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

4. Dezember 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

Formelverzeichnis

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 - gilt es immer zuerst eine gründliche Anforderungsanalyse durchzuführen, um sicherzustellen dass alle Anforderungen erfasst und dokumentiert sind. Ebenso müssen eventuelle unbewusste Anforderungen ebenso herausgefunden werden, wie mögliche Begeisterungsfaktoren.

Um eine zeitliche Einschätzung durchführen zu können bietet sich das Constructive Cost Modell an - kurz COCOMO. Hierbei werden der Aufwand der Implementierung geschätzt und eine ungefähre Anzahl der Codezeilen angegeben. Aus dieser Zahl kann dann per Formel eine Implementierungslaufzeit errechnet werden.

3.1.1 Requirements Engineering

Zu den grundsätzlichen Anforderungen gehören eine performante Umsetzung einer Softwarelösung des TSP, der Verwendung einer effizienten Implementierung des ACO und die Entwicklung einer durchdachten Architektur. Diese generellen Eigenschaften lassen sich aufteilen in genauer spezifizierte Anforderungen, sowohl aus funktionaler Sicht als auch aus nicht-funktionaler Sicht.

Die funktionalen Anforderungen wären somit:

- Möglichkeit das TSP lösen zu können
- Lösungsweg wird mit ACO berechnet

Diese Arbeit beschränkt sich somit rein mit der Thematik das TSP möglichst effizient und performant zu lösen. Allerdings gibt es hier auch Einschränkungen betreffend auf die Umsetzung in Form der nicht-funktionalen Anforderungen, die wie folgt lauten:

- Beachtung der ökonomischen Aspekte
- Objektorientierte Software
- Schnittstelle zum Erzeugen von Log-Daten
- Schnittstelle zum Einlesen von Problemdaten
- Berücksichtigung der SOLID-Prinzipien
- Auswahl leistungsfähiger Datenstrukturen
- Lesbarer, dokumentierter Sourcecode
- Lösungsqualität von 95
- Programmiersprache: Java 8
- Zugelassene externe Bibliothek: JUnit

3.1.2 COCOMO

Um eine wirtschaftliche Analyse in Bezug auf das vorliegende Softwareprojekt durchführen zu können, wurde das COCOMO angewendet. Hierbei wurde die Funktionspunkt-Methode angewendet, da ein Implizieren der Anzahl der Codezeilen nicht verlässlich

möglich ist. Hierbei wurden die verschiedenen Teile der Architektur nach der Komplexität bewertet und aufgeteilt. Daraus ergab sich folgende Liste:

Komponente	**Komplexität**	**Funktionspunkte**
Externe Eingaben	Niedrig	3
Externe Ausgaben	Niedrig	4
Externe Anfragen	Hoch	6
Externe Schnittstellen	Niedrig	5
Interne Logiken	Mittel	10

Tabelle 3.1: Komplexitätseinschätzung des Projekts im Rahmen einer Abschätzung der Funktionspunkte

Somit ergeben sich als Summe der Anforderungen an die Software 28 Funktionspunkte. In der verwendeten Programmiersprache Java entspricht ein Funktionspunkt im Durchschnitt ca. 53 Zeilen Code. Somit ergibt sich als Zeilenanzahl eine erwartete Menge von 1484 Zeilen.

Mithilfe der "COCOMO IIFormel lässt sich aus der Anzahl der Zeilen eine Arbeitsaufwand berechnen:

$$Aufwand = C * (Size)^{Prozessfaktoren} * M \tag{3.1}$$

C = Konstante
$Size$ = Anzahl der Codezeilen
$Prozessfaktoren$ = Kombinierte Prozessfaktoren
M = Leistungsfaktoren

Nach dieser Formel beträgt der zeitliche Aufwand für dieses Projekt ca. 2.5 Monate. Diese Zahl ist entsprechend einer Dauer eines Semesters von drei Monaten nachvollziehbar. Somit lässt sich auch sagen, dass das Projekt durchführbar ist und auch einen ausreichen großen Anspruch besitzt, um eine zu kurze Beschäftigung zu verhindern.

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 starres 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] genutzt, 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.

[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.

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.

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.

