernächster Abschnitt) zu schließen. Da das `CleanEvent` nur einmal zum Programmabschluss erzeugt wird und nur der Notifikation der Ausgabetypen dient, kann auf die Übergabe eines verantwortlichen Individuums verzichtet werden. Stattdessen wird die `send`-Methode automatisch im Konstruktor aufgerufen.
Weiterhin werden in den Ereignissen die für das Ereignis relevanten Daten gespeichert. Diese können über get-Methoden ausgelesen werden. Ein für das Modell relevanter Kern von Ereignissen wurde ausgearbeitet und implementiert (eine kurze Beschreibung folgt). Natürlich kann diese Liste vom Benutzer beliebig erweitert werden.

CleanEvent Das `CleanEvent` wird benutzt, um die vorhandenen Ausgabetypen über das (sofort folgende) Programmende zu benachrichtigen.
Zusätzliche Attribute: Keine.

ConsumptionEvent Das `ConsumptionEvent` enthält Informationen über ein von einem Räuber entferntes Beuteindividuum.
Zusätzliche Attribute:

- `individual`: Das vom Räuber getötete Individuum.

MoveEvent Das `MoveEvent` wird ausgelöst, sobald sich ein Räuber bewegt. Hiermit lässt sich also der Weg eines Räubers verfolgen.
Zusätzliche Attribute:

- `from`: Die ursprüngliche Position.
- `to`: Die Zielposition.
- `individual`: Das sich bewegende Individuum.

MutationEvent Das `MutationEvent` wird ausgelöst, sobald ein Individuum mutiert wird.
Zusätzliche Attribute:

- `newIndividual`: Das Individuum nach der Mutation.
- `oldIndividual`: Das ursprüngliche Individuum, bevor es mutiert wurde.

ReprodutionEvent Das `ReproductionEvent` wird beim Reproduktionsvorgang erzeugt.
Zusätzliche Attribute:

- `newIndividuals`: Die Menge der neu erzeugten Individuen. Diese kann mehr als nur ein Element umfassen, man denke etwa an Crossover.
- `parentIndividuals`: Die Menge der Elternindividuen.

SnapshotEvent Das `SnapshotEvent` wird erzeugt, wenn ein Abbild der Welt erzeugt wird.
Zusätzliche Attribute:

- `positions:` Die Menge aller Positionen der Welt. Auf diesen finden sich die Beuteindividuen und Räuber wieder.

4.7.2 Der Ausgabemanager

Der Ausgabemanager wird durch die Klasse `OutputManager` als Singleton implementiert. Seine Hauptaufgabe besteht aus der Verwaltung einer Menge von Ausgabestrategien (vgl. nächster Abschnitt) und der Weitergabe von Ereignissen. Hierzu implementiert jede Ausgabestrategie eine Methode `getObservedEvents`, die die Menge der für die Ausgabestrategie relevanten Ereignisse als Set⟨Class⟩ zurückliefert. Vom Ausgabemanager wird nun eine Hashtabelle verwaltet, die jeder Ausgabe die Menge seiner relevanten Ereignisse zuordnet. Außerdem existiert eine Methode `notify`, die den Ausgabemanager über ein neu eintreffendes Ereignis (das als Parameter übergeben wird) informiert. Dieses Ereignis wird dann an alle Strategien weiterleitet, für die das Ereignis als relevant eingestuft wird.
Hieraus folgt natürlich auch, dass der Ausgabemanager initial über die Strategien, die er verwalten soll, informiert werden muss. Dies erfolgt über die Methode `addStrategy`. Da es aber äußerst

unschön ist, alle Strategien manuell einzubinden und zu konfigurieren, existiert die Datei ppoSystemConfig.xml. Dies ist eine konfigurierbare XML-Datei, in der Strategien hinzugefügt und konfiguriert werden können. Die Parameter für die Strategien werden dann vom Rahmenprogramm des Räuber-Beute-Modells automatisch gesetzt. Auch werden die Strategien ohne Benutzereingriff beim Ausgabemanager registriert.
Der Ausgabemanager wird für den Nutzer des Frameworks unsichtbar, da nur Events erzeugt, gesendet und dazu passende Strategien implementiert werden müssen.
Neben dieser Hauptaufgabe verwaltet der Ausgabemanager auch die vorhandenen Ausgabetypen (vgl. übernächster Abschnitt). Diese können wie bereits erwähnt dann am Ende durch Erzeugen eines `CleanEvents` geschlossen werden. Dies ist zum Beispiel wichtig für Textdateien, die extern nicht gelesen werden können, solange sie noch geöffnet sind, aber auch für Datenbanken usw.

4.7.3 Das Konzept der Ausgabestrategie

Eine Ausgabestrategie wartet auf eintreffende Ereignisse und reagiert passend auf diese. Die Menge der relevanten Ereignisse wird von der Methode `getObservedEvents` zurückgeliefert. Mittels `initialize` muss die Strategie beim Ausgabemanager registriert werden. Der Ausgabemanager verwaltet eine Map, die jeder Strategie die Menge der relevanten Ereignisse zuordnet. Wird nun ein neues Ereignis an den Ausgabemanager gesendet, so leitet er dieses an alle Strategien, die das entsprechende Ereignis als relevant einstufen, weiter. Dies geschieht, indem der `notify`-Methode der Strategie das Ereignis übergeben wird. Direkt danach wird die Methode `output` aufgerufen. Die Intention ist folgende: Eine Strategie kann auf ein Ereignis reagieren und die ggf. enthaltenen Informationen verarbeiten (zum Beispiel zwischenspeichern, relevante Teile extrahieren usw.) Danach wird mittels der `output`-Methode die Ausgabe erzeugt. Da der Strategie bekannt ist, welches Ereignis vorher eingetroffen ist, kann diese Methode abhängig davon entscheiden, welche Ausgabe generiert wird. Auf diese Art und Weise werden Informationsverarbeitung und Ausgabe getrennt. Die Ausgabe wird dann vermutlich sowieso von einem Ausgabetypen übernommen. Weiterhin existiert die Methode `buildHeader`. Diese ordnet – falls möglich – jeder Ausgabestrategie einen Header zu. Dies ist wichtig für Ausgabestrategien, die tabellarische Daten erzeugen (zum Beispiel PreyValues). Falls zu der Strategie kein Header angegeben werden kann (etwa weil Visualisierungsaufgaben erledigt werden), kann auch der leere Header zurückgeliefert werden. Sinnvoll ist dieses Vorgehen besonders deswegen, weil es den entsprechenden Ausgabetypen erlaubt, Informationen über die Struktur der Ausgabedaten der entsprechenden Strategie abzurufen. Eine Standard-Implementierung findet sich in der Klasse `DefaultOutputStrategy`.
Es folgt eine kurze Liste der vorhandenen Ausgabestrategien.

MetricStrategy Berechnet den aktuellen Wert zuvor registrierter Metriken. Die Klassen `WorldMetric`, `IndividualMetric` und `PositionMetric` bilden hierzu alle Positionen der Welt, einzelne Individuen bzw. einzelne Positionen auf Doublewerte ab. Die gesammelten Ergebnisse, sowie statistische Daten über Positions- und Individuen-metriken werden dann zur weiteren Verarbeitung den `OutputType`-Implementierungen angeboten.
Reagiert auf die folgenden Ereignisse:

- `SnapshotEvent`

ParetoAnalysis Berechnet (im Programm selbst) Paretomenge und -front und visualisiert diese, indem R aufgerufen wird. Die Bilder können auch in eine Datenbank eingefügt werden.
Reagiert auf die folgenden Ereignisse:

- `SnapshotEvent`

PredatorActionTracking Verfolgt den Verzehr und die Reproduktion von Individuen (diese Aktionen werden von einem Räuber durchgeführt, daher „Predator“).
Reagiert auf die folgenden Ereignisse:

- `ConsumptionEvent`
- `ReproductionEvent`

PredatorTracking Verfolgt die Bewegungen der Räuber.
Reagiert auf die folgenden Ereignisse:

- `MoveEvent`

PreyValues Speichert die Gene und Fitnesswerte aller Individuen auf dem Torus.
Reagiert auf die folgenden Ereignisse:

- `SnapshotEvent`

RAnalysis Abstrakte Klasse, dient zur Durchführung einer R-Analyse. Dabei werden alle Gene und Fitnesswerte in eine Datendatei geschrieben, die von einem R-Skript analysiert werden sollen.
Reagiert auf die folgenden Ereignisse:

- `SnapshotEvent`

RParetoAnalysis Diese Ausgabestrategie ist direkt von RAnalysis abgeleitet. Sie erzeugt aus den Daten Paretomenge- und front, im Gegensatz von ParetoAnalysis wird die Berechnung dieser allerdings nicht im Programm selbst, sondern von R durchgeführt.
Reagiert auf die folgenden Ereignisse:

- `SnapshotEvent`

4.7.4 Kapselung der Ausgabetypen

In der Praxis ist es möglich, dass verschiedene Ausgabestrategien zwar verschiedene Ausgaben generieren, die Art und Weise der Ausgabe aber die gleiche ist, zum Beispiel das Schreiben in eine Datenbank oder eine Textdatei. Damit nun nicht jede Strategie die Ausgabe manuell erzeugen muss, wurde folgendes Konzept erarbeitet. Strategien erzeugen die Ausgabe und verpacken sie in sog. Datenpakete. Ein solches existiert für alle gängigen Datentypen. Desweiteren existiert ein Datenpaket, das Binärdateien aufnehmen kann. Die Strategie erzeugt dann eine Zuordnung, die Feldnamen auf Datenpakete abbildet (also beispielsweise pro Gen oder Fitnesswert ein Datenpaket vom Typ Double, für die Paretomengen oder -fronten jeweils ein Datenpaket für Binärdateien usw.) Diese Zuordnung wird dann an den Ausgabetypen übergeben, der die eigentliche Ausgabe durchführt. Er entscheidet, wie die Pakete ausgegeben werden. Hierzu wurde das Interface `OutputType` erzeugt, die die Methoden `initialize`, `append` und `close` besitzt. Mittels `initialize` wird der Ausgabetyp beim Ausgabemanager registriert (hierzu stellt dieser die Methode `addOutputType` zur Verfügung), außerdem werden die für die Initialisierung des Ausgabetypen nötigen Operationen durchgeführt, also etwa eine Datei geöffnet oder die Verbindung zu einer Datenbank hergestellt. Falls ein `CleanEvent` erzeugt wird, ruft der Ausgabemanager die `close`-Methode jedes Ausgabetypen auf, der dann wiederum die nötigen Operationen durchführt, um die Ausgabe zu beenden. Die eigentliche Ausgabearbeit führt dann die Methode `append` durch, die eine Zuordnung von Namen zu Datenpaketen bekommt und diese dann entsprechend verarbeitet, also zum Beispiel eine Tabelle in einer Datenbank mit den passenden Feldern anlegt und die Daten einträgt, oder die Pakete sequenziell nacheinander ausgibt, usw. In der Konfigurationsdatei ppoSystemConfig.xml können den

verschiedenen Ausgabestrategien ihre Ausgabetypen zugewiesen werden. Dieses Vorgehen ermöglicht auch in gewissen Grenzen, Ausgabetypen je nach Belieben dynamisch umzuschalten, so zum Beispiel bei der Ausgabestrategie PreyValues, die die Datenwerte der Beute auf dem Torus ausgibt. Ob diese in eine Textdatei oder in eine Datenbank geschrieben werden, ist nicht relevant. Theoretisch wäre es mit diesem Konzept auch möglich, eine TEX-Datei zu erzeugen, die die Daten als Tabellen enthält. Die Grenzen sind natürlich erreicht, wenn die Strategie einen bestimmen Ausgabetypen erwartet. Als Beispiel kann die R-Analyse ParetoAnalysis genannt werden. Dort werden Paretomengen und -fronten in der Datenbank abgelegt. Ein Umschalten des Ausgabetypen, beispielsweise auf Textdateien, macht dann keinen Sinn.
Derzeit sind die folgenden Ausgabetypen vorhanden:

CommonDatabaseOutput Stellt die Schnittstelle zu einer SQL-Datenbank dar.

StructuredTextOutput Schreibt Daten in tabellarischer Form in eine Textdatei.

4.7.5 Visualisierung als Spezialfall der Ausgabe

Die Visualisierung ist nun als Spezialfall der Ausgabe zu verstehen. Sogenannte Widgets, welche die Beobachtung des Systemzustands während der Simulation ermöglichen, sind als `OutputType` implementiert. Im Rahmen der Methoden `init` und `close` werden hierzu Swing-Fenster geöffnet und geschlossen. Die Methode `append` versorgt das Widget mit aktuellen Daten, die dann für die Visualisierung aufgearbeitet werden. Diese Widgets wurden implementiert:

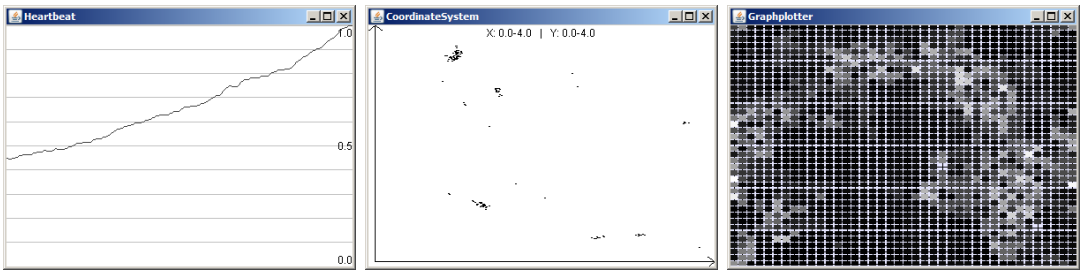

Abbildung 14: Beispiele zu HeartbeatWidget, CoordinateSystemWidget und GraphPlotterWidget

HeartbeatWidget Trägt Doublewerte über die Zeit auf. Wie bei einem Oszilloskop, das einen Herzschlag anzeigt, fahren dabei die neuen Werte links hinein und verdrängen mit der Zeit die alten Werte. Zur Initialisierung erhält das Widget Informationen zur Skalierung von Zeit- und Wert-Achse, sowie die Namen der Schlüssel, die zu überwachen sind.

CoordinateSystemWidget Visualisiert Double-Wertpaare in einem Koordinatensystem. Zur Initialisierung erhält das Widget Informationen zur Achsenskalierung und einen Prefixstring pro Achse. Wenn beim Aufruf von `append` ein Suffix mit beiden Prefixen als Schlüssel vorkommt (wie beim Suffix „12“, falls „a“ bzw. „bc“ die Prefixe sind und die Schlüssel „a12“ und „bc12“ enthalten sind), wird dies als Wertepaar interpretiert.

GraphPlotterWidget Dient der Visualisierung von Positions- und Individuenmetriken und deren räumlicher Verteilung. Dabei werden den Positionen 2D-Koordinaten zugewiesen. Das Widget passt sich automatisch an die Boundingbox dieser Koordinaten an. Zur Initialisierung werden verschiedene Informationen über die Metrik benötigt, die visualisiert werden soll.

MetricTableOutput Bewerkstelligt eine sich aktualisierende tabellarische Ausgabe der Daten.

5 Auswertung, Tests und Analyse

Im Folgenden werden Ergebnisse, die mit Hilfe der im Kapitel 4.3 vorgestellten Testprobleme auf dem Framework erzeugt wurden, vorgestellt. Hierzu wurden verschiedene Einstellungen der Parameter getestet und die Auswirkungen auf die Testprobleme beobachtet.

5.1 BorderMetric zur Quantifizierung von Verklumpungseffekten

Motivation Vergleicht man Räuber-Beute-Modelle mit alternativen Ansätzen zur Lösung mehrkriterieller Optimierungsprobleme, so fällt eine Eigenart sofort ins Auge: Lokalität. Es ist nur sinnvoll zu beleuchten, in welcher Weise die Nachbarschaft zweier Individuen in der strukturierten Population deren Ähnlichkeit impliziert. Da Nachbarschaft in anderen Ansätzen kein so zentrales Konzept darstellt, fehlten hier Werkzeuge zur Bewertung. Darum ist im Rahmen der Projektgruppe die sogenannte BorderMetric entstanden, die im Folgenden vorgestellt wird.

Definition Sei $G = (V, E)$ ein Graph und $c : V \mapsto \{1, \ldots, m\}$ ein Knotenclustering mit m Clustern. Dabei enthalte G mindestens zwei Knoten und eine Kante. Wir interessieren uns für Kanten, deren Endpunkte unterschiedlichen Clustern angehören. Solche Kanten nennen wir Grenzen (Borders) und setzen $B := \{(u, v) \in E | c(u) \neq c(v)\}$. Um nun den Anteil der Grenzen zu erfassen, definieren wir

$$\textsc{BorderProb}(G, c) := \frac{|B|}{|E|}$$

Dies entspricht der Wahrscheinlichkeit, bei uniform zufälliger Auswahl einer Kante eine Grenze vorzufinden.
Man könnte meinen, diese Metrik sei gut normiert. Tatsächlich hängt der Grenzanteil stark davon ab, wie viele Knoten den jeweiligen Clustern angehören. Wenn viele Knoten im selben Cluster sind, würden wir wenige Grenzen erwarten. Bei einem ausgeglichenen Clustering erwarten wir viele Grenzen. Deshalb stellen wir uns folgende Frage: Angenommen, wir würden die Häufigkeiten der Cluster beibehalten, die Zuordnung aber zufällig auf den Knoten umverteilen, wobei jede Umverteilung gleich wahrscheinlich ist - welchen Borderanteil würden wir dann erwarten? Eine uniform zufällig gewählte Kante ist offenbar genau dann eine Border, wenn beide Endpunkte zufällig aus unterschiedlichen Clustern gewählt wurden. Dieses passiert mit Wahrscheinlichkeit

$$\begin{aligned}\textsc{MixedBorderProb}(G, c) &:= \frac{|\{(u, v) \in V \times V | u \neq v \wedge c(u) \neq c(v)\}|}{|\{(u, v) \in V \times V | u \neq v\}|} \\ &= 1 - \sum_{i=1}^{m} \binom{|c^{-1}(i)|}{2} / \binom{|V|}{2}\end{aligned}$$

Dabei bezeichnet der erste Term den Anteil der Knotenpaare mit unterschiedlichen Clusterzuordnungen an Knotenpaaren generell. Der zweite Term ist die Gegenwahrscheinlichkeit dazu, zwei Knoten aus dem selben Cluster auszuwählen. Dabei ist $|c^{-1}(i)|$ gemäß der üblichen Notation die Anzahl der Knoten im Cluster i.
Die BorderMetric setzt nun die tatsächliche Grenzwahrscheinlichkeit BorderProb mit der erwarteten Grenzwahrscheinlichkeit bei zufälliger Durchmischung in Beziehung:

$$\textsc{BorderMetric}(G, c) := \begin{cases} \frac{\textsc{BorderProb}(G,c)}{\textsc{MixedBorderProb}(G,c)} & \text{falls } \textsc{MixedBorderProb}(G, c) \neq 0 \\ & \quad \text{und } |E| \geq 1 \text{ und } |V| \geq 2 \\ 1 & \text{sonst} \end{cases}$$

Offensichtlich kann die BorderMetric bei gegebenem Clustering in Zeit $O(|V|+|E|+m)$ berechnet werden. Anschaulich misst sie, wie überdurschnittlich oft Knoten aus unterschiedlichen Clustern benachbart sind. Ist die BorderMetric über 1, so haben wir eine Tendenz zu vielen Grenzen. In der Simulation von Räuber-Beute-Modellen ist oft das Gegenteil der Fall: Knoten aus gleichen Clustern neigen dazu, zusammenzuklumpen und die BorderMetric sinkt unter 1.

