Lösung des Traveling-Salesman-Problem mithilfe einer parallelisierten Anwendung der Optimierung durch den Ameisen-Algorithmus

Studienarbeit

des Studienganges Angewandte Informatik an der
Dualen Hochschule Baden-Württemberg Mosbach

von
Viktor Rechel

Bearbeitungszeitraum	2 Semester; 6 Monate
Matrikelnummer, Kurs	6335802, Inf15A
Hochschule	DHBW Mosbach
Gutachter der Dualen Hochschule	Dr. Carsten Müller

8. November 2017

Abstract

Zusammenfassung

Inhaltsverzeichnis

Abkürzungsverzeichnis

ACO Ant Colony Optimization oder Ameisenalgorithmus

TSP Traveling Salesman Problem

UML Unified Modeling Language

ER Entity Relationship

COCOMO Constructive Cost Model

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

2 Stand der Technik & Forschung

2.1 Traveling Salesman Problem

2.2 Ant Colony Optimization

3 Konzeptionierung

Um eine präzise und effiziente Umsetzung dieser Arbeit zu gewährleisten, wird zu Beginn der Fokus auf eine durchgängig durchdachte Kozeptionierung gelegt. Dies beginnt bereits bei der Definition der Anforderungen mithilfe des Requirements Engineering. Die definierten auszuarbeitenden Punkte werden im Anschluss mithilfe des COCOMO bewertet.

Nachdem die einzelnen Punkte definiert und bewertet sind, wird eine möglichst effiziente und passende Architektur entworfen. Auf diese wird ausführlich eingegangen, indem die Idee hinter dem Aufbau erklärt wird, sowie auch durch ein UML-Diagramm dargestellt wird.

Im Anschluss werden auch einzelne Methoden vorgestellt, sowie alle Parameter die die Lösung beeinflussen beschrieben.

In einer heutigen Software-Architektur sind die SOLID-Prinzipien nicht mehr weg zu denken. Diese werden in der Architektur beachtet, umgesetzt und auch in dieser Arbeit beschrieben.

3.1 Software Engineering

Innerhalb einer Software-Entwicklung - wie im vorliegenden Fall -

3.1.1 Requirements Engineering

3.1.2 COCOMO

3.2 Architektur

Die Architektur wurde inhaltlich so aufgeteilt, dass eine modulare Implementierung möglich ist. So wurden die vier folgenden Bereiche definiert: Persistenz, Applikation, TSP und ACO

Der Begriff der Persistenz beschreibt in diesem Fall zusätzlich zum dauerhaften Abspeichern der Daten, auch das Einlesen der verschiedenen Problemstellungen.

Applikation umfasst den Teil des Programms, der sich nicht direkt mit Daten befasst aber auch nicht zur Problemlösung beiträgt, wie die zwei folgenden Bereiche.
Innerhalb des Begriffs TSP werden alle nötigen Parameter und Methoden behandelt, die basierend auf dem Traveling Salesman Problem notwendig werden.
Zuletzt gibt es in der Architektur noch das Feld ACO, welches die komplette Berechnung des zu lösenden Problems übernimmt. In dem Fall, dieser Arbeit handelt es sich um das TSP. Allerdings könnte das Problemfeld auch dadurch ausgewechselt werden, dass der Bereich TSP um das neue Problem ersetzt wird.

3.2.1 Persistenz

Um ein stares Definieren des Problems innerhalb der Applikation zu verhindern, werden zu Beginn des Programms alle Parameter, wie Städtematrix, Wahrscheinlichkeiten und Lösungsparameter aus einer gegebenen XML-Datei eingelesen. Durch eine Bearbeitung der XML ist ein einfaches Abändern der Problemstellung möglich. Hierdurch ist auch gesichert, dass die Algorithmen effizient getestet werden können, da mehrere verschiedenen bekannte Testwerte benannt werden können.

3.2.2 Applikation

Wie bereits genannt, liegt bei der vorliegenden Architektur ein Fokus auf Modularität, Portabilität und Usability. Aber auch auf verlässliche und effiziente Algorithmen muss geachtet werden. Daher wird für die Wahrscheinlichkeitsberechnung die externe Klasse MersenneTwisterFast [1] gentutzt, welche eine bessere Distribution der Pseudozufallszahlen bieten als die Default-Implementierung in Java. [2] Ein weiterer Bestandteil des Applikationsbereichs werden das Logging der Arbeitsvorgänge, sowie die zentrale Konfiguration der Problemstellung auf der die Lösung basiert.