3.2.5 Arbeitsweise der Architektur

Die einzelnen Abschnitte der Architektur wurden bereits erklärt. Aber das Zusammenspiel der einzelnen Komponenten und der eigentliche Arbeitsablauf des Systems wurde noch nicht beschrieben. Im Folgenden wird ein Beispiel so durchgeführt, wie es auch die geplante Architektur umsetzen würden.

In Abbildung 3.1 ist ein Beispiel für das TSP gegeben. Sechs Städte sind untereinander so vernetzt, dass jede Stadt von jeder anderen erreichbar ist. Auch ist Stadt A schon rot markiert, wodurch diese als Standort für die Ameisenkolonie ausgewählt wurde.

Nun wird die gleiche Berechnung aufgezeigt, welche die Applikation durchführen würde. In Tabelle 3.2 zu sehen ist die allgemeingültige Streckenmatrix für das vorliegende Beispiel. Innerhalb der Matrix sind jeweils die Streckenlängen von einer Stadt zur Anderen gespeichert. So besitzt die Strecke von D nach C die Länge 4[4]. Auffällig sind die Strecken von den Städten zu sich selbst, welche mit -1 gekennzeichnet sind. Diese Zahl

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

[4] Hier könnte natürlich eine beliebige Längeneinheit ergänzt werden. Dies ist aber für das Lösen des Problems und das Beschreiben des Vorgehens nicht notwendig und wird deswegen weggelassen.

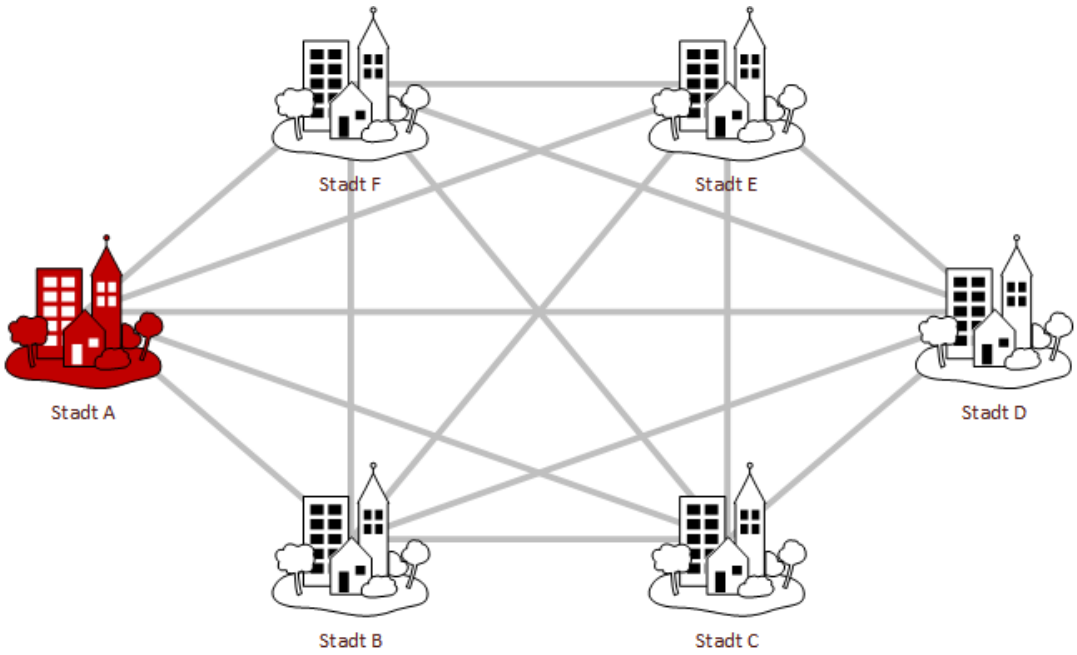

Abbildung 3.1: Darstellung des TSP-Beispiels zur Darlegung der Arbeitsweise der Architektur

	A	B	C	D	E	F
A	-1	3	9	13	11	5
B	3	-1	7	10	9	8
C	9	7	-1	4	6	10
D	13	10	4	-1	2	12
E	11	9	6	2	-1	8
F	5	8	10	12	8	-1

Tabelle 3.2: Distanzmatrix des TSP-Beispiels

wird später für die Applikation ein Hinweis sein, dass ein Fehler vorliegt und dieser korrigiert werden muss.