Beispiele Als Beispiele betrachten wir für gerade n einen zweidimensionalen Torus der Größe $n \times n$. Wir untersuchen den Wert der BorderMetric von zwei sehr extremen Clusterings, die jeweils zwei Cluster mit $z := \frac{1}{2}n^2$ Knoten besitzen. Damit ist der Grenzanteil einer durchschnittlichen Durchmischung schon festgelegt und ergibt sich zu $\text{MixedBorderProb} = 1 - 2\binom{z}{2}/\binom{2z}{2} = 1 - 2z(z-1)/((2z)(2z-1)) \approx 1/2$. Bei gleich großen Clustern und einer zufälligen Zuteilung erwarten wir natürlich mit einer Wahrscheinlichkeit von etwas über $\frac{1}{2}$ einen Nachbarn des anderen Clusters. Bei einem Schachbrettclustering sind alle Kanten Grenzen und es gilt $\text{BorderProb} = 1$. Damit ist die BorderMetric wenig unter 2 und wir haben fast doppelt so viele Grenzen wie bei durchschnittlicher Durchmischung.
Für ein zweites Clustering teilen wir den Torus horizontal in zwei Hälften. Offensichtlich erhalten wir zwei Gebiete, die an $2n$ Grenzen (links und rechts) aufeinandertreffen. Da der Torus $2n^2$ Kanten besitzt, ist $\text{BorderProb} = \frac{1}{n}$. Die BorderMetric liegt damit wenig unter $\frac{2}{n}$. Eine durschnittliche Durchmischung hat wesentlich mehr Grenzen als diese.

Beutegraph In Räuber-Beute-Modellen wollen wir im Allgemeinen kein Knotenclustering evaluieren, sondern ein Beuteclustering. Falls nicht auf jeder Position genau eine Beute sitzt, sind also Anpassungen nötig. Es erweist sich als sinnvoll, die BorderMetric in diesem Fall auf einem Beutegraphen auszuwerten. Dabei entsprechen die Knoten der Beute und wir wählen eine Kante zwischen zwei Beuteindividuen genau dann, wenn die Individuen verschieden sind und auf der gleichen oder direkt benachbarten Positionen sitzen.

Stabile BorderMetric Nun definieren wir die stabile BorderMetric als Kennzahl einer gesamten Optimierung. Sei $\text{BorderMetric}(t)$ eine Funktion, die zu jedem Zeitschritt einer Optimierung eine BorderMetric auswertet. Dann ist

$$\begin{aligned}\text{StableBorderMetric}(t_{end}) := \text{Median}(&\text{BorderMetric}(0),\\ &\ldots,\\ &\text{BorderMetric}(t_{end}))\end{aligned}$$

Der grundsätzliche Gedanke hinter der StableBorderMetric ist, Randeffekte in der Initialisierungsphase der Optimierung und eventuell auftretende Oszillationen herauszumitteln und insgesamt eine stabile Kennzahl für die gesamte Optimierung zu erhalten. Alternativ würde sich hier das geometrische Mittel anbieten. Dabei sollte t_{end} hinreichend groß gewählt werden, sodass

$$\text{StableBorderMetric}(t_{end}) \approx \lim_{t_{veryend} \to \infty} \text{StableBorderMetric}(t_{veryend})$$

Wir vermuten stark, dass in vernünftigen Szenarien ein solcher Grenzwert existiert. Es lassen sich jedoch leicht zeitabhängige Clusterzuweisung konstruieren, bei welchen dies nicht der Fall ist. In solch einem Fall macht es wenig Sinn, eine stabile BorderMetric zu betrachten.

5.2 Analyse der Multi-Sphere Funktion

Es wurden eine Reihe von Parametern variiert, um zu untersuchen, wie sich das Multisphere-Problem mit einem Räuber-Beute-Modell verhält. Zur Erinnerung: Der Lösungsraum besteht aus

Individuen $(x, y) \in [-10, 10]^2$, zu optimieren sind die Funktionen

$$f_1 : (x, y) \mapsto x^2 + y^2$$
$$f_2 : (x, y) \mapsto (x - 2)^2 + y^2$$

Alle Ergebnisprotokolle zu den durchgeführten Versuchen finden sich im Anhang. Als Torus wurde hauptsächlich ein doppelt verketteter Torus der Größe 40×40 verwendet. Bei Verwendung eines kleineren bzw. größeren Torus ändert sich nicht das Verhalten, es ändern sich u.U. lediglich die Anzahl der Iterationen, bis gewisse Eigenschaften und Verhaltensmuster zu Tage treten.

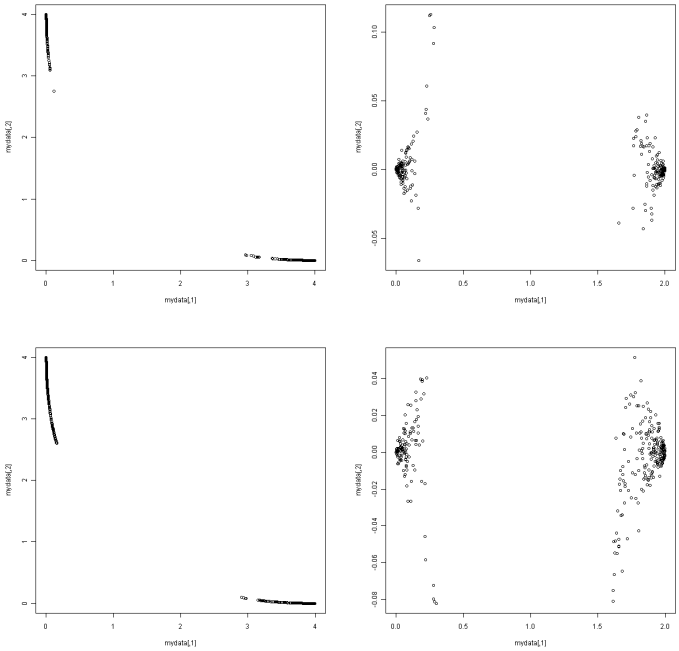

Abbildung 15: Beispiel für die Paretofronten und -mengen zweier ausgewählter Läufe nach jeweils 250000 Iterationen. Es wurde je ein Räuber pro Kriterium verwendet. Der Bewegungsradius beträgt $r = 1$, die Mutationsschrittweite $\sigma = 0,02$.

Mit den Standardeinstellungen (Zufällig gleichverteilte Bewegung mit Schrittweite 1, Standard-Mutationsoperator, der jedem neu erzeugten Individuum einen Vektor aufaddiert, der gemäß Gauß-Verteilung mit den Parametern $\mu = 0$ und $\sigma = 0,02$ gewählt wurde) stellt sich ein äußerst unbefriedigendes Langzeitverhalten ein. Es werden hauptsächlich Lösungen gefunden, die in der Nähe der Extremallösungen $(0, 0)$ und $(2, 0)$ liegen. (Diese stellen die Enden der Paretofront dar, und werden auf $(0, 4)$ bzw. $(4, 0)$ abgebildet.) Man erkennt, dass zunächst die Paretofront aufgebaut wird, nach einigen tausend Iterationen bilden sich „Löcher“ in der Paretofront, schließlich degeneriert die Paretofront, das heißt, im Lösungsraum entstehen nur Lösungen nahe der Extremallösungen. Es stellt sich ebenfalls heraus, dass die Paretofront auch dann nicht gleichmäßig approximiert wird, wenn dies der Fall zu sein scheint. (Diese Erkenntnis kann durch Betrachtung der durch die Paretomenge induzierte Verteilung der Punkte im Lösungsraum gewonnen werden.) Ab einer gewissen Iterationszahl fallen im Lösungsraum Verklumpungen um die Lösungen $(0, 0)$ und $(2, 0)$ auf. Auch zeigt sich, dass die Punkte trichterförmig angeordnet sind (vgl. auch Abbildung 15).
Das beschriebene Langzeitverhalten kann leicht begründet werden: Haben sich genügend Lösungen in der Nähe der Extremallösungen gesammelt, werden bei der Rekombination wieder Lösungen entstehen, die in der Nähe der Extremallösungen angesiedelt sind. Es liegt der Verdacht nahe, dass dieser Effekt insbesondere auftritt, wenn die Mutationsschrittweite klein gewählt wird (mehr dazu später.) Dann ist die Wahrscheinlichkeit, dass die bei einer Rekombination erzeugte neue Lösung

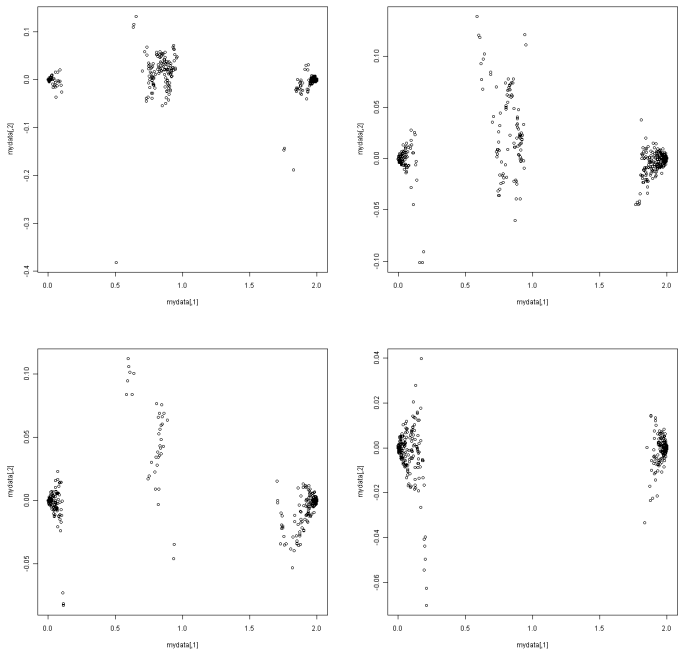

Abbildung 16: Beispiel für die Paretomengen eines ausgewählten Laufs nach 100000 (o.l.), 150000 (o.r.), 200000 (u.l.) und 250000 (u.r.) Iterationen. Es wurde je ein Räuber pro Kriterium verwendet. Der Bewegungsradius beträgt $r = 1$, die Mutationsschrittweite $\sigma = 0,01$.

mit anschließender Mutation nicht in der Nähe der Extremallösungen liegt, sehr gering. Tatsächlich bestätigten Experimente für $\sigma = 0,01$ diese Vermutung. Weiterhin ist anzumerken, dass selbst im Fall, dass ein neues Individuum ungewöhnlich weit von den Extremallösungen entfernt liegt, viele Lösungen nahe der Extremallösungen liegen. Wird die erzeugte Lösung dann irgendwann wieder entfernt (weil ein Räuber das betreffende Individuum gefressen hat), dann wird dieses mit einer hohen Wahrscheinlichkeit durch ein Individuum ersetzt, welches nah am Rand liegt. Das beschriebene Phänomen wird also immer deutlicher auftreten.
Um die Auswirkungen einer noch kleineren Mutationsschrittweite zu erkennen, wurden wie bereits erwähnt weitere Versuche mit $\sigma = 0,01$ durchgeführt (Abbildung 16). Natürlich werden deutlich mehr Iterationen benötigt, bis Änderungen sichtbar werden. Es zeigt sich, dass sich im Lösungsraum Verklumpungen bilden, die auch durchaus nicht in der Nähe von $(0,0)$ bzw. $(2,0)$ liegen. Diese sind aber nur temporär verhanden; durch die kleine Mutationswahrscheinlichkeit dauert es entsprechend lange, bis die entsprechenden Individuen entfernt werden. Nach etwa 250000 Iterationen zeigt sich dann wieder das altbekannte Verhalten: Es sind nahezu nur noch Lösungen nahe $(0,0)$ bzw. $(2,0)$ vorhanden. Damit bestätigt sich die Hypothese, dass das problematische Langzeitverhalten mit kleineren Mutationsschrittweiten nicht gelöst werden kann.
Es stellt sich die Frage, ob größere Mutationswahrscheinlichkeiten das Problem lösen können. Um dies zu klären, wurden weitere Experimente mit den Parametern $\sigma = 0,03$ und $\sigma = 0,04$ durchgeführt. Auf den ersten Blick sahen die Ergebnisse tatsächlich vielversprechend aus. Bei Betrachtung der durch die Paretomenge gegebenen Verteilung der Lösungen im Lösungsraum zeigte sich dann aber, dass sich wieder die Punkte nahe $(0,0)$ und $(2,0)$ häuften. Je größer natürlich die Mutationsschrittweite gewählt wird, desto stärker wird die Mutation und desto größer streuen dann die neu erzeugten Individuen. Ihr Anteil ist allerdings gering. Daher ist die Paretofront auch nicht gleichmäßig approximiert. Damit steht fest: Allein durch Anpassung der Mutationsschrittweite kann das problematische Langzeitverhalten nicht entfernt, sondern nur abgeschwächt werden. Für weitere Untersuchungen wurde zunächst wieder $\sigma = 0,02$ gesetzt, um die Ergebnisse besser untersuchen zu können.

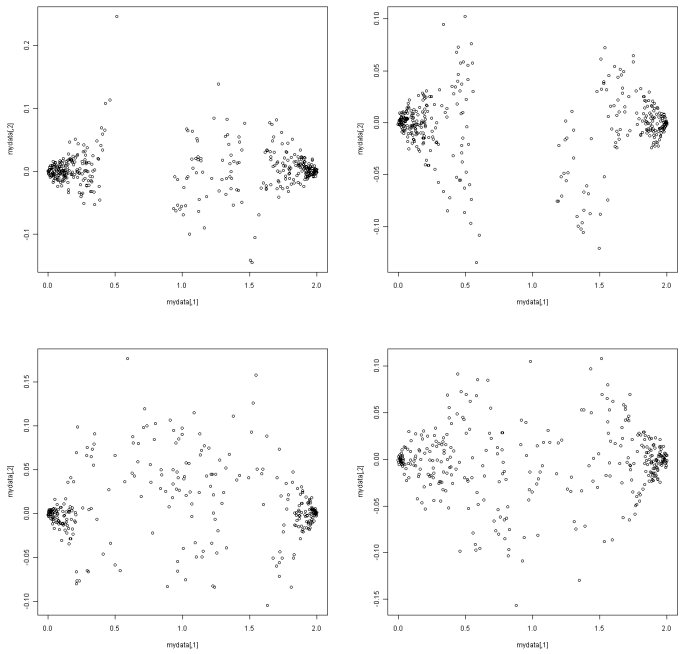

Abbildung 17: Beispiel für die Paretomengen ausgewählter Läufe nach 350000 Iterationen. Die Mutationsschrittweiten betragen $\sigma = 0,03$ (oben) bzw. $\sigma = 0,04$ (unten).

Als nächstes wurde die Bewegung des Räubers variiert. Standardmäßig bewegt sich der Räuber über den Torus, indem eine neue Position mit Abstand $r = 1$ zufällig gleichverteilt auswählt und sich dort hinbewegt. Dieser Abstand wurde nun sukzessive erhöht. Versuche wurden für $r \in \{2, 3, 4\}$ durchgeführt. Das Ergebnis ist mehr als überraschend: Für $r = 2$ zeigen sich schon deutlich bessere Ergebnisse als bisher gesehen. Allerdings muss auch gesagt werden, dass die Ergebnisse trotzdem nicht immer optimal waren: Zwar existierten Läufe, bei denen viele Lösungen gefunden wurden, die beide Kriterien mehr oder weniger erfüllen. Lösungen in der Nähe von $(0, 0)$ bzw. $(2, 0)$ waren die Ausnahme. Allerdings existierten auch Läufe, bei denen sich nach einiger Zeit wieder das altbekannte Verhalten zeigte und sich viele Punkte in der Umgebung der Extremallösungen sammelten (vgl. Abbildung 18). Deutlich besser dagegen fiel das Verhalten für $r = 3$ bzw. $r = 4$ aus. Es werden keine Lösungen nahe $(0, 0)$ bzw. $(2, 0)$ gefunden, sondern die Lösungen sind (relativ) gut im Lösungsraum verteilt. Dieses Verhalten zeigte sich auch nicht nur in einigen wenigen, sondern in allen zehn durchgeführten Testläufen. Die besten Ergebnisse wurden für $r = 3$ erzielt. Für $r = 4$ zeigt eine Betrachtung der Paretofronten, dass fast alle Lösungen **sehr** nahe an der guten Kompromisslösung $(1, 0)$ lagen. Daher ist die Diversität gering. Es muss allerdings beachtet werden, dass hier jeweils die Anzahl der Räuber pro Kriterium gleich groß waren. Auf den Fall, dass unterschiedliche Anzahlen von Räubern (um die Kriterien unterschiedlich stark zu gewichten) verwendet werden, wird später eingegangen.
Sollen jedoch beide Funktionen etwa gleich stark optimiert werden, dann scheint sich mit der Wahl von $\sigma = 0,02$ und $r = 3$ ein ziemlich vielversprechender Ansatz gefunden zu haben. Ein gutes Ergebnis zeigte sich häufig nach etwa 200000 Iterationen. Dieses blieb in den weiteren (150000) Iterationen dann auch stabil. Allerdings muss beachtet werden, dass sich die besten Kompromisslösungen (also Lösungen, die beide Kriteren gleich gut erfüllen) bei $(1, 0)$ liegen. In den Testläufen glichen die Paretomengen zwar Punktwolken, deren Mittelpunkt lag aber nicht immer in der Nähe von $(1, 0)$. Stattdessen wurden Abweichungen in beide Richtungen ermittelt. Daher wurden teilweise Kriterium 1 und teilweise auch Kriterium 2 stärker gewichtet. Da die Anzahl der Läufe, die nach links bzw. rechts abwichen aber ungefähr gleich groß waren, kann diese Beobachtung aber ignoriert werden.

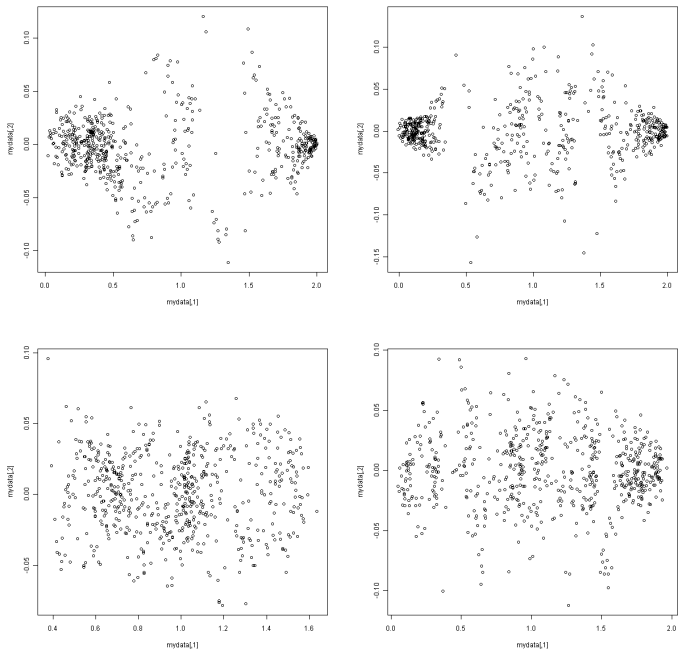

Abbildung 18: Beispiel für die Paretomengen ausgewählter Läufe nach 350000 Iterationen. Die Mutationsschrittweiten betragen $\sigma = 0,02$, als Abstand für die Bewegung wurde $r = 2$ gewählt. Oben sind schlechte, unten bessere Ergebnisse zu sehen.

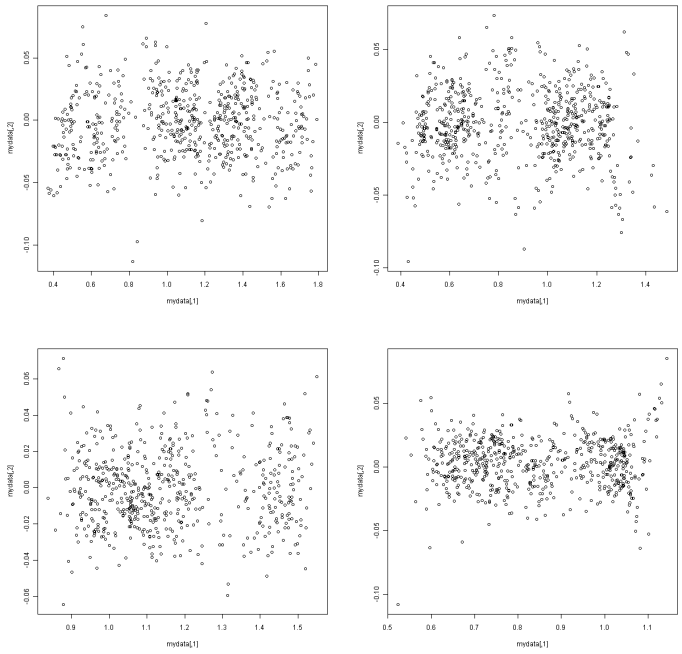

Abbildung 19: Beispiel für die Paretomengen ausgewählter Läufe nach 350000 Iterationen. Die Mutationsschrittweiten betragen $\sigma = 0,02$, als Abstand für die Bewegung wurden $r = 3$ (oben) bzw. $r = 4$ gewählt.