3.2.3 TSP

Der Bereich des TSP definiert sich in dieser Architektur rein durch die Städte-Objekte, welche zur Darstellung der Städtematrix genutzt werden. Der einzige andere Bestandteil ist die zentrale Aufstellung der Städtematrix, die von den restlichen Klassen nur kopiert wird.

[1] s. http://www.math.sci.hiroshima-u.ac.jp/ m-mat/MT/emt.html
[2] Um eine einfache und schnelle Benutzung zu gewährleisten, wird in dieser Arbeit darauf verzichtet echte Zufallszahlen zu nutzen, die beispielsweise aus Weltallstrahlung berechnet werden. Diese seien hier nur zur Vollständigkeit halber erwähnt.

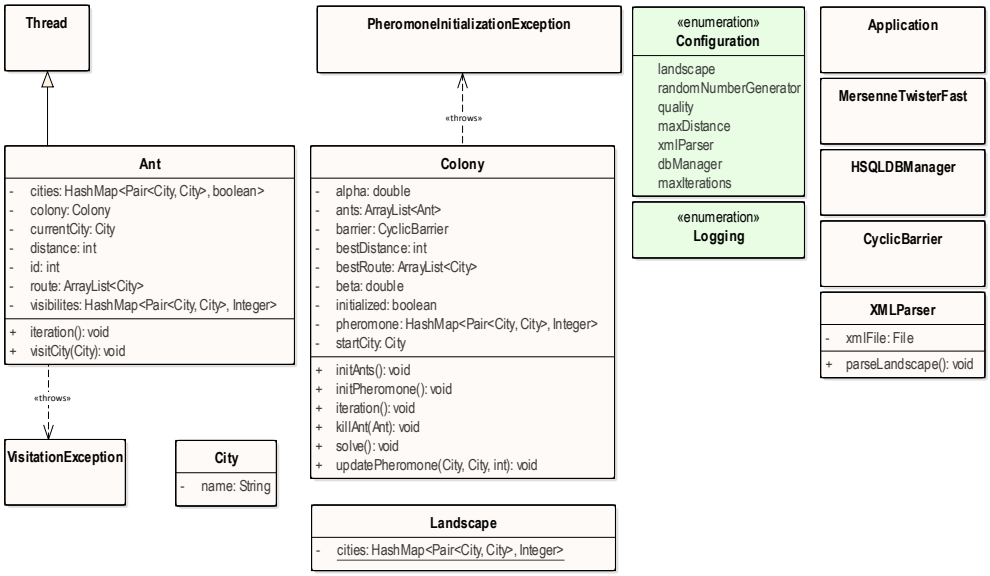

Abbildung 3.1: UML-Modellierung des Architekturentwurfs

3.2.4 ACO

Um den Ameisenalgorithmus umzusetzen sind deutlich mehr Aufwände nötig, als zur Darstellung des TSP. Hier werden mindestens eine Ameisenkolonie benötigt [3], sowie je Kolonie mehrere Ameisen.
Die Ameisen werden in der Applikation als einzelne Threads gestartet, die über eine CyclicBarrier kontrolliert werden. Dies erlaubt ein möglichst effizientes Parallelisieren der Lösung.

3.3 UML-Diagramm

Um an der Planung auch visuell arbeiten zu können, wurde diese zu Beginn als UML-Diagramm erarbeitet. Wie in Abbildung 3.1 zu sehen ist, reichen für die Implementierung eine wenige Klassen aus. So wird der Teil des Programms, welcher das TSP-Problem darstellt nur durch die Klassen *Landscape* und *City* vertreten. *Landscape* beinhaltet die Matrix der Städte inklusive der Distanzen zwischen den Städten.

[3] Die Architektur würde bei ausreichenden Systemressourcen auch eine parallele Berechnung mehrerer Städtematrizen erlauben

3.4 Umsetzung der SOLID-Prinzipien

Um eine saubere und übersichtliche Implementierung gibt es heutzutage eine Vielzahl von Regelwerken, Anleitungen und Vorgaben. Eine Sammlung von Grundsätzen sind die SOLID-Prinzipien. SOLID steht für **S**ingle Responsibility, **O**pen/Closed, **L**iskov Substitution, **I**nterface Segregation und **D**ependency Inversion. Alle diese Eigenschaften werden im Folgenden kurz erklärt und dann ihre Verwirklichung in der Architektur gezeigt.

