jMole

A Genetic Approach to the Australia-Problem

Jochen Christ  
Daniel Kimmig  
Benjamin Kühnlenz

Inhalt

[Einleitung 3](#_Toc174676405)

[Das Australia-Problem 3](#_Toc174676406)

[Der Algorithmus 3](#_Toc174676407)

[Der Algorithmus im Überblick 3](#_Toc174676408)

[Die Codierung 3](#_Toc174676409)

[Der Strafkosten-Ansatz 3](#_Toc174676410)

[Das Erzeugen der Startpopulation 3](#_Toc174676411)

[Die Selektion 3](#_Toc174676412)

[Die Rekombination 3](#_Toc174676413)

[Die Mutation 3](#_Toc174676414)

[Optimierung der Ergebnisqualität 3](#_Toc174676415)

[Performance 3](#_Toc174676416)

[Die GUI 3](#_Toc174676417)

[Die GUI 3](#_Toc174676418)

[Ergebniswerte 3](#_Toc174676419)

[Ergebnisse Holmberg 3](#_Toc174676420)

[Ergebnisse Boccia 3](#_Toc174676421)

[Verhalten bei Änderungen 3](#_Toc174676422)

[Änderung der Strafkosten 3](#_Toc174676423)

[Änderung der Populationsgröße 3](#_Toc174676424)

[Änderung der Generationenanzahl 3](#_Toc174676425)

[Änderung der Selektionsmethode 3](#_Toc174676426)

[Lizenz 3](#_Toc174676427)

[LGPL 3](#_Toc174676428)

# Einleitung

Jochen

jMole ist eine tolle Sache. Wir können hier jetzt tolle sachen hin schreiben. Wenn wir wollen zumindest.

Ein neuer Absatz steht hier.

Hier steht dann auch ein bisschen was dazu woher der Name jMole kommt und was denn das Australia-Problem ist...

# Das Australia-Problem

Beschreibung des Problems.

Der Algorithmus

Im Folgenden wird der Algorithmus und die Details vorgestellt.

# Der Algorithmus im Überblick

Die Genetische Algorithmus

# Die Codierung

Nachdem im vorherigen Abschnitt der Algorithmus vorgestellt wurde, steht im folgenden Abschnitt die Codierung im Vordergrund. Eine Codierung dient dazu, Informationen zu formulieren. Ein Beispiel aus der Biologie ist die Codierung von Informationen in den Genen eines Lebewesens, die zur Bildung von spezifischen Proteinen herangezogen werden.

Auch beim Australia-Problem existieren Informationen, die mittels einer geeigneten Codierung formuliert werden müssen, damit sie maschinenlesbar werden und entsprechend verarbeitet werden können. Hierbei sticht ganz besonders die Information heraus, welcher Kunde von welchem Lager versorgt werden soll. Durch die Auswahl einer möglichen Technik zur Codierung ergeben sich Vor- und Nachteile für beispielsweise die Verständlichkeit und Komplexität, aber auch für den Aufwand zur Implementierung von Operatoren für bspw. die Rekombination. Die folgende Abbildung zeigt ein Beispiel für die gewählte Codierung, um die Zuordnung von Kunden zu Lagern ausdrücken zu können.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **5** | **2** | **4** | **1** | **2** | **5** | **3** | **7** | **7** | **2** | **…** |  |  |  |  |  | **m** |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | … |  |  |  |  |  | n |

dHierbei wird ein Feld (Array) veranschaulicht, dass dem Index n die verschiedenen Werte m zuordnet. Der Index steht bei der gewählten Codierung für die Kunden und ergibt sich aus den Werten der jeweiligen Probleminstanz. Der Wert an der n-ten Stelle entspricht dem Lager, zu dem der Kunde n zugeordnet wird. Auf diese relativ simple Art und Weise kann Konsistenz sichergestellt werden, d.h., dass ein Kunde immer zu einem Lager zugeordnet ist.