Als letztes wurde untersucht, was passiert, wenn die zwei Kriterien unterschiedlich stark gewichtet werden. Hierzu wurden drei Räuber eingeführt, zwei Räuber, die nach f_1 optimieren und einer, der f_2 optimiert. Hier zeigen sich sehr unbefriedigende Ergebnisse: Es werden nahezu nur Lösungen

gefunden, die nahe $(0, 0)$ liegen (also der Lösung, die f_1 minimiert). Auch die im symmetrischen Fall hilfreiche Erhöhung des Abstandes r bei der Bewegung half nicht. Tatsächlich zeigt Abbildung 20, dass sich das Problem nur verstärkt. Es ist also grundsätzlich sinnvoll, die Anzahl der Räuber pro Kriterium bei Multisphere gleich groß zu wählen, selbst wenn ein bestimmtes Kriterium bevorzugt werden soll. Ansonsten werden auf lange Zeit gesehen wieder nur Lösungen gefunden, die der Extremallösung nahe sind. Die gleichen Überlegungen gelten natürlich auch für andere Aufteilungen der Räuber, und insbesondere auch, falls die Räuber, die nach f_2 optimieren, in der Überzahl sind.

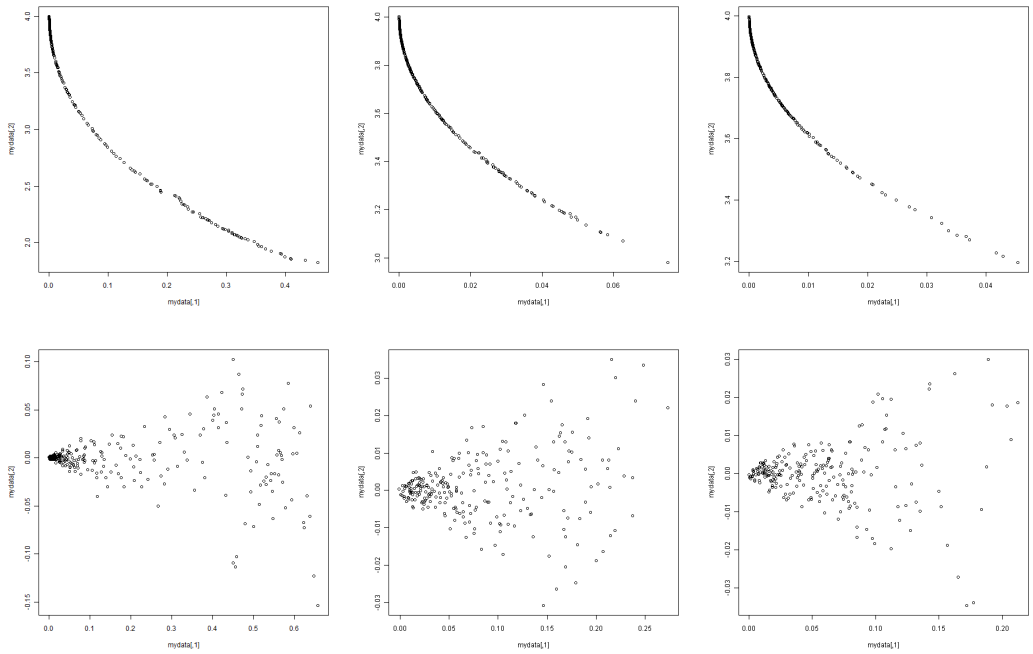

Abbildung 20: Beispiel für die Paretofronten und -mengen ausgewählter Läufe nach 350000 Iterationen. Es wurden zwei Räuber für f_1 und einer für f_2 verwendet. Die Mutationsschrittweite beträgt $\sigma = 0,02$, als Abstand für die Bewegung wurden $r = 1$ (links), $r = 2$ (Mitte) sowie $r = 3$ (rechts) gewählt.

Fazit: Das Räuber-Beute-Modell kann verwendet werden, um das symmetrische Multisphere-Problem zu approximieren. Eine Anpassung der Mutationsschrittweite allein hilft nicht, das Problem der Häufung von Punkten nahe der Extremallösungen zu lösen, eine Anpassung des Abstandes bei der Bewegung der Räuber sehr wohl. Das asymmetrische Multisphere-Problem, bei dem die Kriterien unterschiedlich stark gewichtet werden sollen, können mit dem Räuber-Beute-Modell nicht gut approximiert werden. Es bilden sich fast nur Lösungen, die sich in der Nähe der Extremallösungen $(0, 0)$ bzw. $(2, 0)$ (je nachdem, welches Kriterium stärker optimiert werden soll) befinden.

5.3 Analyse der Kursawe Funktion

Das von F. Kursawe vorgestellte Testproblem ist ein Vektoroptimierungsproblem. Mathematisch lässt sich das Problem folgendermaßen beschreiben:

$$
\begin{aligned}
f_1(\vec{x}) &= \sum_{i=1}^{n-1}(-10 \cdot \exp(-0.2 \cdot \sqrt{x_i^2 + x_{i+1}^2})) \\
f_2(\vec{x}) &= \sum_{i=1}^{n}(\mid x_i \mid^a + 5\sin(x_i^b)) \\
\text{Wobei } \vec{x} &= \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{R}^n, a = 0.6 \text{ und } b = 3 \text{ gelten.}
\end{aligned}
$$

Das Problem besteht aus einem konkaven und einem konvexen Teil. Beide Teile gleichermaßen gut zu approximieren gilt als schwierig, da durch Mutation meist nur ein Teil (entweder der konkave oder der konvexe Teil) gefunden wird. Wird hingegen nur Rekombination verwendet, werden beide Teile gefunden, die Extrempunkte hingegen, die nur ein Kriterium besonders gut optimieren, kommen in der Lösungsmenge nicht vor.
In dem Testdurchlauf wurde das Problem auf einem Torus mit 1600 Individuen angewandt, wobei die Individuen gleichverteilt initialisiert wurden. Je Zielkriterium wurde ein Räuber erzeugt, der sich uniform zufällig auf dem Torus bewegt. Der Räuber hat durch die elitäre Consumption immer das schlechteste Individuum zum „fressen“ ausgewählt. Neue Individuen wurden durch die `WeightedMeanValueDAChromosomeReproduction` erzeugt, das heißt, dass aus den bestehenden Eltern der Durchschnitt der Chromosome ermittelt wurde und als neue Chromosome dem neuen Individuum gegeben wurden. Eine Mutation wurde nicht eingesetzt.
Alle 20 Sekunden wurde das Bild der Population gespeichert und ist in den folgenden Abbildungen zu sehen. Die Punkte stellen die Paretomenge des zweidimensionalen Problems dar:

Abbildung 21: links: Initiale Paretofront, rechts: Paretofront nach 20 Sekunden

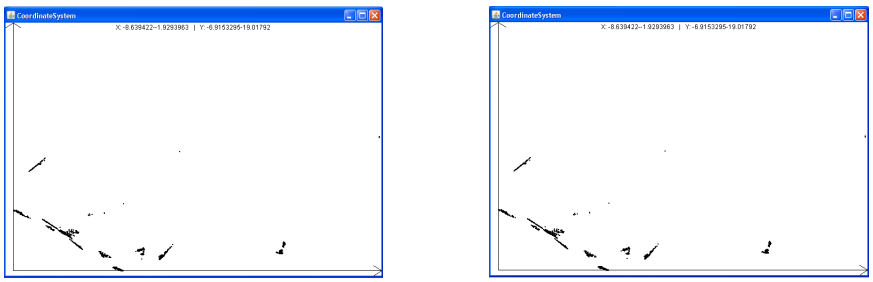

Abbildung 22: links: Paretofront nach 40 Sekunden, rechts: Paretofront nach 60 Sekunden

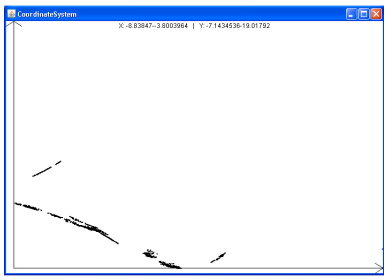

Abbildung 23: Paretofront nach 80 Sekunden

Es ist zu erkennen, dass im Verlauf des Algorithmus die Paretofront gut approximiert wird. Im Testdurchlauf mit Mutation und ohne Rekombination war dieses nicht zu beobachten. Hier wurden wie vorher beschrieben nur kleine Teilbereiche der Lösungsmenge gefunden. Alle Punkte verliefen genau auf der Y-Achse.

5.4 Analyse der P^* Funktion

P* stellt das Problem dar, die konvexe Hülle einer beliebigen Punktemenge zu berechnen. Siehe hierzu auch Seite 4.3.3 . Der Vorteil bei P* ist, dass die Anzahl der Optimierungs-Kriterien erhöht werden kann ohne die Dimensionalität der Ausgabe zu erhöhen. Jeder Punkt der gegebenden Punktmenge stellt ein Optimierungskriterium dar. Somit kann unabhängig von der Anzahl der Punkte und somit unabhängig von der Anzahl der Kriterien, das Ergebnis im 2 bzw. 3-dimendionalem Raum dargestellt werden. Hierdurch kann das Verhalten des Frameworks bei mehr als 3 Optimierungskriterien einfach getestet werden. Im Weiteren werden vier Punkte im 2-dimensionalem Raum betrachtet.
Jedes Beuteindividuum stellt einen Punkt im Raum der betrachteten Punktemenge dar.

Sei P die gegebende Punktemenge zu der die konvexe Hülle berechnet werden soll.
Dann gibt es $|P|$ Fitnessfunktionen. Sei $p_i \in P$. Dann ist die zugehörige Fitnessfunktion der euklidische Abstand zwischen dem Punkt den die Beute darstellt und dem Punkt p_i.
Da reine Mutation nicht zielgerichtet neue Individuen erzeugt, ist das Ergebnis voraussichtlich nicht befriedigend. Die Mutation zieht die Punkte wahrscheinlich zu sehr auseinander, wodurch das Ergebnis zu sehr gestreut ist. Diese Hypothese bestä tigt sich auch, wie im Ergebnisprotokoll zu P* sowohl für 4 als auch für 5 Punkte zu sehen ist.

Wird nur eine intermediä re Rekombination verwendet, die die Eltern zufällig wählt, ist das Ergebnis zwar voraussichtlich besser als bei reiner Mutation, aber durch das zufällige Wählen der Eltern kö nnen Eltern gewählt werden, die fü r unterschiedliche Räuber optimaler sind, so dass ein Nachkommen erzeugt werden kann, der sich eher im Zentrum der konvexen Hülle befindet. Hierduch könnte ein Verklumpen im Mittelpunkt der konvexen Hülle entstehen.

Wird die Simplex-Rekombination mit zwei positiv nach ihrer Fitness selektierten Eltern mit gleicher Gewichtung verwendet (also wieder intermediär), so ist das Ergebnis voraussichtlich gut. Jeder Räuber ersetzt das Beuteindividuum, welches den Punkt mit dem größten Abstand zum Referenzpunkt darstellt, durch ein Beuteindividuum, dessen Punkt näher zum Referenzpunkt des Räubers gezogen wurde.

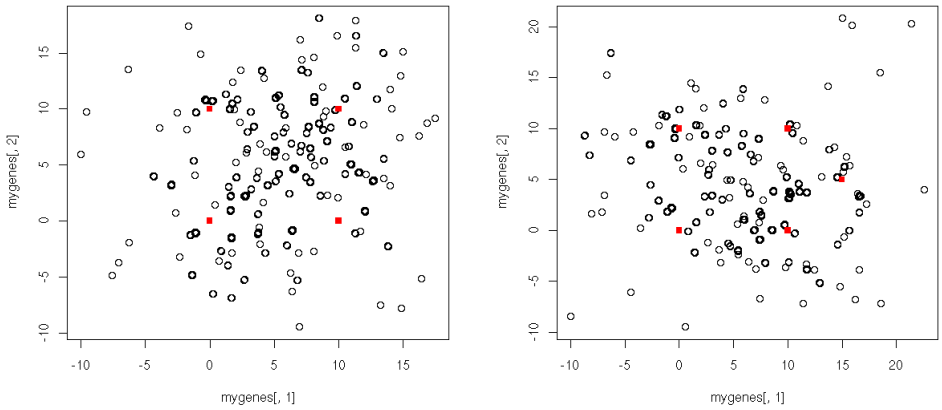

Abbildung 24: 4 und 5 Punkte in P: Punkte, die durch die Beutepopulation nach 10000 Iterationen dargestellt werden

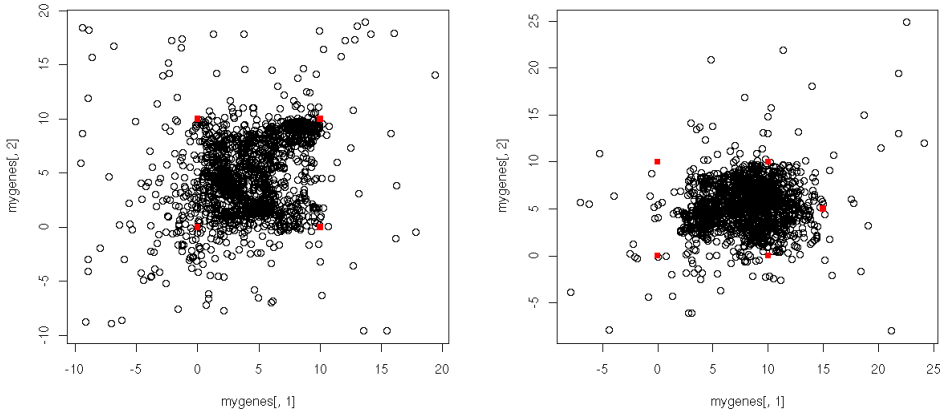

Abbildung 25: 4 und 5 Punkte in P: Punkte, die durch die Beutepopulation nach 10000 Iterationen dargestellt werden

5.5 Analyse der Binh Funktion

Beim 1996 vorgestellten Binh-Problem sollen die Funktionen

$$f_1 : (x, y) \mapsto x^2 + y^2$$
$$f_2 : (x, y) \mapsto (x - 5)^2 + (y - 5)^2$$

minimiert werden. Hierbei sind $-5 \leq x, y \leq 10$. Bei Betrachtung der Funktionen kann man eine starke Ähnlichkeit zum Multisphere-Problem feststellen. Es liegt also die Vermutung nahe, dass sich das Räuber-Beute-Modell, angewendet auf Binh, analog verhält, wie dies für Multisphere der Fall war. Tatsächlich bestätigt sich die Vermutung. Aus diesem Grund existieren im Anhang keine Ergebnisprotokolle, weil sie (wortwörtlich) genauso für Binh aufgebaut wären (einzig und allein die konkreten Werte sind dann etwas anders).
Gearbeitet wurde wieder auf einem 40×40-Torus mit jeweils einem Räuber pro Kriterium. Initial wurde mit Schrittweite $\sigma = 0,02$ mutiert. Hierbei zeigten sich wieder zunächst mehrere „Verklumpungen“ im Lösungsraum, die auch nicht unbedingt in der Nähe der Extremallösungen $(0, 0)$ und $(5, 5)$ lagen. Dieses Verhalten ist völlig analog zu dem Verhalten von Multisphere für $\sigma = 0,01$. Nach etwa 200000 Iterationen waren die Verklumpungen nahezu vollständig verschwunden, mit

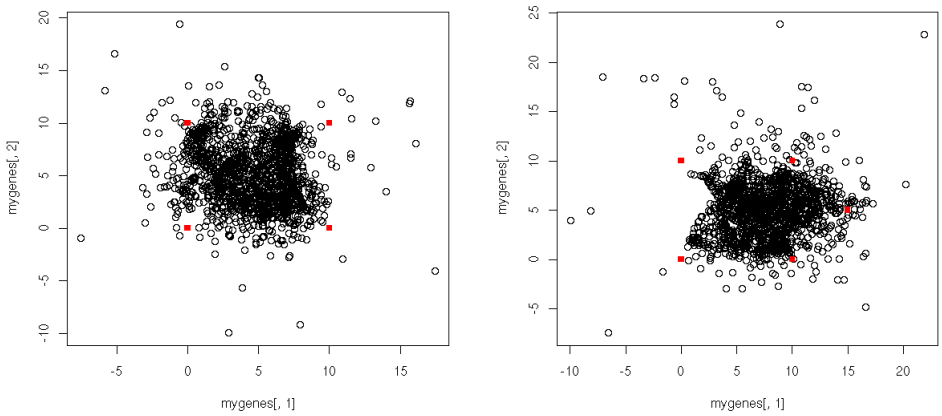

Abbildung 26: 4 und 5 Punkte in P: Punkte, die durch die Beutepopulation nach 10000 Iterationen dargestellt werden

Ausnahme der Punktwolken um $(0, 0)$ und $(5, 5)$. Wie beim Multisphere-Problem führen auch hier höhere Mutationswahrscheinlichkeiten nicht zum gewünschten Ziel (vgl. auch Abbildung 28). Für $\sigma = 1$ ergeben sich zumindest bei manchen Läufen akzeptable Ergebnisse, dennoch ist der Bereich guter Kompromisslösungen $[2, 3] \times [2, 3]$ relativ dünn besetzt. Abgesehen davon rückt bei einer derart riesigen Mutationsschrittweite die eigentliche Rekombination in den Hintergrund.
Als nächstes wurden wieder die Bewegungsschrittweiten angepasst. Hierbei zeigen sich für $r = 2$ durchwachsene Ergebnisse. Zwar existierten Läufe, bei denen sich relative viele gute Kompromisslösungen fanden, auf der anderen Seite gab es aber auch Läufe, bei denen sich viele Individuen um $(0, 0)$ und $(5, 5)$ sammelten. Die besten Ergebnisse wurden wie bei Multisphere auch für $r = 3$ erzielt.
Schließlich wurde das Verhalten für unterschiedliche Anzahlen von Räubern untersucht. Auch hier stimmt das Verhalten mit dem (asymmetrischen) Multisphere-Problem überein. Es wurden zwei Räuber verwendet, die versuchten, den Wert der Fitnessfunktion f_1 zu minimieren sowie einer, der versuchte, den Wert von f_2 zu minimieren. Hierbei zeigte sich, dass nach einiger Zeit nahezu ausschließlich (bis auf sehr wenige Ausnahmen) nur noch Lösungen in der Nähe von $(0, 0)$ (also der Lösung, die f_1 minimiert) existierten. Also ist das Räuber-Beute-Modell auch nicht dafür geeignet, das asymmetrische Binh-Problem zu approximieren.
Fazit: Binh ist Multisphere sehr ähnlich. Daher kann man alle Ergebnisse von Multisphere auf Binh übertragen. Es bilden sich auf lange Zeit gesehen nur Lösungen um $(0, 0)$ bzw. $(5, 5)$. Dieses Problem kann durch eine Variation der Mutationsschrittweite nicht gelöst werden. Gute Ergebnisse wurden erzielt, indem der Einzugsbereich des UniformMovement-Operators auf $r = 3$ erhöht wurde. Zur Approximation des asymmetrischen Binh-Problems ist das Räuber-Beute-Modell nicht geeignet.