3.4.1 Single Responsibility

Das Single-Responsibility-Prinip umschreibt den Zustand, dass Klassen max. eine Zuständig haben sollen. Klassen dürfen nicht mehrere Aufgabengebiete gleichzeitig übernehmen, da hierdurch eine Änderung an dieser Klasse zu einer Änderung mehrerer Komponenten führt und dadurch die ganze Struktur beeinflusst werden kann. Stattdessen wird die Software in mehrere kleinere Klassen aufgeteilt, die jeweils einen Teil übernehmen. Hierdurch ist zum Einen die Struktur einfacherer erkennbar und nachvollziehbar, da die Klassen eindeutiger definiert sind und der Code besser zusammengefasst ist. Zum Anderen sind die Abhängigkeiten aber auch auf ein Minimum reduziert.

In der vorliegenden Konzeption ist dieser Grundsatz dadurch erfüllt, dass die Berechnungslogik an die Ameisen ausgegliedert ist, welche selbst keine Manipulation vornehmen. Die Ameisenkolonie hingegen funktioniert nur als Quelle der Threads und behält einige wenige Kolonie-spezifische Attribute bereit, sonst nichts. Andere Anwendungen, wie das Logging, das Einlesen von Konfigurationsdateien und die Zufallszahlerzeugung wurde alle in getrennte Klassen ausgegliedert, die zentral in *Configuration* initialisiert werden.

3.4.2 Open / Closed

Die Open-Closed-Eigenschaft, welche eine moderen Software besitzen muss, umschreibt den Umstand, dass ein möglichst modularer Aufbau genutzt wird. So sollen Klassen so aufgebaut sein, dass bei einer Erweiterung keine Änderung bestehenden Codes notwendig wird, sondern zum Beispiel über das Implementieren eines Interfaces die alten Strukturen genutzt werden können. Unterscheiden muss man hier zwischen einer Erweiterung und einer Änderung. Die Software muss nicht und soll auch nicht auf eine einfache Änderung ausgelegt sein. Der Code, welcher erfolgreich implementiert wurde, soll auch in seinem Funktionsumfang weiter genutzt werden. Die bestehenden Implementierungen müssen keine Schnittstellen zur einfach Änderung enthalten.

Durch die Schlichtheit der entworfenen Architektur und den auf die Problemstellung zugeschnittenen Charakter ist eine Umsetzung des Open-Closed-Prinzips nur in geringem Umfang möglich. Alleine die Parser können hiernach umgesetzt werden, in dem

ein Interface eingesetzt wird. Dadurch wird ermöglich in Zukunft auch andere Datei-Typen akzeptieren zu können. In der restlichen Implementierung ist eine Umsetzung nicht hilfreich oder zweckführend.

3.4.3 Liskov Substitution

Die Substitutionsregel von Liskov besagt, dass es möglich sein sollte eine Klasse in einem beliebigen Aufruf auch durch eine Subklasse aus zu tauschen ohne den Programmablauf zu ändern. Folglich müssen entweder die Implementierung abgestimmt sein, dass ein Austauschen generell möglich ist. Dies ist aber meist nicht zweckdienlich. Andererseits ist es auch möglich, über eine abstrakte Klasse zu arbeiten, die für Methodenaufrufe benutzt wird. So wird der Programmablauf nicht durch unterschliede Klassentypen unterbrochen, sondern der Entwickler ist in der Pflicht die abstrakte Klasse entsprechend umzusetzen.

Ähnlich zu der Open-Closed-Eigenschaft ist auch diese Leitlinie in dem vorliegenden Entwurf nur sehr schwierig umzusetzen. Da von ein Hineinquetschen von Regeln und Leitlinien in eine Architektur generell abzuraten ist, wurde auf eine Umsetzung der Substitutionsregel generell verzichtet.

3.4.4 Interface Segregation

Die Trennung der Interfaces zielt darauf ab, Klassen nicht dazu zu zwingen Methoden zu implementieren, die gar nicht benötigt werden. So müssen in den meisten Programmiersprachen alle Methoden eines Interfaces zwingend umgesetzt werden. Dies führt aber meist zu einer aufgeblähten Codestruktur, da unnötige Methoden implementiert werden müssen, aber nie benutzt werden. Indem man die Interfaces in kleine Interfaces unterteilt ist es möglich durch eine Implementierung von mehrerer Interfaces auf das gleiche Ergebnis zu kommen ohne den Zwang alle anderen Interface auch umzusetzen.

Aufgrung des Mangels an Interfaces in der Codestruktur der hier behandelten Architektur ist auch diese Regel nicht umgesetzt. Sobald Interface allerdings zum Einsatz kommen, muss diese Regel umgesetzt werden.