Besonders wichtig für die Berechnung der ACO-Lösung ist die Pheromonmatrix, welche in Tabelle 3.3 im initialisierten Zustand gezeigt ist. Die Normalisierung mit einem durchgängigen Wert von 1 bewirkt, dass im ersten Durchlauf die Pheromone keinerlei Einfluss auf die Entscheidungen der Ameisen haben. Dies entspricht dem Umstand einer neu aufgebauten Ameisenkolonie, welche sich erst zurecht finden muss.

Mit den vorliegenden Daten, der Streckenlängenmatrix und der initialen Matrix, kann nun das TSP mithilfe von ACO berechnet werden. In diesem Beispiel werden drei Ameisen verwendet, um das Verhalten zeigen zu können, aber gleichzeitig den Aufwand gering

	A	B	C	D	E	F
A	1	1	1	1	1	1
B	1	1	1	1	1	1
C	1	1	1	1	1	1
D	1	1	1	1	1	1
E	1	1	1	1	1	1
F	1	1	1	1	1	1

Tabelle 3.3: Initiale Pheromonmatrix des TSP-Beispiels

zu halten.

Jede Ameise startet bei der Kolonie, welche bei Stadt A liegt. Daher muss für jede Ameise jetzt ausgewählt werden, wohin diese gehen soll. Für jede mögliche Strecke wird nun eine Variabel berechnet, welche im Folgenden mit λ [5] bezeichnet wird. λ setzt sich zusammen aus dem Pheromonwert der betrachteten Strecke und der zugehörigen Streckenlänge.

Nun wird für jede Ameise einzeln betrachtet, welche Städte noch erreichbar sind - also noch nicht besucht sind - und welchen Wert λ für die Strecke zu dieser Stadt besitzt. λ einer Stadt wird dann mit der Summe aus den λ aller erreichbaren Städten verglichen. Hierdurch wird ein Wert - im Folgenden mit ρ referenziert - erzeugt, welcher zwischen 0 und 1 liegt.

Um nun bestimmen zu können, welche Ameise welchen Weg wählt, wird eine Zufallszahl n bestimmt, welche ebenfalls zwischen 0 und 1 liegt. Nacheinander wird für alle erreichbaren Städte nun verglichen, ob sich $\rho > n$ ergibt. Sobald eine Stadt die Gleichung erfüllt, wählt die Ameise diesen Weg und zieht weiter. Erfüllt keine Stadt diese Gleichung so werden nacheinander die ρ der Städte aufaddiert, bis eine Stadt x die Gleichung $n < \sum_x \rho$ erfüllt. Wenn diese Gleichung erfüllt ist, wählt die Ameise der Strecke zu Stadt x.

In dem vorliegenden Beispiel bedeutet das, dass von Stadt A aus alle möglichen Strecken ausgewertet werden. Folgend ist die Berechnung von λ aller erreichbaren Städte aufgezeigt.

$$A- > B \quad (1)^2 * (\frac{1}{3}) = \frac{1}{3} \tag{3.2}$$

$$A- > C \quad (1)^2 * (\frac{1}{5}) = \frac{1}{5} \tag{3.3}$$

[5] Mehr zu diesen beiden Werten in Kapitel 3.5.

$$A- > D \quad (1)^2 * (\frac{1}{9}) = \frac{1}{9} \tag{3.4}$$

$$A- > E \quad (1)^2 * (\frac{1}{13}) = \frac{1}{13} \tag{3.5}$$

$$A- > F \quad (1)^2 * (\frac{1}{11}) = \frac{1}{11} \tag{3.6}$$

Aus diesen kann nun ρ errechnet werden, in dem alle λ durch die Summe aller λ - welche 0,812 beträgt - dividiert werden. ρ steht dann für die Wahrscheinlichkeit, dass dieser Weg gewählt wird. Somit ergeben sich folgende fünf Wahrscheinlichkeiten:

$$\rho(AB) = \frac{\frac{1}{3}}{0,8122} = 0.410 \tag{3.7}$$

$$\rho(AC) = \frac{\frac{1}{5}}{0,8122} = 0.136 \tag{3.8}$$

$$\rho(AD) = \frac{\frac{1}{9}}{0,8122} = 0.094 \tag{3.9}$$

$$\rho(AE) = \frac{\frac{1}{13}}{0,8122} = 0.111 \tag{3.10}$$

$$\rho(AF) = \frac{\frac{1}{11}}{0,8122} = 0.246 \tag{3.11}$$

Nun muss für jede der drei Ameisen eine Zufallszahl bestimmt werden, welche mit ρ verglichen werden kann. Es wurden die Zahlen 0.242 für Ameise 1, 0.033 für Ameise 2 und 0.455 für Ameise 3 errechnet. Nach dem bereits beschriebenen Vorgehen wählen Ameise 1 und Ameise 2 nun die Stadt B als Zielstadt. Ameise 3 wählt Stadt C, da der Zufallswert zu groß war um eine direkte Auswahl zu treffen. Ein Aufaddieren der Werte $\rho(AB)$ und $\rho(AC)$ hat allerdings schon ausgereicht, um den Wert zu überschreiten.

Diesen Vorgehen kann nun für alle Städte wiederholt werden, sodass am Ende alle Ameisen alle Städte besucht haben und auch wieder in der Anfangsstadt in der Kolonie

Ameise	Weglänge	Wegstrecke
1	40	A, B, F, D, E, C, A
2	36	A, B, F, E, D, C, A
3	50	A, C, D, B, F, E, A

Tabelle 3.4: Ergebnisse der drei Ameisen der ersten Generation

	A	B	C	D	E	F
A	1	$\frac{5}{3}$	$\frac{10}{9}$	1	1	1
B	1	1	1	1	1	$\frac{5}{4}$
C	$\frac{11}{9}$	1	1	$\frac{5}{4}$	1	1
D	1	$\frac{11}{10}$	$\frac{5}{4}$	1	$\frac{3}{2}$	1
E	$\frac{12}{11}$	1	$\frac{7}{6}$	$\frac{3}{2}$	1	1
F	1	1	1	$\frac{13}{12}$	$\frac{5}{4}$	1

Tabelle 3.5: Pheromonmatrix des TSP-Beispiels nach dem ersten Durchlauf

angekommen sind. Nun muss noch die Pheromonmatrix aktualisiert werden, um der Kolonie mitzuteilen welche Wege profitable sind. Hierbei wird von jeder Ameise auf jedem Weg den sie ablaufen konstant Pheromone abgegeben. Dadurch ergibt sich als zusätzlicher Pheromonwert die der Kehrwert der Streckenlänge. Der Kehrwert wird auf den aktuellen Wert in der Pheromonmatrix addiert, wodurch die nächste Generation diese in die Berechnung einbeziehen kann.

In Tabelle 3.4 zu erkennen sind die kompletten Wegstrecken, die die Ameisen gewählt haben. Aus dieser Liste ableiten lassen sich nun die zusätzlichen Pheromonwerte, in dem man die Tabelle 3.2 miteinbezieht. Addiert man die zusätzlichen Pheromonwerte auf die vorherige Pheromonmatrix, so erhält man die in Tabelle 3.5 gezeigte neue Pheromonmatrix.

Nach der Berechnung der neuen Pheromonmatrix ist die erste Generation der Ameisen abgeschlossen. Nun werden sich drei neue Ameisen auf die Reise begeben. Da die Pheromonmatrix nicht mehr normalisiert ist, sondern von 1 abweichende Werte enthält, können die neuen Ameisen die Pheromonmatrix effektiv in die Berechnung der Wahrscheinlichkeiten miteinbeziehen. So wird beispielsweise λ für die Strecke A - B nun mit folgender Gleichung berechnet:

$$A -> B \quad (\frac{5}{3})^2 * (\frac{1}{3}) \tag{3.12}$$

Hierbei wird nun das Verhältnis zwischen Pheromonen und Streckenlänge betrachtet, wobei die Pheromone doppelt so schwer gewichtet werden. Die restliche Berechnung läuft

Ameise	Weglänge	Wegstrecke
1	40	A, C, E, D, B, F, A
2	29	A, F, E, D, C, B, A
3	41	A, F, E, D, B, C, A