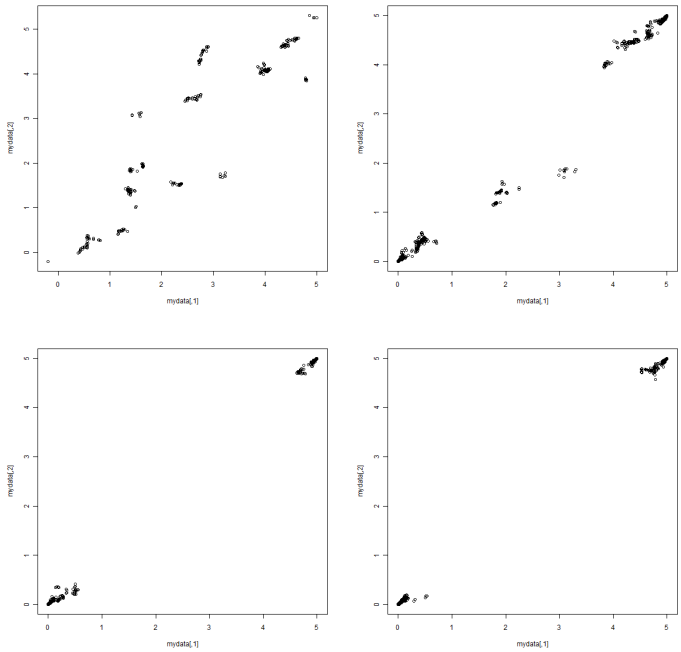

Abbildung 27: Beispiel für die Paretomengen eines ausgewählten Laufs nach 50000 (o.l.), 150000 (o.r.), 250000 (u.l.) und 350000 (u.r.) Iterationen. Es wurde je ein Räuber pro Kriterium verwendet. Der Bewegungsradius beträgt $r = 1$, die Mutationsschrittweite $\sigma = 0,02$.

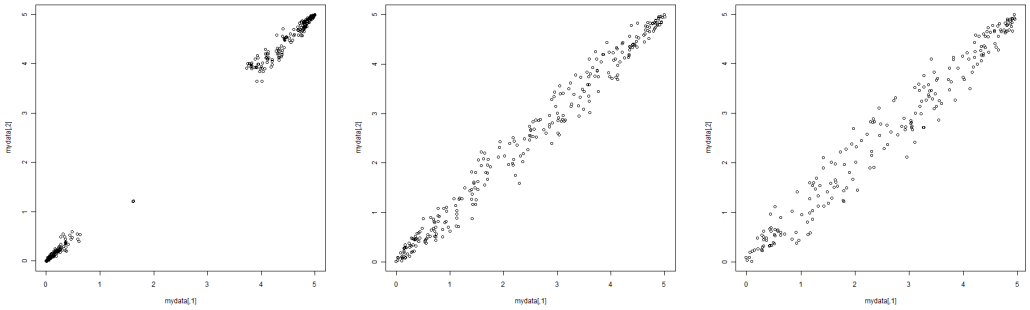

Abbildung 28: Beispiel für die Paretomengen ausgewählter Läufe für die Mutationsschrittweiten $\sigma = 0,05$ (links), $\sigma = 0,5$ (Mitte) sowie $\sigma = 1,0$ (rechts).

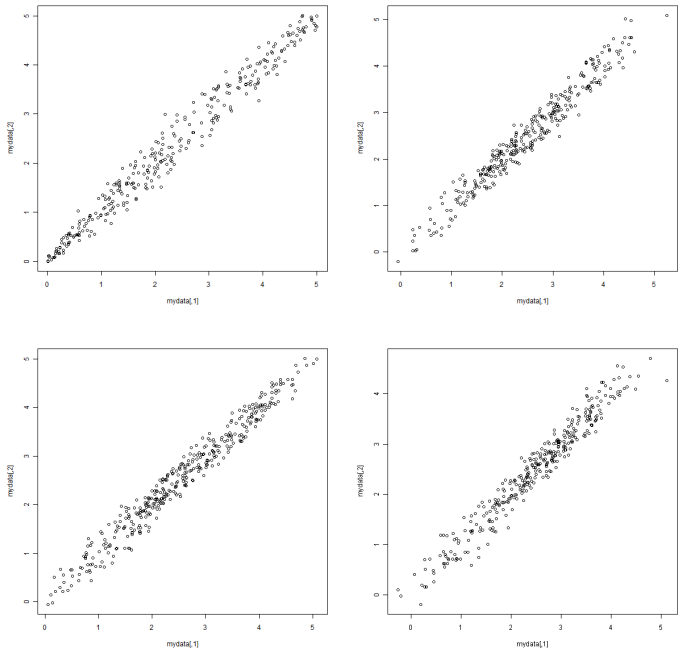

Abbildung 29: Beispiel für die Paretomengen ausgewählter Läufe mit Distanzen $r = 2$ (oben) bzw. $r = 3$ (unten) des UniformMovement-Operators. Links sind jeweils schlechte und rechts jeweils gute Ergebnisse zu sehen. Als Mutationsschrittweite wurde $\sigma = 0,5$ gewählt.

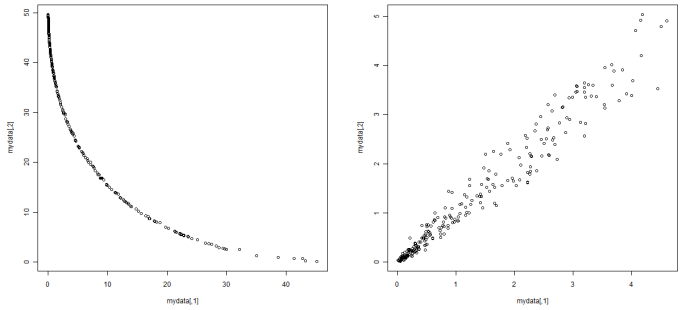

Abbildung 30: Beispiel für die Paretomengen eines ausgewählten Laufs mit zwei Räubern für f_1 sowie einem Räuber für f_2. Als weitere Parameter wurden $\sigma = 0,5$ sowie $r = 1$ gewählt.

5.6 Analyse eines mehrkriteriellen Schedulingproblems

Die allgemeinen Rahmenbedingungen eines jeden Simulationsdurchlaufs unterschieden sich nicht sonderlich. Den grössten Einfluss nahm die Beschaffenheit des Jobshops. Für eine geringe Anzahl an Jobs und somit einer geringen Anzahl an Ausführungsreihenfolgen reichte ein kleiner Torus mit wenigen Räubern aus. Diese Zahlen mussten jedoch für grössere Werte angepasst werden. Untersucht wurde Permutationen über fünf, zehn und zwanzig Jobs der zufällig gewählter Parameter (duedate, penalty und processing time) von Ganzen Zahlen im Intervall von 0 bis 100. Entsprechend belief sich die Grösse der Tori zwischen 64 und 1600 Positionen.

Wie in Kapitel 4.3 erwähnt, war eine grosse Auswahlmöglichkeit an Rekombinationsoperatoren gegeben. Neuere Rekombinationsoperatoren konnten im Vergleich zu den klassischen Operatoren für Permuationen (Crossover und Permutationsmutationen) kaum überzeugten. Es stellte sich heraus, dass bei einer geringen Anzahl an Jobs der Unterschied zwischen einem Crossoveroperator und einer Mutation (beispielsweise Vertauschung zweier Jobs in der Ausführungsreihenfolg), was bei der Arbeitsweise der Crossover nicht weiter verwundert. Ferner konnte bei einer geringen Anzahl an Jobs der Suchraum bereits durch einen sehr kleinen Torus sehr gut abgeckt werden, weswegen die Aufgabe der Räuber wenig mehr umfasste, als die Pareto-dominierten Beutetiere zu entfernen.

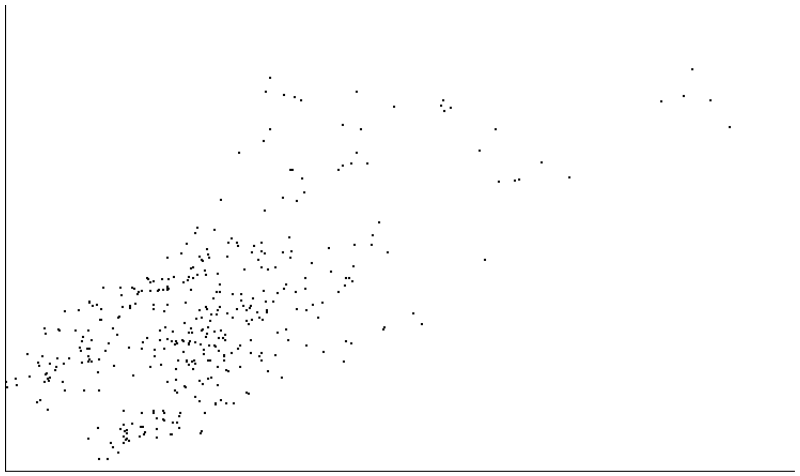

Abbildung 31: Ausschnitt der Punktemenge bei Inversions-Mutations. Iterationen 10000

Interessanter gestalten sich die Durchläufe bei zehn und mehr Permutationen. Hier konnte kein effenzient handhabbarer Torus den Suchraum (bei zehn Jobs schon 3628800 Ausführungsreihenfolgen) vollständig abdecken. Die Mutation alleine erzielte auch keine gezielte Verbesserung, was zur Folge hatte, das eine gute Annährung an die Paretomenge nur durch eine lange Wartezeiten zustande gekommen ist. Crossoveroperatoren brauchten ebenfalls noch eine gewisse Zeit, ehe eine Annährung deutlich wurde, waren aber deutlich schneller als die reine Mutation.

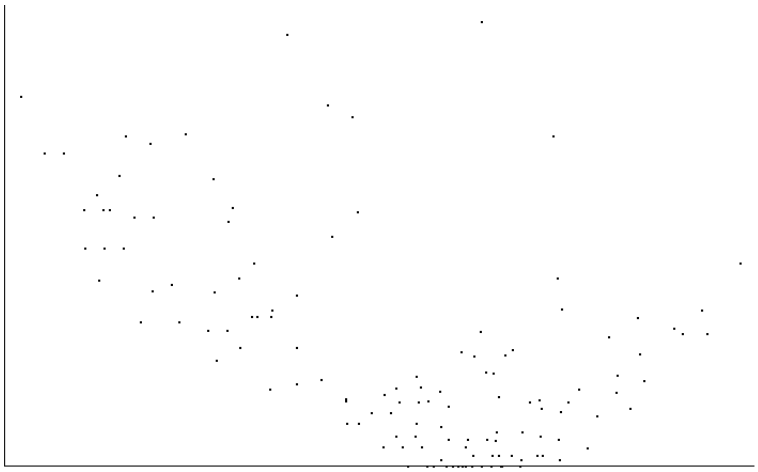

Abbildung 32: Ausschnitt der Punktemenge bei Crossover-Rekombination. Iterationen 10000

A Java Native Interface

Das Java Native Interface, kurz JNI, ist eine standarisierte API die eine Verbindung zwischen Plattform abhängigen Funktionen und Java herstellt. Insbesondere ist es mit JNI möglich auf Funktionen bzw. Methoden zuzugreifen die per C/C++ Biblithek zur Verfügung gestellt werden. Ein Nachteil von JNI ist, dass die Plattformunabhängigkeit verloren geht, da nicht sichergestellt werden kann, dass die genutze Bibliothek auf allen Systemen zur Verfügung steht.
Um eine native Methode in Java nutzen zu können, muss diese auch als `native` deklariert werden. Bevor diese Methode allerdings aufgerufen werden kann, muss die entsprechende Bibliothek durch `System.loadLibrary` geladen werden.

B Erzeugen der libDrmaaJNI.so

Die Beschreibung zur Erzeugung der Bibliothek `libDrmaaJNI.so` bezieht sich sehr auf das verwendete System des IRF-Clusters. Allerdings kann diese Anleitung auch auf anderen Systemen als Hilfe verwendet werden.
Das DRMAA-Java-Binding basiert auf dem von Gridway für ihr System entwickeltes Binding. Allerdings wurde die Klasse `JobTemplateImpl` um einige private Variablen wie Priority, Rerunable, usw. erweitert die auch bei einem Konsolenaufruf von `qsub` an das Cluster übergeben würden. Zu diesen Variablen wurden auch entsprechende get- und set-Methoden in der Klasse angelegt. Zur einfacheren Erstellung der `DrmaaJNI.h` und der `libDrmaaJNI.so` wurde das folgende Ant-Skript (build.xml) angelegt.

```
<project default="cc" basedir=".">
 <target name="javah">
   <javah classpath="bin" outputFile="DrmaaJNI.h" verbose="yes">
     <class name="drmaa.torque.DrmaaJNI" />
   </javah>
 </target>
 <target name="cc" >
   <exec dir="/usr/bin/" executable="/usr/bin/gcc">
     <arg value="-I" />
     <arg value="/usr/lib/jvm/java-1.5.0-sun-1.5.0.14/include" />
     <arg value="-I" />
     <arg value="/usr/lib/jvm/java-1.5.0-sun-1.5.0.14/include/linux" />
```

```
      <arg value="-I" />
      <arg value="/usr/src/torque-2.2.1/src/drmaa/src" />
      <arg value="-L" />
      <arg value="/export/local/lib" />
      <arg value="-shared" />
      <arg value="-Wl,--library-path=/export/local/lib,--library=drmaa" />
      <arg value="-o" />
      <arg value="${basedir}/libDrmaaJNI.so" />
      <arg value="${basedir}/DrmaaJNI.c" />
    </exec>
  </target>
</project>
```

C Nutzung des Frameworks mit Eclipse

Das entwickelte Framework stellt eine Reihe von Klassen und Interfaces zur Verfügung, mit denen ein Räuber-Beute-Modell entwickelt werden kann. Zusätzlich existieren eine Reihe von Standardimplementierungen um den Einstieg zu erleichtern. Auf diese Weise muss der Benutzer nicht sämtliche Interfaces selbst implementieren. Er kann, ausgehend von einem Standardproblem, einzelne Komponenten des Modells manipulieren oder ersetzen, bis es dem gewünschten Modell entspricht.

Zu folgenden Problemen existieren bereits Modelle. Eine genaue Beschreibung dieser Probleme befindet sich in Kapitel 4.3.

Problem:	**Konfigurationsdatei:**
Multi-Sphere	multiSphere.xml
Multi-Sphere	multiSphereWithEnvironment.xml
Kursawe	kursawe.xml
P*	pStar.xml

Für das Problem Multisphere bestehen zwei Konfigurationsdateien. Die zweite Konfiguration veranschaulicht die Benutzung von Environments.

C.0.1 Ein neues Projekt erstellen

Dieser Abschitt beschreibt wie ein neues Projekt angelegt wird und der Benutzer das Framework in sein Projekt integriert. Desweiteren wird eine Standardkonfiguration (Multi-Sphere) eingestellt, mit der eines der vorimplementierten Modelle erstellt wird. Anschließend kann das Projekt direkt gestartet werden.

1. Legen Sie ein neues Projekt in Eclipse an.
2. Legen Sie im Projektverzeichnis die Unterordner *config* und *lib* an. (Dieser Schritt ist nicht unbedingt notwendig, sondern dient lediglich der besseren Übersicht. Theoretisch können die Dateien dieser Ordner auch an anderer Stelle liegen.)
3. Kopieren Sie die jar-Bibliotheken des Frameworks in den Ordner *lib*.
4. Kopieren Sie die Konfigurationsdateien in den Ordner *config*. Da wir als Beispiel Multi-Sphere wählen, müssen die Dateien *multiSphere.xml* und *ppoSystemConfig.xml* kopiert werden.
5. Öffnen Sie die Projekteigenschaften und geben Sie alle jar-Bibliotheken aus dem Ordner *lib* als Librarys an.

6. Öffnen Sie aus dem Menü den Reiter *Run* und wählen Sie den Menüpunkt *Open Run Dialog...*. Legen Sie eine neue Startkonfiguration an. Der Namen kann beliebig gewählt werden.

7. Geben Sie in der neuen Startkonfiguration als main-Klasse die Klasse *de.irf.it.pg527.server.Server* an.

8. Geben Sie in der neuen Startkonfiguration als VM-Argumente die beiden Konfigurationsdateien wie folgt an:

   Windows:

   ```
   -Dppo.xml.file="C:\...\config\multiSphere.xml"
   -Dppo.server.configFile="C:\...\config\ppoSystemConfig.xml"
   ```

   Linux/Mac:

   ```
   -Dppo.xml.file=/home/.../config/multiSphere.xml
   -Dppo.server.configFile=/home/.../config/ppoSystemConfig.xml
   ```

9. Starten Sie die gerade erstellte Konfiguration.

C.0.2 Eine eigene Implementierung schreiben

Dieses Kapitel beschreibt wie eigene Implementierungen von Frameworkkomponenten angelegt werden. Vorraussetzung dafür ist, dass die Bibliotheken des Framekworks (wie im vorherigen Abschnitt) in das Projekt eingebunden wurden.

1. Zunächst muss im Projekt eine geeignete Paketstruktur angelegt werden. Damit das Projekt übersichtlich bleibt wird empfohlen die Paketstruktur des Frameworks zu übernehmen. Dies ist aber nicht zwingend notwendig.

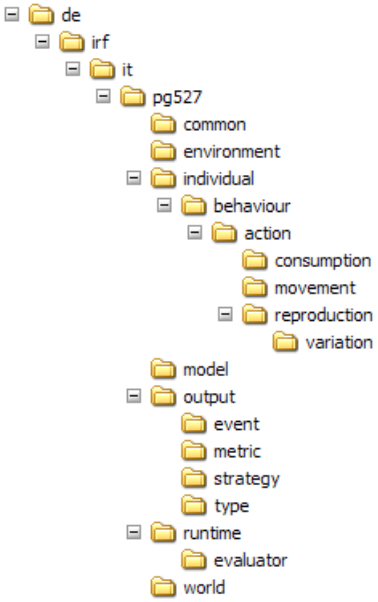

Abbildung 33: Paketstruktur des Frameworks

2. Legen Sie eine neue Klasse im gewünschten Paket an. Der Name der Klasse ist dabei beliebig. Achten Sie darauf, dass das entsprechende Interface bzw. die entsprechende abstrakte Klasse über *implements* bzw. *extends* angegeben wird.

3. Alle Klassen werden durch das Framework per XML erzeugt. Daher muss die Klasse einer Konvention entsprechen, um per XML erzeugt werden zu können. Jedes Attribut der Klasse, das über XML konfigurierbar sein soll, muss über eine Getter- und eine Settermethode zugreifbar sein. Dabei gilt, dass die Methodennamen genau dem Attributname entsprechen müssen.

   Beispiel:

```
// Attribut:
private int size;

// Getter- und Settermethoden:
public int getSize(){...}
public void setSize(int size){...}
```

C.0.3 Eine neue Konfiguration erstellen

In diesem Kapitel wird beschrieben wie eine Konfiguration für ein Räuber-Beute-Modell erstellt wird. Dabei ist zu beachten, dass grundsätzlich zwei Konfigurationsdateien benötigt werden. In der ersten Datei stehen alle Informationen, die für den Ablauf des Frameworks erforderlich sind. Dazu gehören z.B. Angaben wie ein Stopkriterium oder eine Ausgabestrategie. In der zweiten Datei müssen alle Angaben zum eigentlichen Modell enthalten sein. Das umfasst Angaben zum Aufbau der Welt oder die Definitionen von Räuber- und Beuteindividuen.