3.4.5 Dependency Inversion

Die Abhängigkeitsumkehr-Regel besagt, dass Module auf höheren Ebenen nicht auf Module niederer Ebenen angewiesen sein dürfen. Ähnlich zu den anderen Richtlinien sollen auch hier abstrakte Klassen und Interface zur Abstraktion genutzt werden. Allerdings gibt es noch den Zusatz, dass Abstraktionen niemals von einer detaillierten Implementierung abhängen dürfen.

Wie bei den anderen Interface-Umsetzungen ist auch hier eine Umsetzung nicht hilf-

reich bzw. möglich. Dennoch sei auch hier erwähnt, dass eine Benutzung der Regel zu wartbarerem Code führt.

3.5 Parameterbeschreibung

Die Berechnung der optimalen Wegstrecke läuft über eine Wahrscheinlichkeitsrechnung, die jeweils von den einzelnen Threads bzw. Ameisen in jeder Stadt aufs neue durchgeführt wird. Dabei werden die Wahrscheinlichkeiten nach folgender Formel berechnet:

$$p(s_{ij}) = \frac{\tau_{ij}^{\alpha} * \eta_{ij}^{\beta}}{\sum_{x \in N} \tau_{ix}^{\alpha} * \eta_{ix}^{\beta}} \tag{3.1}$$

s : Strecke zwischen zwei Städten

i : Quellestadt

j : Zielstadt

N : Menge der von Stadt i aus erreichbaren Städte j

τ : Pheromonwert auf der Strecke i-j

η : Heuristischer Faktor für die Strecke i-j

α, β : Innerhalb der Applikation festgelegte Parameter

3.5.1 Alpha - α

3.5.2 Beta - β

3.5.3 Tau - τ

3.5.4 Eta - η

3.6 Ausgewählte Algorithmen

Bereits vorgestellt wurden die einzelnen Bestandteile der Architektur, sowie die Architektur als Gesamtbild gezeigt. Im Folgenden sollen beispielhaft die verwendeten Algorithmen thematisch vorgestellt werden. So werden die zentralen Methoden zur Berechnung

der Iterationen aus Sicht der Ameisen vorgestellt, sowie auch das Verhalten der Pheromonänderung. Zusätzlich wird die Methode zum Töten einer Ameise beschrieben, welche in so gut wie keiner Implementierung zu finden ist.

3.6.1 Ant - iteration()

3.6.2 Colony - killAnt()

Ebenfalls in dem in Abbildung 3.1 gezeigten UML-Diagramm erkennbar ist, dass die Ameisenkolonie die Möglichkeit besitzt einzelne Ameisen zu "töten". Diese Methode sollte in einem einwandfreiem Programm keinerlei Verwendung finden, allerdings kann man sich nicht auf eine dauerhafte fehlerfreie Implementierung verlassen. Im Bereich der Software Tests ist dies durch das Prinzip "Fehlen von Fehlern"beschrieben. Dieses sagt aus, dass erfolgreiche Tests nur bestätigen, dass keine Fehler gefunden wurden. Es kann nicht ausgesagt werden, dass keine Fehler vorliegen.[4]

Denn die Methode hat die Funktion, im Falle des Fehlverhaltens eine Ameise aus der Liste der aktiven Ameisen bzw. Threads zu löschen und den Thread zu stoppen. Genutzt werden wird diese vor allem im Bereich der aufgefangenen Fehler innerhalb der Implementierung der Ameisen. Sollte eine aufgetretene Exception schwerwiegend und unlösbar sein, wird die Applikation automatisch die Funktion aufrufen.

3.6.3 Colony - updatePheromone()

3.7 Sensitivitätsanalyse

[4] vgl. **[bibid]**

4 Implementierung

4.1 Klassendiagramm

4.2 Beschreibung der Implementierung

4.3 ER-Diagramm

4.4 Komponenten-Diagramm

4.5 Paket-Diagramm

4.6 Perforamce-Analyse und -Optimierung

5 Fazit

Literaturverzeichnis

[1] B. Booba; Dr. T. V. Gopal. „Comparison of Ant Colony Optimization & Particle Swarm Optimization In Grid Scheduling“. English. In: *Australian Journal of Basic and Applied Sciences* (8. Juni 2014), S. 1–6.

[2] David P. Williamson. *The Traveling Salesman Problem: An Overview*. English. Presentation. Cornell University Ebay Research, 21. Jan. 2014.