# Der Strafkosten-Ansatz

as

# Das Erzeugen der Startpopulation

benni

Greedy, A bit Greety and Random

# Die Selektion

jc

Eltern auswählen oder Kinder aussortieren…

# Die Rekombination

jc

lalala

# Die Mutation

jc

simple mutation, nearNeighbor

# Optimierung der Ergebnisqualität

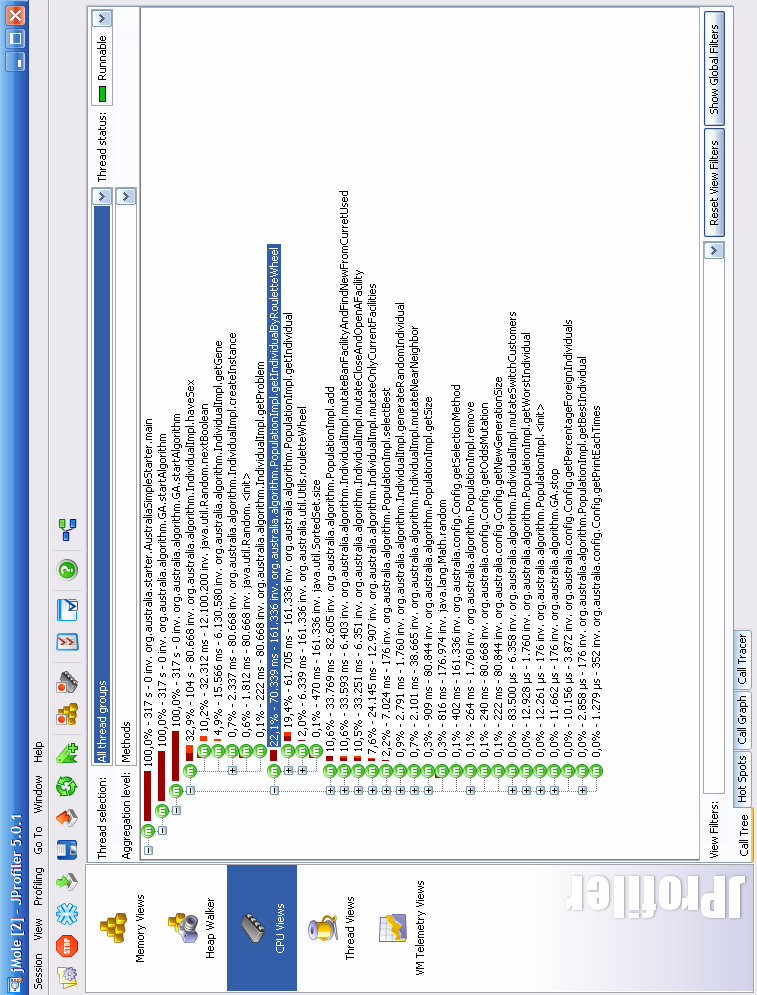
jc

Verwendung der Datenbank und Einspielen bekannter Optima.

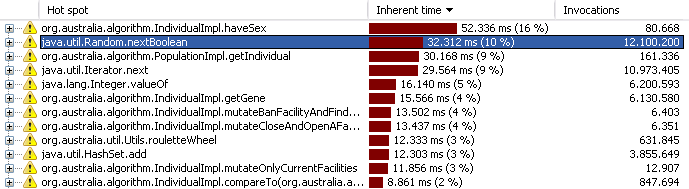
# Performance

Im Bereich Software Engineering ist Performance ein zentraler Begriff und kann daher in vielen Anforderungsspezifikationen wieder-gefunden werden. Darin wird die Performanz eines Systems oft mit quantifizierbaren Messgrößen wie Zeitverbrauch oder einem bestimmten Durchsatz beschrieben. Diesen Zielvorstellungen steht die Notwendigkeit gegenüber, die Leistungsfähigkeit eines Systems zu analysieren, wodurch detailierte Informationen über bspw. mögliche Flaschenhälse gewonnen werden können. Anhand dieser Informationen kann man Veränderungen am System vornehmen, die evtl. zu einer Leistungssteigerung führen. Ist dies nicht ausreichend, können die Ergebnisse der Leistungsmessung als Grundlage für einen neuen konzeptionellen Entwurf des Systems dienen.