Zunächst wird eine Konfiguration für das Framework beschrieben. In dieser Datei können Angaben zu folgenden Komponenten des Frameworks enthalten sein:

- Strategie der Engine
- Stopkriterien
- Ausgabestrategien
- Metrikstrategien

Da die Konfiguration mit XML beschrieben wird, muss eine entsprechende Datei angelegt werden. Eine Systemkonfiguration hat folgenden Aufbau:

```
<?xml version="1.0" encoding="UTF-8"?>
<predatorPreyConfig
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="ppoConfig.xsd">

  <server>

  <!-- Beschreibung der vier genannten Komponenten. -->

  </server>
</predatorPreyConfig>
```

An der markierten Stelle sind die Einstellungen der Komponenten anzugeben. Die Strategy der Engine wird über eine Javaklasse bestimmt. Daher ist in der XML-Datei nur diese Klasse anzugeben. Im Beispiel wird die Strategie *SimpleSerialStrategy* verwendet:

```
  <engineStrategy value="de.irf.it.pg527.runtime.SimpleSerialStrategy" />
```

Wenn Sie eine eigene Strategie implementiert haben, könnte die Zeile z.B. so aussehen:

```
<engineStrategy value="myPackage.myStrategy" />
```

Anschliessend werden die Abschnitte für *Stopkriterien*, *Ausgabestrategien* und *Metrikstrategien* angegeben. Hierbei können allerdings auch mehrere Kriterien bzw. Strategien angegeben werden. Daher muss zuerst der entsprechende Abschnitt eingeleitet werden:

```
<stopCriteria>
  <!-- Beschreibung der Stopkriterien -->
</stopCriteria>

<outputStrategies>
  <!-- Beschreibung der Ausgabestrategien -->
</outputStrategies>

<metricStrategies>
  <!-- Beschreibung der Metrikstrategien -->
</metricStrategies>
```

Jetzt dürfen beliebig viele Kriterien/Strategien an der entsprechenden Stelle angegeben werden. Ein Stopkriterium z.B. kann so aussehen:

```
<stopCriteria value="de.irf.it.pg527.runtime.IterationStopCriteria">
  <property name="limit" value="10" />
</stopCriteria>
```

Es wird ein neues Stopkriterium von der Klasse *IterationStopCriteria* hinzugefügt. Diese Klasse besitzt eine Variable *limit*, die über die Konfiguration gesetzt wird. Auf die gleiche Weise können auch mehrere Kriterien hinzugefügt werden.
Die Komponenten *Ausgabestrategie* und *Metrikstrategie* sind auf die gleiche Weise zu konfigurieren. (Beispiele dafür können den XML-Konfigurationen der Standardprobleme entnommen werden.) Sie können auf eine Komponente verzichten, indem Sie z.B. keine Ausgabestrategie unter *outputStrategies* angeben. Bedenken Sie aber, dass das Framework ohne ein Stopkriterium nicht terminiert.

Nun wird eine Konfiguration für ein Räuber-Beute-Modell beschrieben. In dieser Datei müssen Angaben zu folgenden Komponenten des Modells enthalten sein:

- Individuen (Räuber, Beute)
- Umgebungen (Umwelteinflüsse)
- Positionen
- Strukturerzeugung der Positionen

Der Grundaufbau einer XML-Datei für das Räuber-Beute-Modell sieht wie folgt aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<predatorPreyModel
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="de/irf/it/pg527/creation/ppo.xsd">

        <!-- Beschreibung der vier genannten Komponenten. -->

</predatorPreyModel>
```

Innerhalb dieses Rahmens sollte zunächst eine Welt definiert werden, indem Sie eine Menge von Positionen erzeugen und diese untereinander verknüpfen. Diese Verknüpfung wird im Framework durch eine Implementierung des Interfaces *structureMaker* erzeugt:

```
<positionGroup size="1600" class="de.irf.it.pg527.world.Position" />

<structureMaker class="de.irf.it.pg527.world.DefaultTorusmaker">
 <list name="sizes">
  <listProperty value="40" />
  <listProperty value="40" />
 </list>

</structureMaker>
```

Anschliessend muss diese Welt mit Individuen bevölkert werden. Im Multi-Sphere-Modell sind zwei Arten von Individuen vorgesehen: Einfache Beute- und Räuberindividuen. Die Beute benötigt eine Spezies (*PreySpecies*) und Chromosome vom Typ *Double*. Da die Beute in diesem Modell selbst nicht aktiv wird, muss ihr kein *Behaviour* zugewiesen werden.

```
<individualGroup size="1600" class="de.irf.it.pg527.individual.Individual">

 <property name="species" value="de.irf.it.pg527.individual.properties.species.
     PreySpecies" />

 <property name="chromosome" value="de.irf.it.pg527.individual.properties.chromosomes.
     DoubleArrayChromosome">
  <subproperty name="numberOfAlleles" value="2" />
  <subproperty name="upperbound" value="10" />
  <subproperty name="lowerbound" value="-10" />
 </property>

 <behaviour />

 <positioning class="de.irf.it.pg527.world.SimplePositioning" />

</individualGroup>
```

Die Beschreibung der Räuber hingegen ist wesentlich umfangreicher, da der Räuber neben seiner Spezies auch ein *Behaviour* und einen *Evaluator* braucht. Der Evaluator ist für die Auswertung der Beutefitness nötig. Durch das Behaviour wird beschrieben, welche Aktionen der Räuber ausführen kann.

```
<individualGroup size="1" class="de.irf.it.pg527.individual.Individual">

 <property name="species" value="de.irf.it.pg527.individual.properties.species.
     PredatorSpecies" />

 <property name="behaviour" value="de.irf.it.pg527.individual.behaviour.CommonBehaviour"
     >
  <subproperty name="movement" value="de.irf.it.pg527.individual.behaviour.action.
      movement.UniformMovement">
   <subproperty name="range" value="1" />
  </subproperty>

  <subproperty name="consumption"value="de.irf.it.pg527.individual.behaviour.action.
      consumption.ElitistMaximumConsumption" />

  <subproperty name="reproduction" value="de.irf.it.pg527.individual.behaviour.action.
      reproduction.ControlledReproduction">
   <list name="operators">
    <listProperty value="de.irf.it.pg527.individual.behaviour.action.reproduction.
        variation.CreatingMutationVO">
     <subproperty name="mutation" value="de.irf.it.pg527.individual.behaviour.action.
         reproduction.variation.TestDAChromosomeMutation">
```

```
      <subproperty name="distance" value="0.01" />
     </subproperty>
    </listProperty>
   </list>
  </subproperty>
 </property>

 <property name="evaluator" value="de.irf.it.pg527.runtime.evaluator.Evaluator">
  <subproperty name="testproblem" value="de.irf.it.pg527.runtime.evaluator.Multisphere.
      MultisphereTestProblem" />
 </property>

 <list name="objectives">
  <listProperty value="de.irf.it.pg527.individual.properties.objectives.Objective1" />
 </list>

 <positioning class="de.irf.it.pg527.world.RandomPositioning" />

</individualGroup>
```

Durch das Beispiel wird allerdings nur ein Räuber erzeugt. Für einen weiteren Räuber muss der gesamte Abschnitt ein zweites Mal in die Konfiguration eingetragen werden, wobei aber ein anderes Objective zugewiesen wird:

```
<list name="objectives">
 <listProperty value="de.irf.it.pg527.individual.properties.objectives.Objective2" />
</list>
```

Zum Schluss kann die Welt noch mit Umgebungseigenschaften versehen werden. Dies geschieht durch *Environments*, die Einflüsse auf die Welt und ihre Individuen ausüben können.

```
<environmentGroup class="de.irf.it.pg527.environment.ForestEnvironment">
 <list name="influences">
  <listProperty value="de.irf.it.pg527.environment.GrowingInfluence">
   <property name="noCoveredPositions" value="True" />
  </listProperty>
 </list>
 <placement class="de.irf.it.pg527.world.RandomEnvironmentPositioning">
  <property name="environmentSize" value="100" />
  <property name="noCoveredPositions" value="true" />
 </placement>
</environmentGroup>
```

C.0.4 Eigene Implementierungen ins Framework einbinden

Um eine eigene Klasse einzubinden muss zunächst festgestellt werden, um welche Schnittstelle es sich handelt. Fast alle Schnittstellen können über eine XML-Konfiguration eingebunden werden. Lediglich das Interface *PPOModel* kann nicht über XML erzeugt werden. Falls Sie eine eigene Implementierung für *PPOModell* nutzen wollen, müssen die VM-Argumente der Startkonfiguration angepasst werden.

Für das normale Hinzufügen einer neuen Implementierung muss die Klasse in der XML-Konfiguration angegeben werden. Vorraussetzung dafür ist, dass die neue Klasse den beschriebenen Konventionen C.0.2 entspricht. Wenn z.B. eine neue Klasse *myPackage.MyMovement* für die Bewegungsfunktion programmiert wurde, kann diese in ein bestehendes Modell eingebunden werden indem der bestehende Code für die Standardbewegung

```
<subproperty name="movement" value="de.irf.it.pg527.individual.behaviour.action.movement
    .UniformMovement">
```

```
 <subproperty name="range" value="1" />
</subproperty>
```

ersetzt wird durch die neue Klasse:

```
<subproperty name="movement" value="myPackage.MyMovement">
 <!-- Beschreibung der Eigenschaften von MyMovement, falls vorhanden. -->
</subproperty>
```

Auf diese Weise können sämtliche neuen Klassen in das Framework eingebunden werden. Dabei ist das *PPOModell* die einzige Ausnahme. Diese Klasse ist die zentrale Einheit für die Erzeugung des Räuber-Beute-Modells. Die bereitgestellte Implementierung dieser Klasse verwendet eine XML-Konfiguration, um dieses Modell zu erzeugen. Wenn diese Klasse ersetzt werden soll müssen sämtliche Routinen zur Erzeugung des Modells neu implementiert werden. Im Umkehrschluss ergibt sich daraus, dass eine XML-Konfiguration für das Modell nicht mehr benötigt wird.

Um z.B. die Klasse *myPackage.MyPPOModell* einzubinden, müssen die VM-Argumente angepasst werden. Die Angabe einer XML-Datei für das Räuber-Beute-Modell entfällt:

Windows:

```
-Dppo.xml.file="C:\...\config\multiSphere.xml"
```

Linux/Mac:

```
-Dppo.xml.file=/home/.../config/multiSphere.xml
```

Stattdessen muss die Klasse direkt angegeben werden:

Windows/Linux/Mac:

```
-Dppo.model= myPackage.MyPPOModel
```

Protokoll: Untersuchung von MultiSphere (Versuch 1)

Persönliche Daten:

Name: Tobias Pröger

Email: TobiasProeger@t-online.de

Problemstellung:

Einleitung: Wir untersuchen, wie sich die Variation verschiedener Parameter des Räuber-Beute-Modells auf die berechneten Ergebnisse auswirkt. Insbesondere werden die Auswirkungen der Variation von Torusgröße, Mutationsschrittweite, verschiedene Bewegungsarten sowie das Verhalten bei einer unterschiedlichen Anzahl von Räubern untersucht.

Fitnessfunktionen:

$$f_1 : x \mapsto x^2 + y^2$$
$$f_2 : x \mapsto (x-2)^2 + y^2$$

Hypothese: Bei einer zu kleinen Mutationsschrittweite stellt sich ein unbefriedigendes Langzeitverhalten ein, da nur extremale Lösungen gefunden werden. Im Folgenden untersuchen wir die Auswirkungen für $\sigma = 0,02$.

Testproblem: de.irf.it.pg527.runtime.evaluator.Multisphere.MultisphereTestProblem

Objectives:

- de.irf.it.pg527.individual.properties.objectives.Objective1
- de.irf.it.pg527.individual.properties.objectives.Objective2

Chromosomen: de.irf.it.pg527.individual.properties.chromosomes.DoubleArrayChromosome mit den Einstellungen

- numberOfAlleles = 2
- upperbound = 10
- lowerbound = -10

Systemeinstellungen:

Strategie der Engine: de.irf.it.pg527.runtime.SimpleSerialStrategy

Iterationen: 10 unabhängige Durchläufe mit je 250000 Iterationen je Durchlauf

Modelleinstellungen:

Torus:

Anzahl der Positionen: $40 \times 40 = 1600$

Aufbau: Doppelt verketteter Torus

Positioning: de.irf.it.pg527.world.SimplePositioning

Räuber:

1. Gruppe: Predator ID1

Species: de.irf.it.pg527.individual.properties.species.PredatorSpecies

Anzahl: 1

Verhalten: Als Behaviour wird die Klasse de.irf.it.pg527.individual.behaviour.CommonBehaviour verwendet. Die Bewegung findet zufällig gleichverteilt mit Distanz 1 statt. Weiterhin wird die Klasse de.irf.it.pg527.individual.behaviour.action.consumption.ElitistMaximumConsumption als Consumption verwendet. Für die Reproduktion wird de.irf.it.pg527.individual.behaviour.action.reproduction.ControlledReproduction benutzt. Inbesondere wird zur Variation der Operator de.irf.it.pg527.individual.behaviour.action.reproduction.variation.TestDAChromosomeMutation mit Parameter distance = 0.02 benutzt. Dies addiert einen zufälligen Vektor aus $\mathbb{R}^2$, dessen Komponenten gemäß der Gauß-Verteilung mit $\mu = 0$ und $\sigma = 0,02$ gewählt werden. Initial positioniert wird der Räuber zufällig.

Fitnessfunktion: f_1

2. Gruppe: Predator ID2

Species: de.irf.it.pg527.individual.properties.species.PredatorSpecies

Anzahl: 1

Verhalten: Analog zu Räuber 1.

Fitnessfunktion: f_2

Beute:

1. Gruppe: Prey ID1

Species: de.irf.it.pg527.individual.properties.species.PreySpecies

Anzahl: 1600

Initiale Chromosomen: Zufällig gewählt

Evaluation:

Die Evaluation erfolgt nach Systemstart und danach alle 2500 Iterationen bis das Abbruchkriterium erfüllt ist.

Interpretation und Diskussion:

Die durchgeführten Testläufe bestätigten weitestgehend die aufgestellte Hypothese. Die Paretofront wird häufig (aber nicht immer) in den ersten Iterationen nahezu vollständig aufgebaut, danach bilden sich erste „Löcher“ und führen schließlich dazu, dass fast nur noch extremale (Bilder von) Lösungen dargestellt werden. Diese finden sich an den Enden der Paretofront in den Punkten $(0, 4)$ bzw. $(4, 0)$. Die Paretomenge bestätigt ebenfalls die Vermutung. Es bilden sich Punktemengen von Lösungen um die Punkte $(0, 0)$ bzw. $(2, 0)$ herum. Insbesondere fällt auf, dass die Ausrichtung der Lösungen im Lösungsraum trichterförmig ist. Auch in den Fällen, in denen die Paretofront vollständig aufgebaut wird und vermeintlich relativ gleichmäßig approximiert erscheint, zeigt die Paretofront, dass es mehr Extremallösungen als Lösungen, die beide Kriterien gleichmäßig erfüllen, gibt.

Das beschriebene Langzeitverhalten ist auch intuitiv einleuchtend: Sobald sich genügend Lösungspunkte in der Nähe der Extremallösungen gesammelt haben, werden bei der Rekombination wieder Lösungen entstehen, die in der Nähe der Extremallösungen liegen. Wird die Mutationsschrittweite relativ klein (hier: $\sigma = 0,02$) gewählt, ist die Wahrscheinlichkeit, dass die neu erzeugte Lösung nicht in der Nähe der Extremallösungen liegt, ebenfalls gering. Außerdem existieren dann auch in diesem Fall weiterhin viele Punkte in der Nähe der Extremallösungen. Wird die erzeugte Lösung dann irgendwann wieder entfernt (weil ein Räuber das betreffende Individuum gefressen hat), dann wird dieses mit einer hohen Wahrscheinlichkeit durch ein Individuum ersetzt, welches nah am Rand liegt. Dieses Phänomen wird daher immer stärker auftreten.

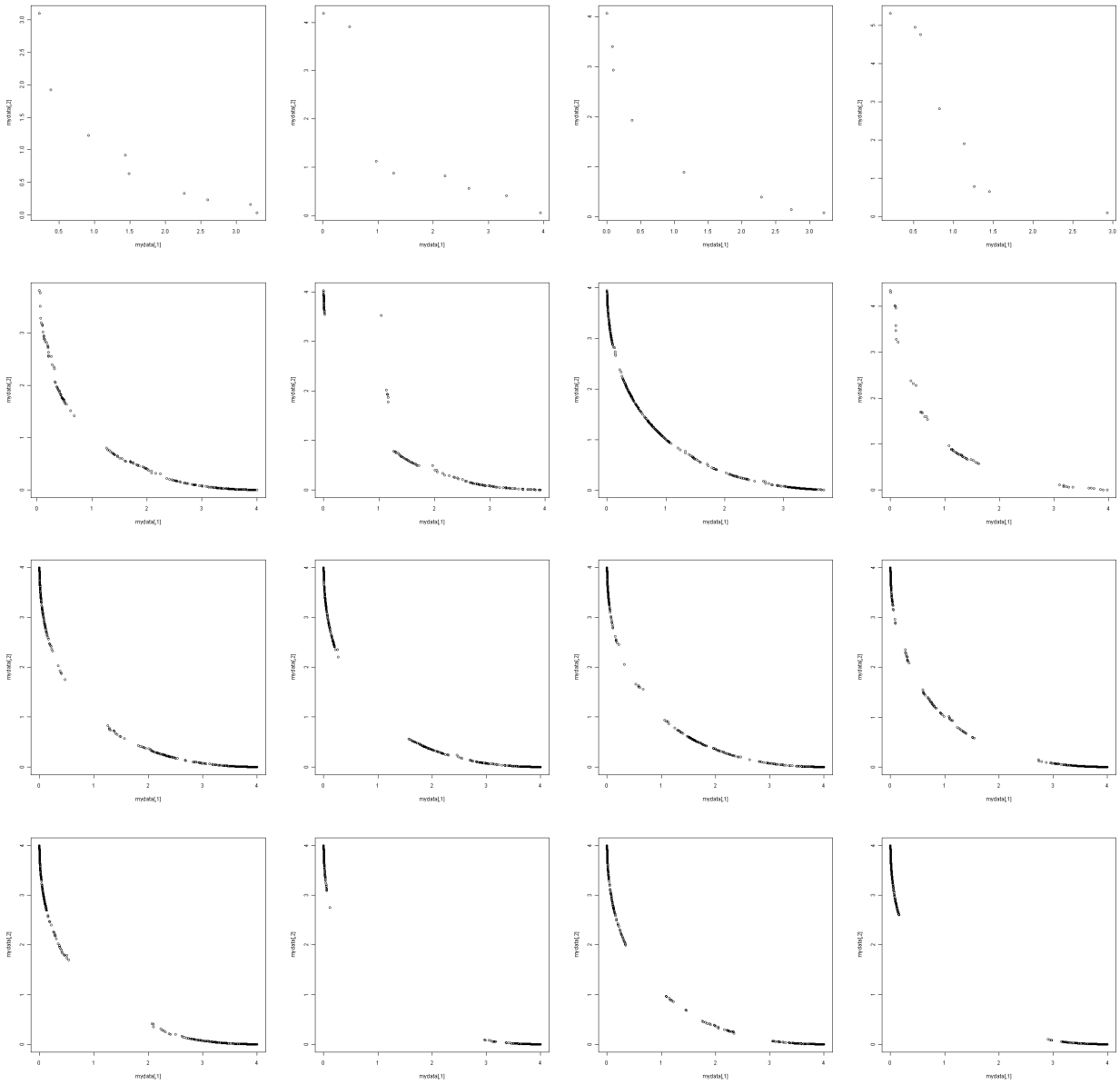

Abbildung 34: Die Paretofronten (von oben nach unten) initial, nach 50000, nach 150000, nach 250000 Iterationen.

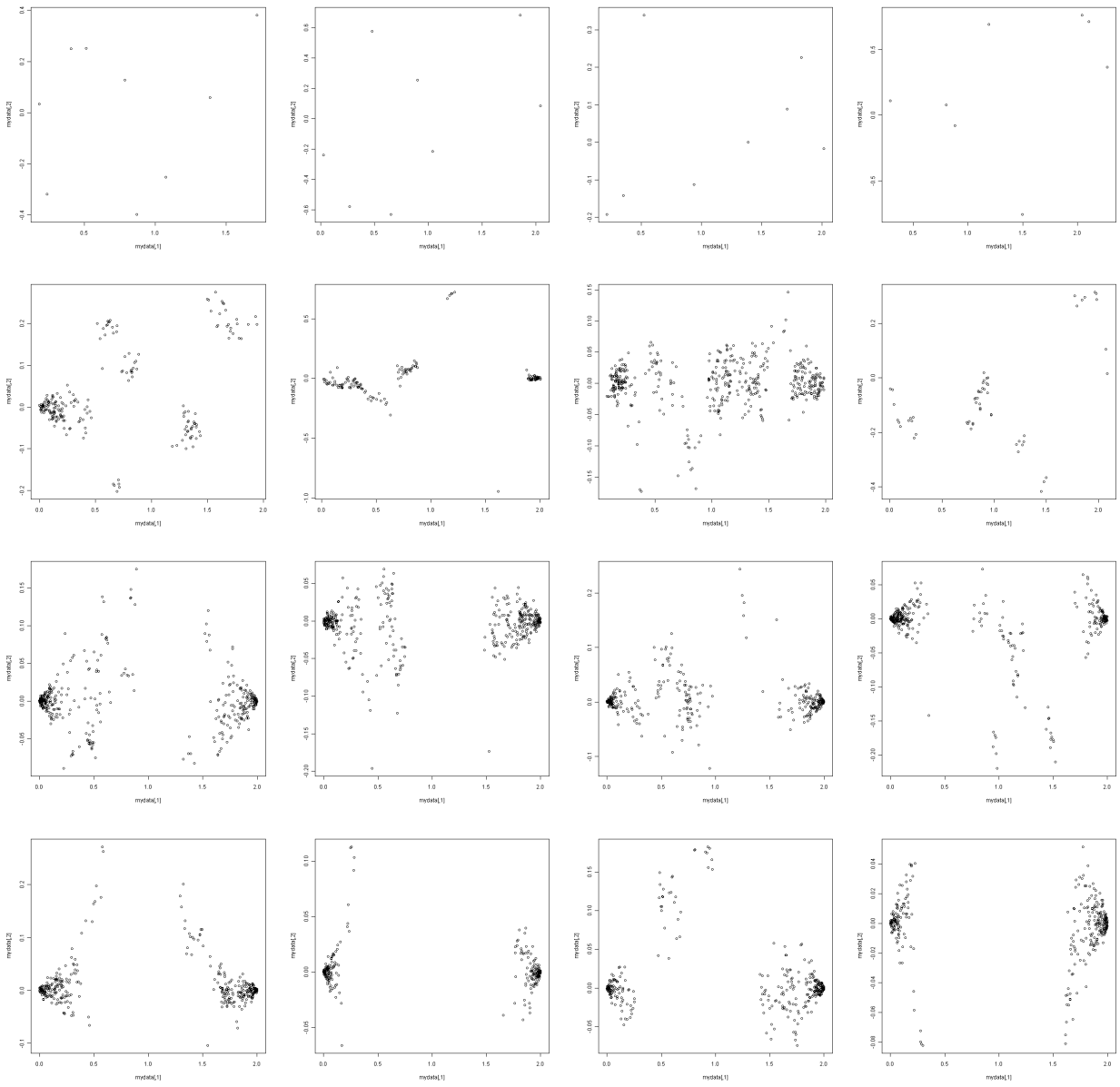

Abbildung 35: Die Paretomengen (von oben nach unten) initial, nach 50000, nach 150000, nach 250000 Iterationen.

Protokoll: Untersuchung von MultiSphere (Versuch 2)

Persönliche Daten:

Name: Tobias Pröger

Email: TobiasProeger@t-online.de

Problemstellung:

Einleitung: Wir untersuchen, wie sich die Variation verschiedener Parameter des Räuber-Beute-Modells auf die berechneten Ergebnisse auswirkt. Insbesondere werden die Auswirkungen der Variation von Torusgröße, Mutationsschrittweite, verschiedene Bewegungsarten sowie das Verhalten bei einer unterschiedlichen Anzahl von Räubern untersucht.

Fitnessfunktionen:

$$f_1 : x \mapsto x^2 + y^2$$
$$f_2 : x \mapsto (x-2)^2 + y^2$$

Hypothese: Wird der Torus viel kleiner als die (Standard-)Größe 40×40 gewählt, dann erkennt man die Eigenschaft, dass im Langzeitverhalten hauptsächlich Lösungen nahe der Extremallösungen auftreten, noch früher.

Testproblem: de.irf.it.pg527.runtime.evaluator.Multisphere.MultisphereTestProblem

Objectives:

- de.irf.it.pg527.individual.properties.objectives.Objective1
- de.irf.it.pg527.individual.properties.objectives.Objective2

Chromosomen: de.irf.it.pg527.individual.properties.chromosomes.DoubleArrayChromosome mit den Einstellungen

- numberOfAlleles = 2
- upperbound = 10
- lowerbound = -10

Systemeinstellungen:

Strategie der Engine: de.irf.it.pg527.runtime.SimpleSerialStrategy

Iterationen: 10 unabhängige Durchläufe mit je 250000 Iterationen je Durchlauf

Modelleinstellungen:

Torus:

Anzahl der Positionen: $20 \times 20 = 400$

Aufbau: Doppelt verketteter Torus

Positioning: de.irf.it.pg527.world.SimplePositioning

Räuber:

1. Gruppe: Predator ID1

Species: de.irf.it.pg527.individual.properties.species.PredatorSpecies

Anzahl: 1

Verhalten: Als Behaviour wird die Klasse de.irf.it.pg527.individual.behaviour.CommonBehaviour verwendet. Die Bewegung findet zufällig gleichverteilt mit Distanz 1 statt. Weiterhin wird die Klasse de.irf.it.pg527.individual.behaviour.action.consumption.ElitistMaximumConsumption als Consumption verwendet. Für die Reproduktion wird de.irf.it.pg527.individual.behaviour.action.reproduction.ControlledReproduction benutzt. Inbesondere wird zur Variation der Operator de.irf.it.pg527.individual.behaviour.action.reproduction.variation.TestDAChromosomeMutation mit Parameter distance = 0.02 benutzt. Dies addiert einen zufälligen Vektor aus $\mathbb{R}^2$, dessen Komponenten gemäß der Gauß-Verteilung mit $\mu = 0$ und $\sigma = 0,02$ gewählt werden. Initial positioniert wird der Räuber zufällig.

Fitnessfunktion: f_1

2. Gruppe: Predator ID2

Species: de.irf.it.pg527.individual.properties.species.PredatorSpecies

Anzahl: 1

Verhalten: Analog zu Räuber 1.

Fitnessfunktion: f_2

Beute:

1. Gruppe: Prey ID1

Species: de.irf.it.pg527.individual.properties.species.PreySpecies

Anzahl: 400

Initiale Chromosomen: Zufällig gewählt

Evaluation:

Die Evaluation erfolgt nach Systemstart und danach alle 2500 Iterationen bis das Abbruchkriterium erfüllt ist.

Interpretation und Diskussion:

Tatsächlich bestätigt sich die Vermutung. Die Paretofront wird zunächst aufgebaut, teilweise erkennt man aber schon sehr schnell, dass sich hauptsächlich Enden an den Bildern der Extremallösungen bilden. Im Prinzip ist dieser Versuch analog zum ersten Versuch zu sehen. Da allerdings der Torus früher 1600 Positionen aufwies und heute nur noch ein Viertel, tritt das Langzeitverhalten dann natürlich viel früher ein. Häufig war der auch am nach 250000 Iteration zu erkennende Zustand schon nach etwa 75000 Iterationen vorhanden. Von diesem Zeitpunkt an wuchsen die Enden der Paretofront nur noch kurz periodisch an und nahmen wieder ab.

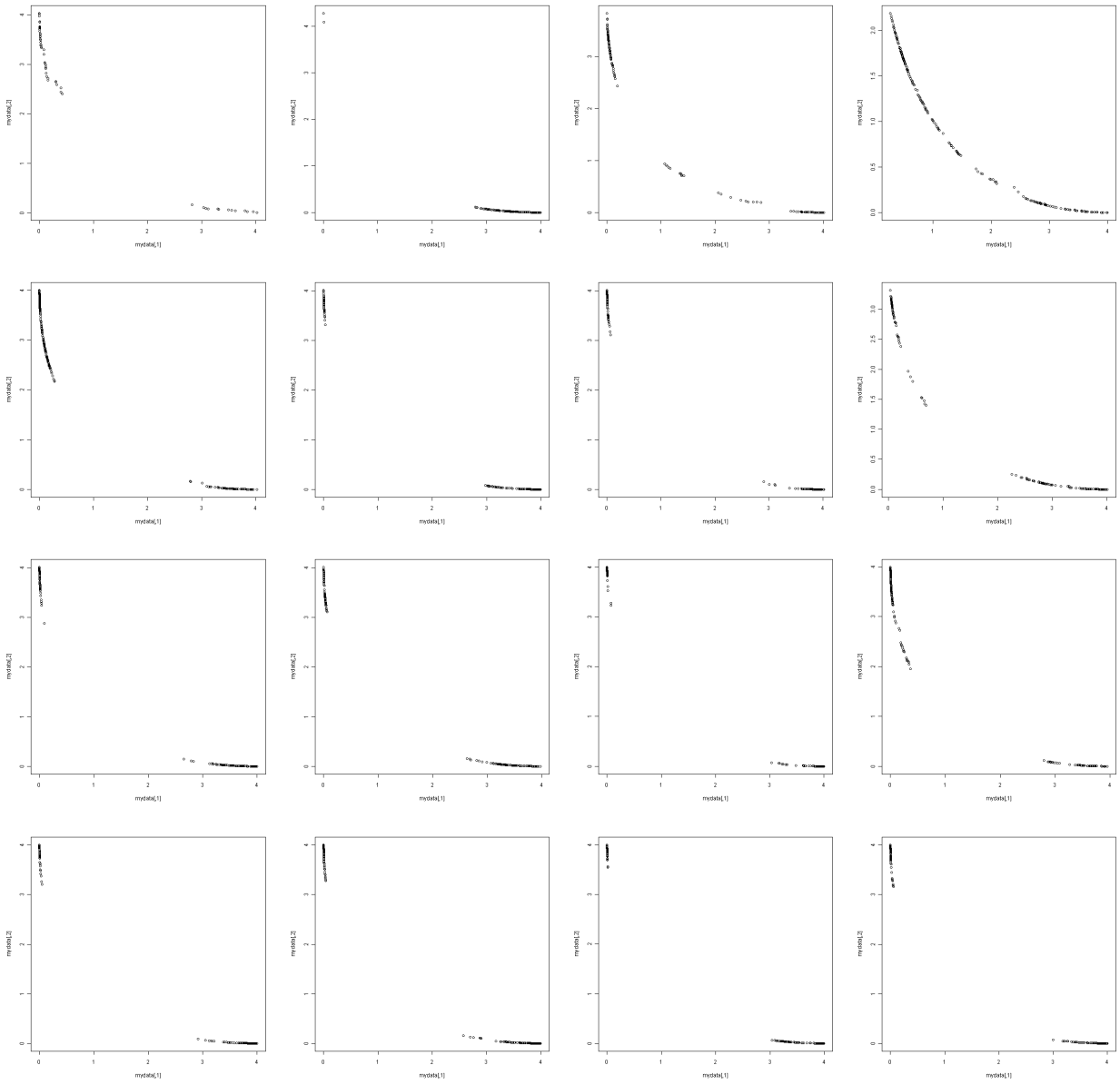

Abbildung 36: Die Paretofronten (von oben nach unten) nach 25000, nach 50000, nach 100000, nach 250000 Iterationen.

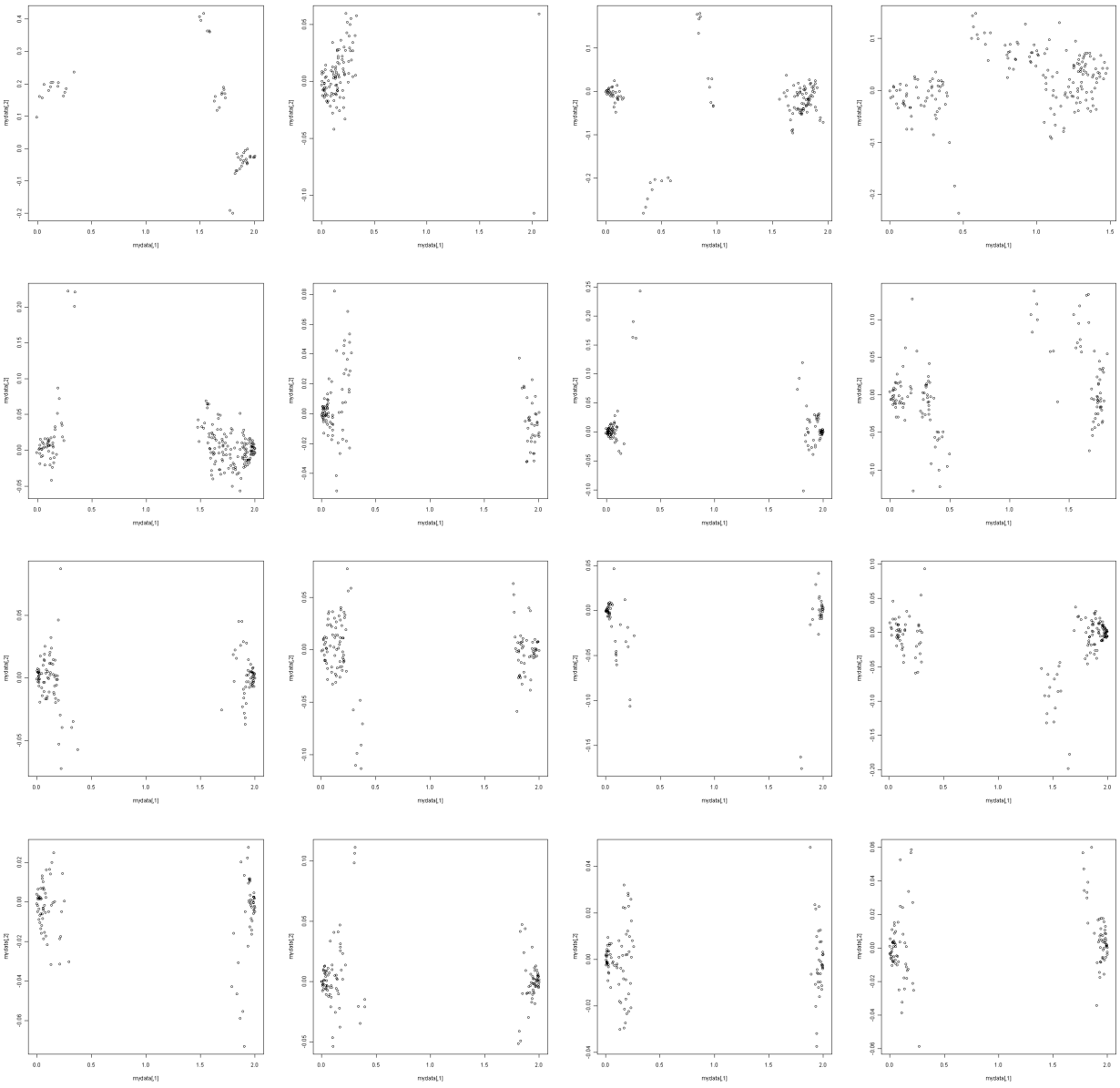

Abbildung 37: Die Paretomengen (von oben nach unten) nach 25000, nach 50000, nach 100000, nach 250000 Iterationen.

Protokoll: Untersuchung von MultiSphere (Versuch 3)

Persönliche Daten:

Name: Tobias Pröger

Email: TobiasProeger@t-online.de

Problemstellung:

Einleitung: Wir untersuchen, wie sich die Variation verschiedener Parameter des Räuber-Beute-Modells auf die berechneten Ergebnisse auswirkt. Insbesondere werden die Auswirkungen der Variation von Torusgröße, Mutationsschrittweite, verschiedene Bewegungsarten sowie das Verhalten bei einer unterschiedlichen Anzahl von Räubern untersucht.

Fitnessfunktionen:

$$f_1 : x \mapsto x^2 + y^2$$
$$f_2 : x \mapsto (x-2)^2 + y^2$$

Hypothese: Wird die Mutationsschrittweite sehr klein gewählt, dann zeigt sich, dass Lösungen verklumpen (was nicht notwendigerweise ausschließlich bei den Extremallösungen der Fall sein muss). Dennoch zeigt auch hier das im ersten Versuch beschriebene Langzeitverhalten. Im Folgenden untersuchen wir $\sigma = 0,01$.

Testproblem: de.irf.it.pg527.runtime.evaluator.Multisphere.MultisphereTestProblem

Objectives:

- de.irf.it.pg527.individual.properties.objectives.Objective1
- de.irf.it.pg527.individual.properties.objectives.Objective2

Chromosomen: de.irf.it.pg527.individual.properties.chromosomes.DoubleArrayChromosome mit den Einstellungen

- numberOfAlleles = 2
- upperbound = 10
- lowerbound = −10

Systemeinstellungen:

Strategie der Engine: de.irf.it.pg527.runtime.SimpleSerialStrategy

Iterationen: 10 unabhängige Durchläufe mit je 250000 Iterationen je Durchlauf

Modelleinstellungen:

Torus:

Anzahl der Positionen: $40 \times 40 = 1600$

Aufbau: Doppelt verketteter Torus

Positioning: de.irf.it.pg527.world.SimplePositioning

Räuber:

1. Gruppe: Predator ID1

Species: de.irf.it.pg527.individual.properties.species.PredatorSpecies

Anzahl: 1

Verhalten: Als Behaviour wird die Klasse de.irf.it.pg527.individual.behaviour.CommonBehaviour verwendet. Die Bewegung findet zufällig gleichverteilt mit Distanz 1 statt. Weiterhin wird die Klasse de.irf.it.pg527.individual.behaviour.action.consumption.ElitistMaximumConsumption als Consumption verwendet. Für die Reproduktion wird de.irf.it.pg527.individual.behaviour.action.reproduction.ControlledReproduction benutzt. Inbesondere wird zur Variation der Operator de.irf.it.pg527.individual.behaviour.action.reproduction.variation.TestDAChromosomeMutation mit Parameter distance = 0.01 benutzt. Dies addiert einen zufälligen Vektor aus $\mathbb{R}^2$, dessen Komponenten gemäß der Gauß-Verteilung mit $\mu = 0$ und $\sigma = 0,01$ gewählt werden. Initial positioniert wird der Räuber zufällig.

Fitnessfunktion: f_1

2. Gruppe: Predator ID2

Species: de.irf.it.pg527.individual.properties.species.PredatorSpecies

Anzahl: 1

Verhalten: Analog zu Räuber 1.

Fitnessfunktion: f_2

Beute:

1. Gruppe: Prey ID1

Species: de.irf.it.pg527.individual.properties.species.PreySpecies

Anzahl: 1600

Initiale Chromosomen: Zufällig gewählt

Evaluation:

Die Evaluation erfolgt nach Systemstart und danach alle 2500 Iterationen bis das Abbruchkriterium erfüllt ist.

Interpretation und Diskussion:

Bei einer sehr kleinen Mutationsschrittweise werden die neu erzeugten Individuen kaum verändert, und es vergeht viel Zeit, bis größere Änderungen sichtbar werden. Die Betrachtung der Paretomengen zeigt, dass es im Lösungsbereich temporär zu Verklumpungen kommt. Dies ist relativ klar, da die Individuen initial zufällig gewählt werden und die Mutationsschrittweite klein ist. Sobald sich eine Verklumpung im Lösungsraum herausgebildet hat, dauert es durch die sehr schwache Mutation sehr lange, bis sich diese Verklumpung auflöst. Diese Verklumpungen zeigen sich auch im Bildbereich, wie man anhand der Paretomenge feststellen kann. Sie ist zwischenzeitlich nur stückweise verhanden. Nach 250000 Iterationen zeigt sich dann aber wieder das aus Versuch 1 bekannte Langzeitverhalten. Lösungen bilden sich hauptsächlich in der Nähe der Extremallösungen. Der Grund ist der gleiche wie in Versuch 1 beschrieben. Da die Anzahl der Lösungen in den Verklumpungen im Vergleich zu der Anzahl der Lösungen, die nahe der Extremallösungen liegen, meist gering ist, lösen sich erstere dann auch auf.

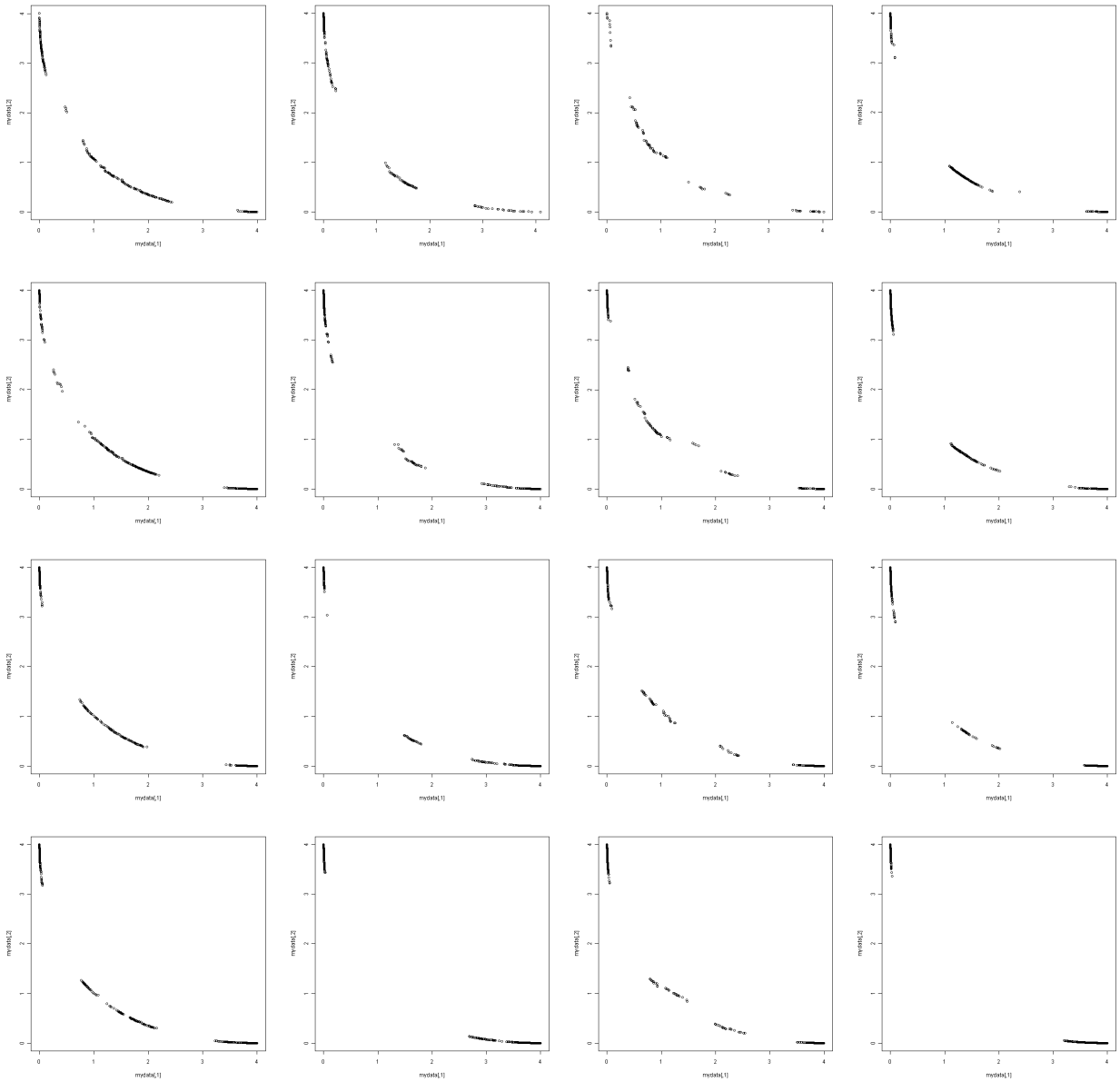

Abbildung 38: Die Paretofronten (von oben nach unten) nach 100000, nach 150000, nach 200000, nach 250000 Iterationen.

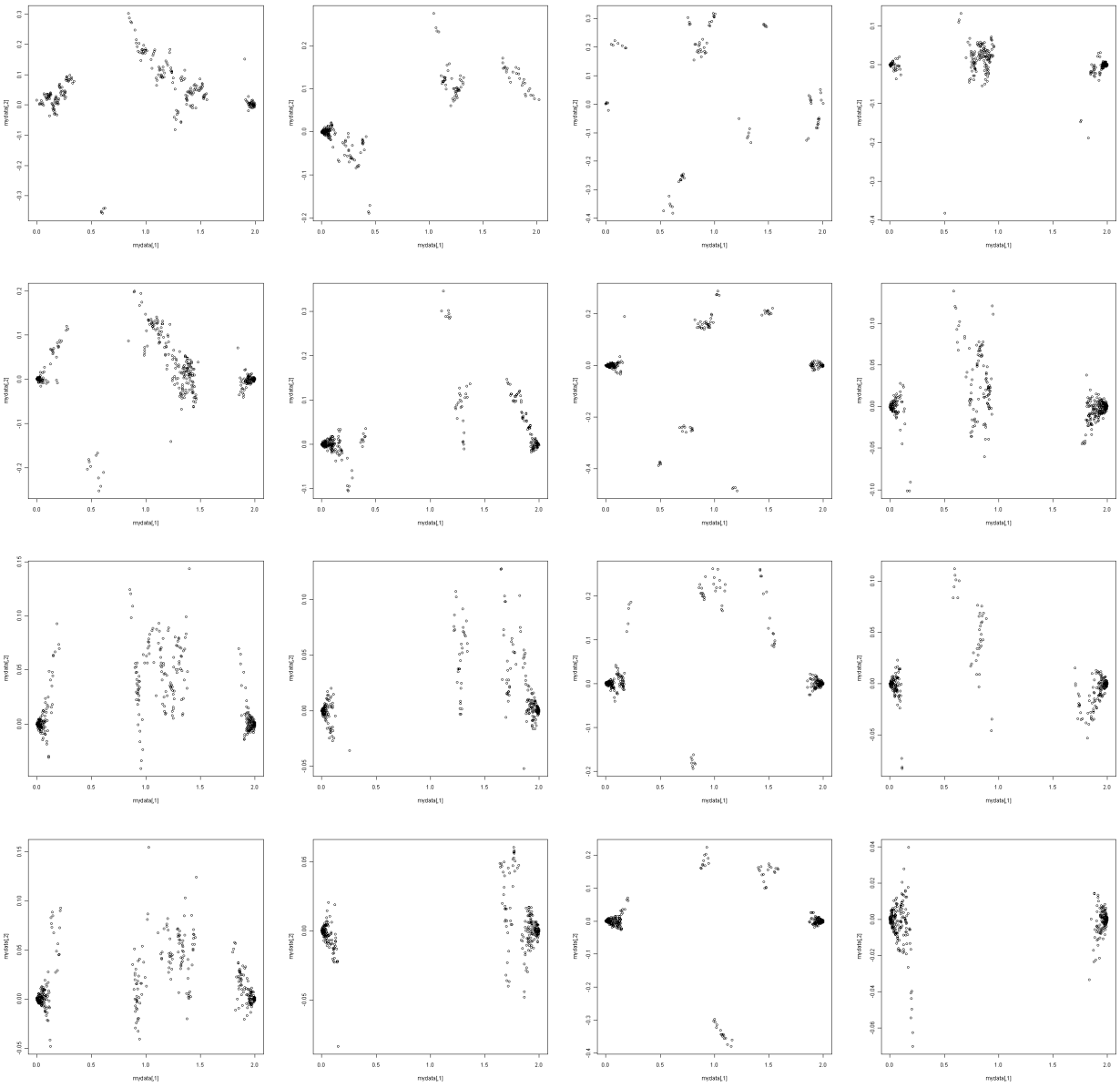

Abbildung 39: Die Paretomengen (von oben nach unten) nach 100000, nach 150000, nach 200000, nach 250000 Iterationen.

Protokoll: Untersuchung von MultiSphere (Versuch 4)

Persönliche Daten:

Name: Tobias Pröger

Email: TobiasProeger@t-online.de

Problemstellung:

Einleitung: Wir untersuchen, wie sich die Variation verschiedener Parameter des Räuber-Beute-Modells auf die berechneten Ergebnisse auswirkt. Insbesondere werden die Auswirkungen der Variation von Torusgröße, Mutationsschrittweite, verschiedene Bewegungsarten sowie das Verhalten bei einer unterschiedlichen Anzahl von Räubern untersucht.

Fitnessfunktionen:

$$f_1 : x \mapsto x^2 + y^2$$
$$f_2 : x \mapsto (x-2)^2 + y^2$$

Hypothese: Selbst bei größerer Wahl der Mutationsschrittweite als bisher getestet (bisher: $\sigma = 0,01$, $\sigma = 0,02$) wird die Paretofront nicht gleichmäßig approximiert. Zwar gilt, dass die Anzahl der guten Kompromisslösungen (zumindest bis zu einem gewissen Zeitpunkt) wachsend in σ ist, dennoch ist ihr Anteil an der Gesamtanzahl der nicht-dominierter Lösungen gering. Auf lange Zeit gesehen bilden sich wieder nur Lösungen nahe der Extremallösungen. Getestet werden $\sigma = 0,03$ und $\sigma = 0,04$.

Testproblem: de.irf.it.pg527.runtime.evaluator.Multisphere.MultisphereTestProblem

Objectives:

- de.irf.it.pg527.individual.properties.objectives.Objective1
- de.irf.it.pg527.individual.properties.objectives.Objective2

Chromosomen: de.irf.it.pg527.individual.properties.chromosomes.DoubleArrayChromosome mit den Einstellungen

- numberOfAlleles = 2
- upperbound = 10
- lowerbound = -10

Systemeinstellungen:

Strategie der Engine: de.irf.it.pg527.runtime.SimpleSerialStrategy

Iterationen: Je 10 unabhängige Durchläufe mit je 350000 Iterationen je Durchlauf

Modelleinstellungen:

Torus:

Anzahl der Positionen: $40 \times 40 = 1600$

Aufbau: Doppelt verketteter Torus

Positioning: de.irf.it.pg527.world.SimplePositioning

Räuber:

1. Gruppe: Predator ID1

Species: de.irf.it.pg527.individual.properties.species.PredatorSpecies

Anzahl: 1

Verhalten: Als Behaviour wird die Klasse de.irf.it.pg527.individual.behaviour.CommonBehaviour verwendet. Die Bewegung findet zufällig gleichverteilt mit Distanz 1 statt. Weiterhin wird die Klasse de.irf.it.pg527.individual.behaviour.action.consumption.ElitistMaximumConsumption als Consumption verwendet. Für die Reproduktion wird de.irf.it.pg527.individual.behaviour.action.reproduction.ControlledReproduction benutzt. Inbesondere wird zur Variation der Operator de.irf.it.pg527.individual.behaviour.action.reproduction.variation.TestDAChromosomeMutation mit den Parametern distance = 0.03 bzw. distance = 0.04 benutzt. Dies addiert einen zufälligen Vektor aus $\mathbb{R}^2$, dessen Komponenten gemäß der Gauß-Verteilung mit $\mu = 0$ und $\sigma =$ distance gewählt werden. Initial positioniert wird der Räuber zufällig.

Fitnessfunktion: f_1

2. Gruppe: Predator ID2

Species: de.irf.it.pg527.individual.properties.species.PredatorSpecies

Anzahl: 1

Verhalten: Analog zu Räuber 1.

Fitnessfunktion: f_2

Beute:

1. Gruppe: Prey ID1

Species: de.irf.it.pg527.individual.properties.species.PreySpecies

Anzahl: 1600

Initiale Chromosomen: Zufällig gewählt

Evaluation:

Die Evaluation erfolgt nach Systemstart und danach alle 2500 Iterationen bis das Abbruchkriterium erfüllt ist.

Interpretation und Diskussion:

Betrachtet man die auf den folgenden Seiten dargestellten Paretofronten und -mengen, dann erkennt man, dass sich die Hypothese weitestgehend bestätigt. Tatsächlich bilden sich wieder hauptsächlich Lösungen in der Nähe der Extremallösungen. Je größer natürlich die Mutationsschrittweite gewählt wird, desto stärker wird die Mutation und desto größer streuen dann die neu erzeugten Individuen. Auf den ersten Blick auf die Paretofront kann man vermuten, dass sie gleichmäßig approximiert wird. Die Paretomengen zeigen dann, dass dies leider nicht der Fall ist. Allein durch Variation der Mutationsschrittweite kann man das Problem des schlechten Langzeitverhaltens also nicht lösen, sondern das Verhalten nur dahingehend verbessern, dass es etwas mehr Lösungen gibt, die beiden Kriterien in etwa gleich gut entsprechen. Mit wachsender Torusgröße nimmt ihr Anteil dann aber wieder ab.

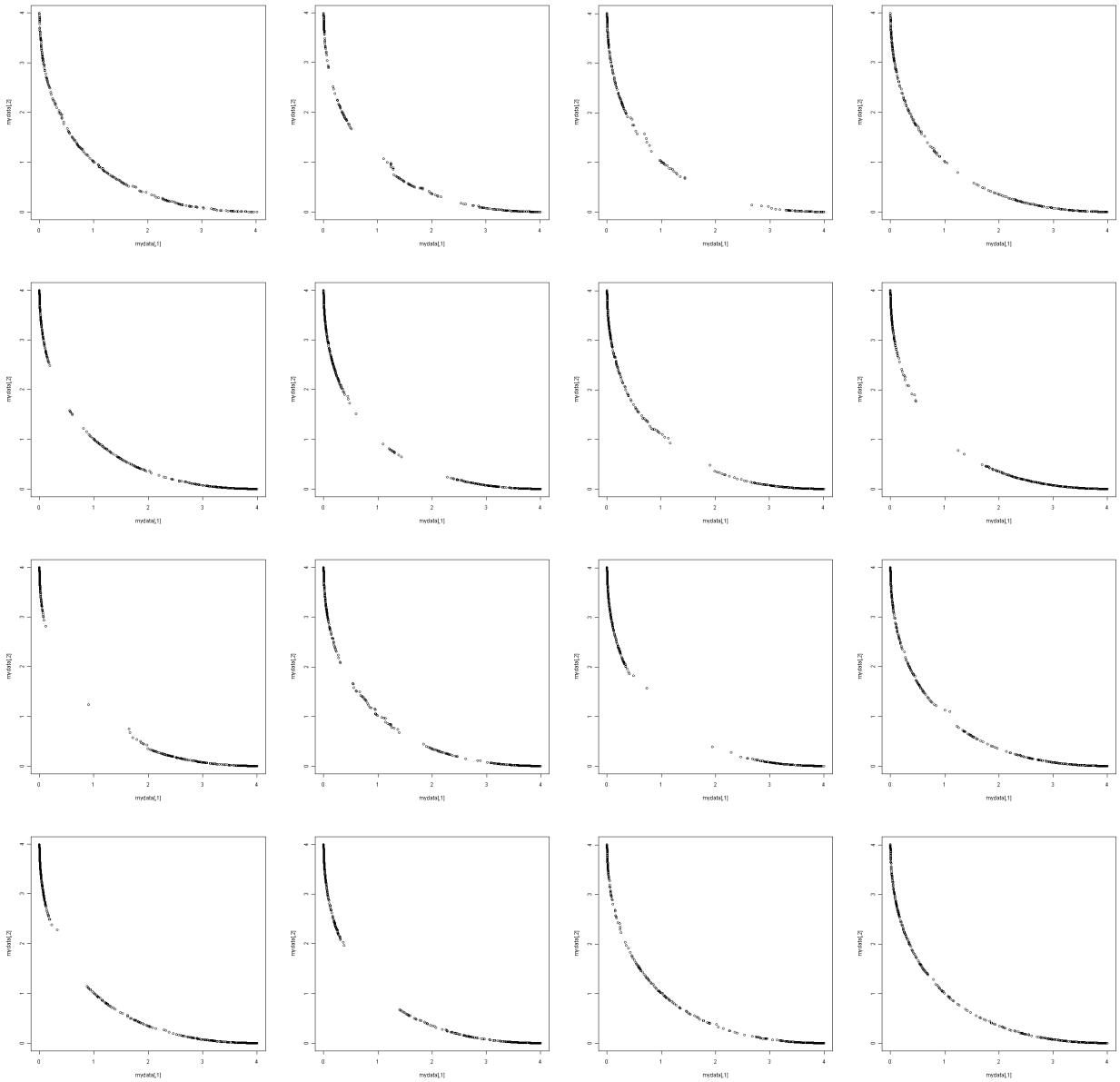

Abbildung 40: Die Paretofronten (von oben nach unten) nach 50000, nach 150000, nach 250000, nach 350000 Iterationen. Als Einstellungen wurden $\sigma = 0,03$ (links und mitte links) bzw. $\sigma = 0,04$ (mitte rechts und rechts) verwendet.

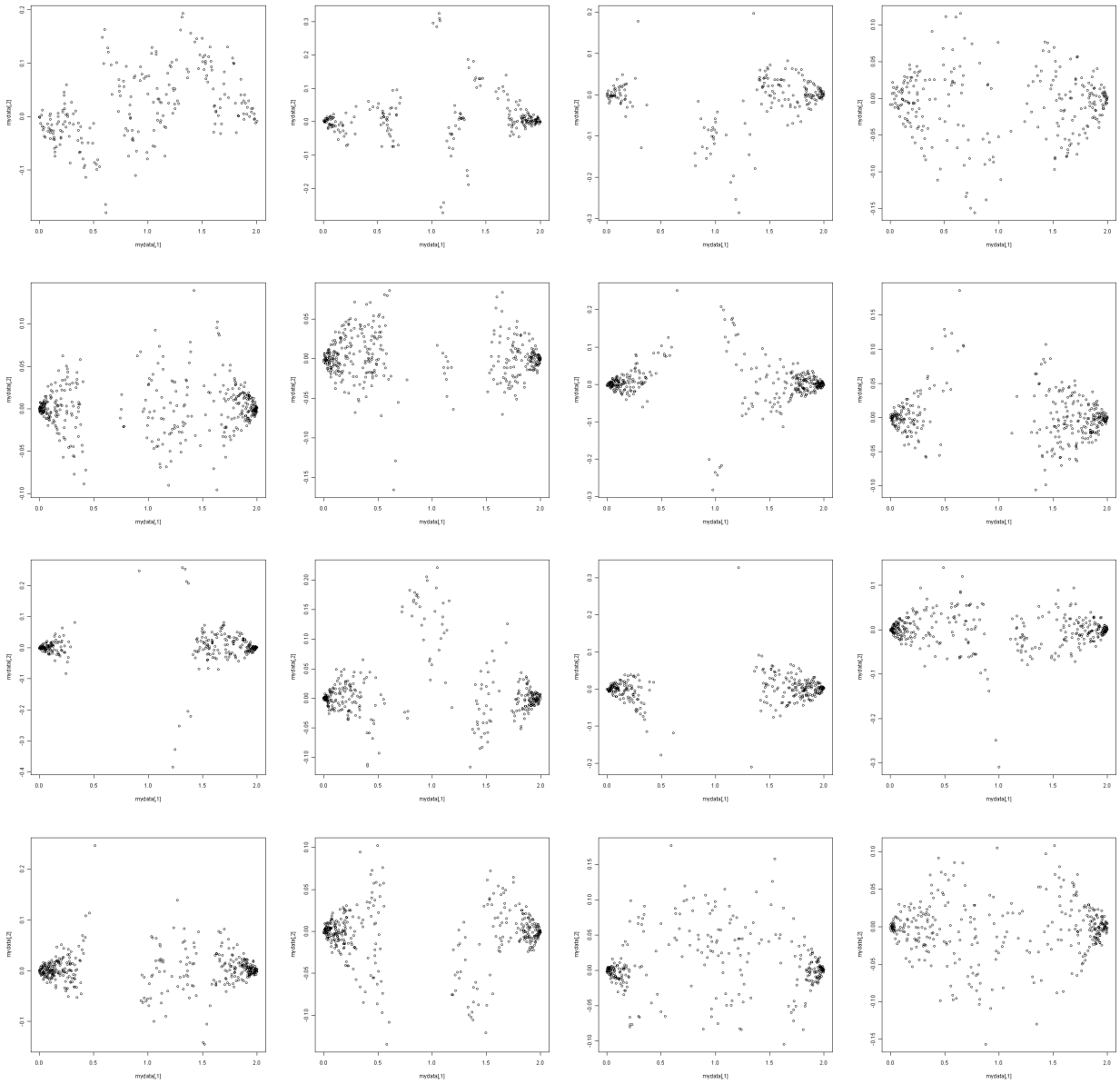

Abbildung 41: Die Paretomengen (von oben nach unten) nach 50000, nach 150000, nach 250000, nach 350000 Iterationen. Als Einstellungen wurden $\sigma = 0,03$ (links und mitte links) bzw. $\sigma = 0,04$ (mitte rechts und rechts) verwendet.

Protokoll: Untersuchung von MultiSphere (Versuch 5)