Gerade bei Algorithmen, wie dem vorliegenden Algorithmus zur Lösung des Australia-Problems, die keine direkte Abhängigkeit zu Drittsystemen wie bspw. Datenbanken haben, und nicht durch sich wiederholende I/O-Prozesse in ihrer Leistungsfähigkeit beschränkt sind, ist es ratsam eine Leistungsmessung durchzuführen. Der Grund dafür liegt in der Tatsache, dass bis auf die Kerneinheiten des Rechners (Bsp.: CPU) keine Abhängigkeit zur Leistungsfähigkeit der Umgebung besteht und dadurch ein großes Potential für Optimierungen in der Konzeption des Algorithmus gegeben ist. Um aber sinnvolle Entscheidungen treffen zu können bedarf es einem soliden Werkzeug zur Leistungsmessung, da ansonsten die Gefahr besteht, dass die gewonnen Informationen falsch sind bzw. die Leistung nur in geringem Maße beeinflussen. Für Java-Applikationen hat sich solch ein Werkzeug, der sog. JProfiler, bereits etabliert und findet breite Anwendung in der Praxis. Die folgende Abbildung zeigt einen Auschnitt aus der Benutzerschnittstelle von JProfiler bei der Leistungsmessung von jMole.



In der Abbildung wird veranschaulicht, wie mittels JProfiler das Laufzeitverhalten von jMole analysiert werden kann. Von besonderem Interesse sind dabei die Funktionen, die entweder am längsten dauern oder am häufigsten aufgerufen werden. Daraus lassen sich am deutlichsten Bereiche erkennen, die sich positiv auf die Leistung auswirken. JProfiler unterstützt dies durch eine Auflistung dieser sogenannten „Hot spots“, also kritischen Bereichen der Applikation. Die folgende Abbildung zeigt ein Beispiel aus der Benutzerschnittstelle von JProfiler zur Auflistung von Hot-spots aus jMole.



Durch diese Information konnte jMole schrittweise optimiert werden, da sich Fragen über das Konzept und die Struktur des Algorithmus im Groben, sowie über Details der Implementierung einzelner Methoden nach jedem Durchlauf aufs Neue gestellt haben.

Hierfür lassen sich einige Beispiele aus der Entwicklung von jMole nennen. Der zentrale Strafkosten-Ansatz hat sich ergeben, als durch JProfiler herausgefunden wurde, dass die Funktionen zur Garantie der Zulässigkeit der Individuen einen sehr schlechten Einfluss auf die Leistung hatten. Darüber hinaus haben sich Experimente ergeben bezüglich der Nutzung unterschiedlicher Collection-Klassen (Bsp.: TreeSet) aus dem Java Development Kit (JDK) um gewisse Zugriffsmodalitäten erfüllen zu können. Allerdings ist den verschiedenen Gedankengängen zur Optimierung an gewissen Stellen eine Grenze gesetzt. Beispielsweise finden sich weit oben in der Rangliste der häufigst aufgerufenen bzw. der zeitlich längsten Funktionen einige Utility-Methoden des JDKs, auf die der Algorithmus letztlich angewiesen ist und nicht „weg-optimiert“ werden können.

Die GUI

# Die GUI

Console

und unsere Supaduba-Swing-GUI -> Hier kann Daniel ganz viel reinschreiben.

Ergebniswerte

# Ergebnisse Holmberg

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Instance** | **jMole** | **Computational Optima** | **Deviation (absolute)** | **Deviaton (relative)** |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |

# Ergebnisse Boccia

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Instance** | **jMole** | **Computational Optima** | **Deviation (absolute)** | **Deviaton (relative)** |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |
| **p1** | 8848 | 8848 | 0 | 0.0 % |
| **p2** | 1234 | 1234 | 4 | 10.2 % |

Verhalten bei Änderungen

# Änderung der Strafkosten

jc

p43

# Änderung der Populationsgröße

a

# Änderung der Generationenanzahl

jc

a

# Änderung der Selektionsmethode

In diesem Abschnitt wird die Entwicklung der Fitness bei Veränderung der Selektionsmethode betrachtet. Unter Selektionsmethode versteht man dabei u.a. die Auswahl der Eltern für eine Rekombination. Aber auch die Größe der neu zu erzeugenden Individuen wird darunter gefasst. Hierfür werden die bereits in den vorherigen Untersuchungen beschriebenen Startparameter verwendet, um eine einheitliche Betrachtungsweise zu ermöglichen. Außerdem werden fünf Stichproben pro Veränderung der Selektionsmethode genommen, damit Ausreißer besser ausgeglichen werden können. Die einzigen Variablen in dieser Betrachtung sind ersten die Auswahl der Eltern („Roulette“, „Random“) und das Verhältnis an Kindern zu den Eltern (1-5). Die folgende Tabelle gibt einen Überblick über die erzielten Werte bei Veränderung der Selektion der Eltern.



Lizenz

# LGPL

LGPG