Persönliche Daten:

Name: Tobias Pröger

Email: TobiasProeger@t-online.de

Problemstellung:

Einleitung: Wir untersuchen, wie sich die Variation verschiedener Parameter des Räuber-Beute-Modells auf die berechneten Ergebnisse auswirkt. Insbesondere werden die Auswirkungen der Variation von Torusgröße, Mutationsschrittweite, verschiedene Bewegungsarten sowie das Verhalten bei einer unterschiedlichen Anzahl von Räubern untersucht.

Fitnessfunktionen:

$$f_1 : x \mapsto x^2 + y^2$$
$$f_2 : x \mapsto (x-2)^2 + y^2$$

Hypothese: Wird für das Movement der Parameter „range“ (der Bewegungsradius) größer als 1 angesetzt, gilt das früher beschriebene Verhalten nicht mehr. Es werden mehr gute Kompromisslösungen als Extremallösungen gefunden.

Testproblem: de.irf.it.pg527.runtime.evaluator.Multisphere.MultisphereTestProblem

Objectives:

- de.irf.it.pg527.individual.properties.objectives.Objective1
- de.irf.it.pg527.individual.properties.objectives.Objective2

Chromosomen: de.irf.it.pg527.individual.properties.chromosomes.DoubleArrayChromosome mit den Einstellungen

- numberOfAlleles = 2
- upperbound = 10
- lowerbound = -10

Systemeinstellungen:

Strategie der Engine: de.irf.it.pg527.runtime.SimpleSerialStrategy

Iterationen: Je 10 unabhängige Durchläufe mit je 350000 Iterationen je Durchlauf

Modelleinstellungen:

Torus:

Anzahl der Positionen: $40 \times 40 = 1600$

Aufbau: Doppelt verketteter Torus

Positioning: de.irf.it.pg527.world.SimplePositioning

Räuber:

1. Gruppe: Predator ID1

Species: de.irf.it.pg527.individual.properties.species.PredatorSpecies

Anzahl: 1

Verhalten: Als Behaviour wird die Klasse de.irf.it.pg527.individual.behaviour.CommonBehaviour verwendet. Die Bewegung findet zufällig gleichverteilt mit den Distanzen 2, 3 und 4 statt. Weiterhin wird die Klasse de.irf.it.pg527.individual.behaviour.action.consumption.ElitistMaximumConsumption als Consumption verwendet. Für die Reproduktion wird de.irf.it.pg527.individual.behaviour.action.reproduction.ControlledReproduction benutzt. Inbesondere wird zur Variation der Operator de.irf.it.pg527.individual.behaviour.action.reproduction.variation.TestDAChromosomeMutation mit Parameter distance = 0.02 benutzt. Dies addiert einen zufälligen Vektor aus $\mathbb{R}^2$, dessen Komponenten gemäß der Gauß-Verteilung mit $\mu = 0$ und $\sigma = 0,02$ gewählt werden. Initial positioniert wird der Räuber zufällig.

Fitnessfunktion: f_1

2. Gruppe: Predator ID2

Species: de.irf.it.pg527.individual.properties.species.PredatorSpecies

Anzahl: 1

Verhalten: Analog zu Räuber 1.

Fitnessfunktion: f_2

Beute:

1. Gruppe: Prey ID1

Species: de.irf.it.pg527.individual.properties.species.PreySpecies

Anzahl: 1600

Initiale Chromosomen: Zufällig gewählt

Evaluation:

Die Evaluation erfolgt nach Systemstart und danach alle 2500 Iterationen bis das Abbruchkriterium erfüllt ist.

Interpretation und Diskussion:

Wenn sich der Räuber zufällig gleichverteilt im Radius $r = 2$ bewegt, fallen die Ergebnisse unterschiedlich aus. Es existieren Läufe, bei denen viele Lösungen gefunden werden, bei denen beide Kriterien zum Tragen kommen. Insbesondere werden kaum Lösungen in der Nähe der Extramllösungen gefunden. Auf der anderen Seite existieren aber auch Läufe, bei denen es wieder zu starken Clusterbildungen in der Nähe der Extremallösungen kommen (vgl. Abbildungen). Für $r = 3$ und $r = 4$ ist die Situation besser: Es werden keine Extremallösungen gefunden, sondern die Lösungen sind (relativ) gut im Lösungsraum verteilt. Dieses Verhalten zeigte sich auch nicht nur in einigen wenigen, sondern in allen zehn durchgeführten Testläufen. Die besten Ergebnisse wurden für $r = 3$ erzielt. Man beachte allerdings, dass hier jeweils die Anzahl der Räuber pro Kriterium gleich waren. Sollen beide Funktionen optimiert werden und es soll eine gute Lösung gefunden werden, die beide

Kriterien ungefähr gleich gut erfüllt, scheint hier ein relativ vielversprechender Ansatz zu existieren. Ein gutes Ergebnis zeigte sich häufig nach etwa 200000 Iterationen. Allerdings muss beachtet werden, dass sich die besten Kompromisslösungen (also Lösungen, die beide Kriteren gleich gut erfüllen) um $(1, 0)$ liegen. In den Testläufen glichen die Paretomengen Punktwolken, deren Mittelpunkt aber nicht immer in der Nähe von $(1, 0)$ lag. Stattdessen wurden Abweichungen in beide Richtungen ermittelt. Daher wurden teilweise Kriterium 1 und teilweise auch Kriterium 2 stärker bewertet. Möglicherweise ist dieses Verhalten aber auch gewünscht. Auf jeden Fall zeigen mehrere Testläufe die unterschiedlichen Verhaltensmuster ungefähr gleich oft.

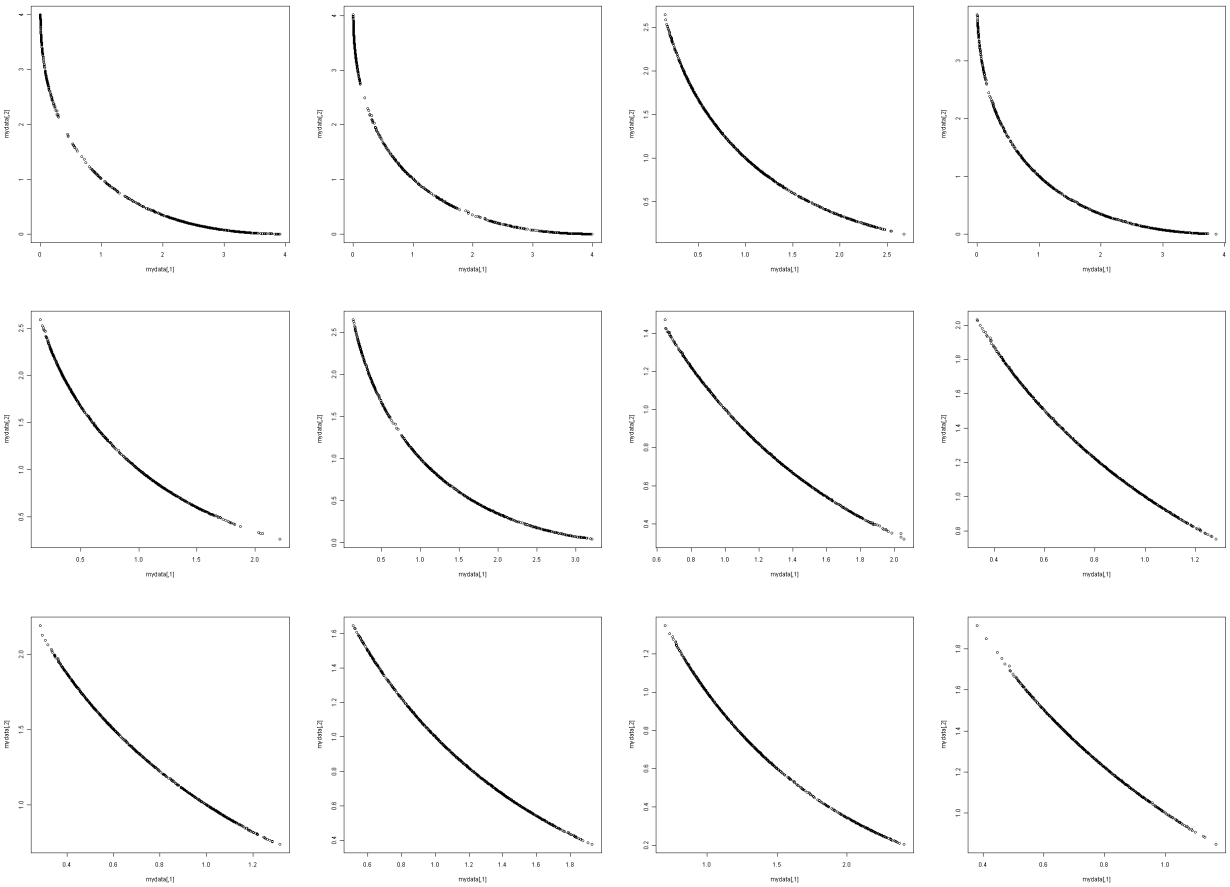

Abbildung 42: Die Paretofronten für die Radien $r = 2$ (oben, links: schlechte Ergebnisse, rechts: bessere Ergebnisse), $r = 3$ (Mitte) und $r = 4$ (unten).

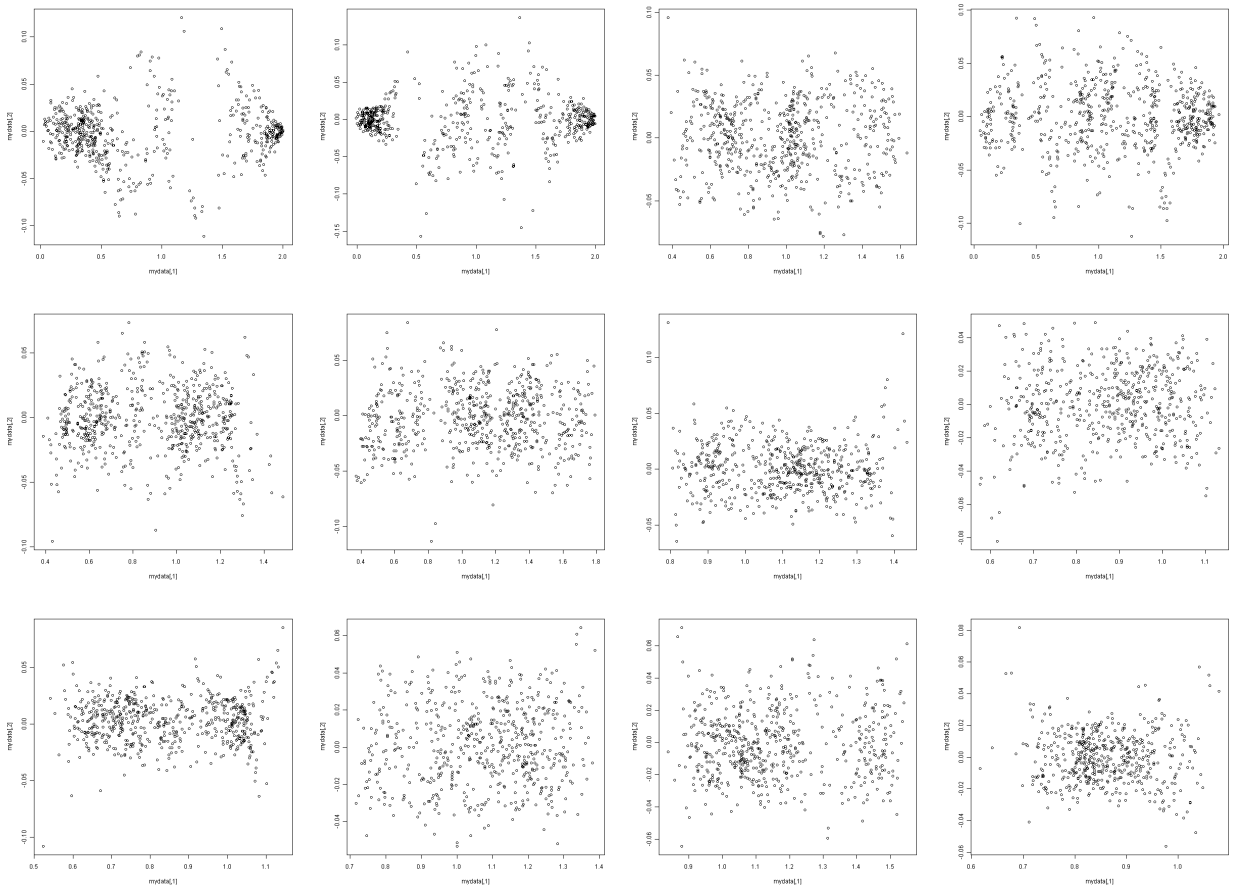

Abbildung 43: Die Paretomengen für die Radien $r = 2$ (oben, links: schlechte Ergebnisse, rechts: bessere Ergebnisse), $r = 3$ (Mitte) und $r = 4$ (unten).

Protokoll: Untersuchung von MultiSphere (Versuch 6)

Persönliche Daten:

Name: Tobias Pröger

Email: TobiasProeger@t-online.de

Problemstellung:

Einleitung: Wir untersuchen, wie sich die Variation verschiedener Parameter des Räuber-Beute-Modells auf die berechneten Ergebnisse auswirkt. Insbesondere werden die Auswirkungen der Variation von Torusgröße, Mutationsschrittweite, verschiedene Bewegungsarten sowie das Verhalten bei einer unterschiedlichen Anzahl von Räubern untersucht.

Fitnessfunktionen:

$$f_1 : x \mapsto x^2 + y^2$$
$$f_2 : x \mapsto (x-2)^2 + y^2$$

Hypothese: Sind von jedem Räuber-Typen nicht gleich viele vorhanden, werden – auf lange Zeit gesehen – nur Lösungen gefunden, die der Extremallösung der Räuber, die in der Überzahl sind, nahe sind.

Testproblem: de.irf.it.pg527.runtime.evaluator.Multisphere.MultisphereTestProblem

Objectives:

- de.irf.it.pg527.individual.properties.objectives.Objective1
- de.irf.it.pg527.individual.properties.objectives.Objective2

Chromosomen: de.irf.it.pg527.individual.properties.chromosomes.DoubleArrayChromosome mit den Einstellungen

- numberOfAlleles = 2
- upperbound = 10
- lowerbound = -10

Systemeinstellungen:

Strategie der Engine: de.irf.it.pg527.runtime.SimpleSerialStrategy

Iterationen: 10 unabhängige Durchläufe mit je 350000 Iterationen je Durchlauf

Modelleinstellungen:

Torus:

Anzahl der Positionen: $40 \times 40 = 1600$

Aufbau: Doppelt verketteter Torus

Positioning: de.irf.it.pg527.world.SimplePositioning

Räuber:

1. Gruppe: Predator ID1

Species: de.irf.it.pg527.individual.properties.species.PredatorSpecies

Anzahl: 2

Verhalten: Als Behaviour wird die Klasse de.irf.it.pg527.individual.behaviour.CommonBehaviour verwendet. Die Bewegung findet zufällig gleichverteilt mit Distanzen 1, 2 und 3 statt. Weiterhin wird die Klasse de.irf.it.pg527.individual.behaviour.action.consumption.ElitistMaximumConsumption als Consumption verwendet. Für die Reproduktion wird de.irf.it.pg527.individual.behaviour.action.reproduction.ControlledReproduction benutzt. Inbesondere wird zur Variation der Operator de.irf.it.pg527.individual.behaviour.action.reproduction.variation.TestDAChromosomeMutation mit Parameter distance = 0.03 benutzt. Dies addiert einen zufälligen Vektor aus $\mathbb{R}^2$, dessen Komponenten gemäß der Gauß-Verteilung mit $\mu = 0$ und $\sigma = 0,03$ gewählt werden. Initial positioniert wird der Räuber zufällig.

Fitnessfunktion: f_1

2. Gruppe: Predator ID2

Species: de.irf.it.pg527.individual.properties.species.PredatorSpecies

Anzahl: 1

Verhalten: Analog zu Räuber 1.

Fitnessfunktion: f_2

Beute:

1. Gruppe: Prey ID1

Species: de.irf.it.pg527.individual.properties.species.PreySpecies

Anzahl: 1600

Initiale Chromosomen: Zufällig gewählt

Evaluation:

Die Evaluation erfolgt nach Systemstart und danach alle 2500 Iterationen bis das Abbruchkriterium erfüllt ist.

Interpretation und Diskussion:

Tatsächlich bestätigt sich die Hypothese. Es wurden unabhängige Läufe durchgeführt, bei denen auch die Schrittweite beim Laufen des Räubers variiert wurde. Bei einer gleichen Anzahl von Räubern führte eine Erhöhung der Schrittweite (etwa auf $r = 3$) dazu, dass viele gute Kompromisslösungen gefunden wurden. Hier zeigt sich allerdings, dass nach einiger Zeit nur noch Lösungen vorhanden sind, die in der Nähe der Extremallösung liegen, die für die Räuber in der Überzahl gut ist. Es ist also grundsätzlich sinnvoll, die Anzahl der Räuber pro Kriterium bei MultiSphere gleich groß zu wählen, selbst wenn ein bestimmtes Kriterium bevorzugt werden soll. Ansonsten werden auf lange Zeit gesehen wieder nur Lösungen gefunden, die der Extremallösung nahe sind.

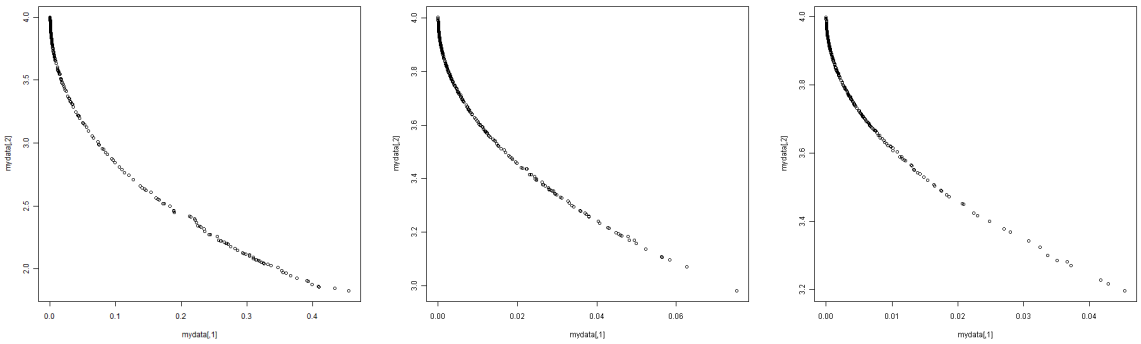

Abbildung 44: Die Paretofronten nach 350000 Iterationen für Radius $r = 1$ (links), $r = 2$ (Mitte), $r = 3$ (rechts).

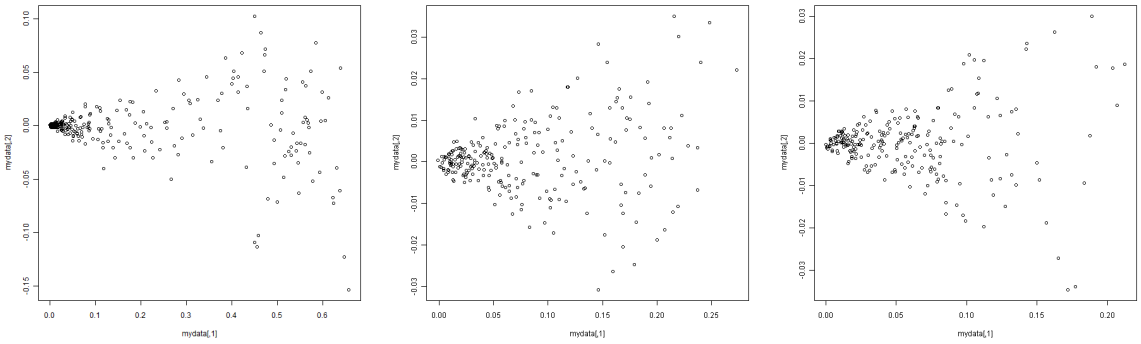

Abbildung 45: Die Paretomengen nach 350000 Iterationen für Radius $r = 1$ (links), $r = 2$ (Mitte), $r = 3$ (rechts).