Tabelle 3.6: Ergebnisse der drei Ameisen der zweiten Generation

	A	B	C	D	E	F
A	1	$\frac{5}{3}$	$\frac{11}{9}$	1	1	$\frac{7}{5}$
B	$\frac{4}{3}$	1	$\frac{8}{7}$	1	1	$\frac{11}{8}$
C	$\frac{4}{3}$	$\frac{11}{10}$	1	$\frac{5}{4}$	$\frac{7}{6}$	1
D	1	$\frac{13}{10}$	$\frac{3}{2}$	1	$\frac{3}{2}$	1
E	$\frac{12}{11}$	1	$\frac{7}{6}$	3	1	1
F	$\frac{6}{5}$	1	1	$\frac{13}{12}$	$\frac{3}{2}$	1

Tabelle 3.7: Pheromonmatrix des TSP-Beispiels nach dem zweiten Durchlauf

allerdings parallel zum vorherigen Vorgehen ab. Die zweite Generation der Ameisen hat, wie man aus dem Vergleich zwischen Tabelle 3.4 und Tabelle 3.6 erkennt, bei einigen Strecken eine andere Wahl getroffen. Im Durchschnitt ist neue Generation auch schneller voran gekommen, was an den Weglängen - also der Summe aller gelaufenen Strecken - erkennbar ist.

Auch diese Generation verteilt auf allen besuchten Strecken ihre Pheromone, was wieder zu einer Aktualisierung der Pheromonmatrix führt. In Tabelle 3.7 zu erkennen ist, dass nun deutlich mehr Streckenabschnitte besucht wurden und einen von 1 abweichenden Pheromonwert besitzen.

Nach der zweiten Generation macht sich noch eine letzte Generation der Ameisen auf den Weg und läuft wieder die gleichen Orte ab. Wieder wird die vorher aktualisierte Pheromonmatrix in die Gewichtung miteinbezogen, um ein effizienteres Ergebnis zu erhalten. Betrachtet man das in Tabelle 3.8 gezeigte Ergebnis der Ameisen, so ist deutlich sichtbar dass eine Verbesserung vorliegt. Alle Ameisen kamen schneller voran, als ihre Vorgänger.

Abschließend lässt sich aus der finale Pheromonmatrix - in Tabelle 3.9 gezeigt - er-

Ameise	Weglänge	Wegstrecke
1	35	A, B, C, E, D, F, A
2	33	A, B, E, D, C, F, A
3	29	A, B, C, D, E, F, A

Tabelle 3.8: Ergebnisse der drei Ameisen der dritten Generation

	A	B	C	D	E	F
A	1	$\frac{8}{3}$	$\frac{11}{9}$	1	1	$\frac{7}{5}$
B	$\frac{4}{3}$	1	$\frac{10}{7}$	1	$\frac{10}{9}$	$\frac{11}{8}$
C	$\frac{4}{3}$	$\frac{11}{10}$	1	$\frac{3}{2}$	$\frac{8}{6}$	$\frac{11}{10}$
D	1	$\frac{13}{10}$	$\frac{7}{4}$	1	2	$\frac{13}{12}$
E	$\frac{12}{11}$	1	$\frac{7}{6}$	4	1	$\frac{9}{8}$
F	$\frac{9}{5}$	1	1	$\frac{13}{12}$	$\frac{3}{2}$	1

Tabelle 3.9: Pheromonmatrix des TSP-Beispiels nach dem finalen dritten Durchlauf

kennen, dass zum Einen ein Großteil der Strecken besucht wurde. Zum Anderen liegt bereits nach drei Generationen teilweise eine deutliche Gewichtung vor, sodass bei Stadt E meist nur noch die Stadt D gewählt wird. Es lässt sich bereits bei dieser manuellen Berechnung schlussfolgern, dass mehr Ameisen lediglich bedeuten, dass die Pheromonverteilung schneller angepasst und optimiert wird. Dies hat zur Folge, dass bei höherer Ameisenzahl weniger Generationen benötigt werden, um ein besseres Ergebnis zu erhalten. Allerdings hat es keinen Einfluss auf das Endergebnis.

3.3 UML-Diagramm

Um an der Planung auch visuell arbeiten zu können, wurde diese zu Beginn als UML-Diagramm erarbeitet. Wie in Abbildung 3.2 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.

3.4 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.

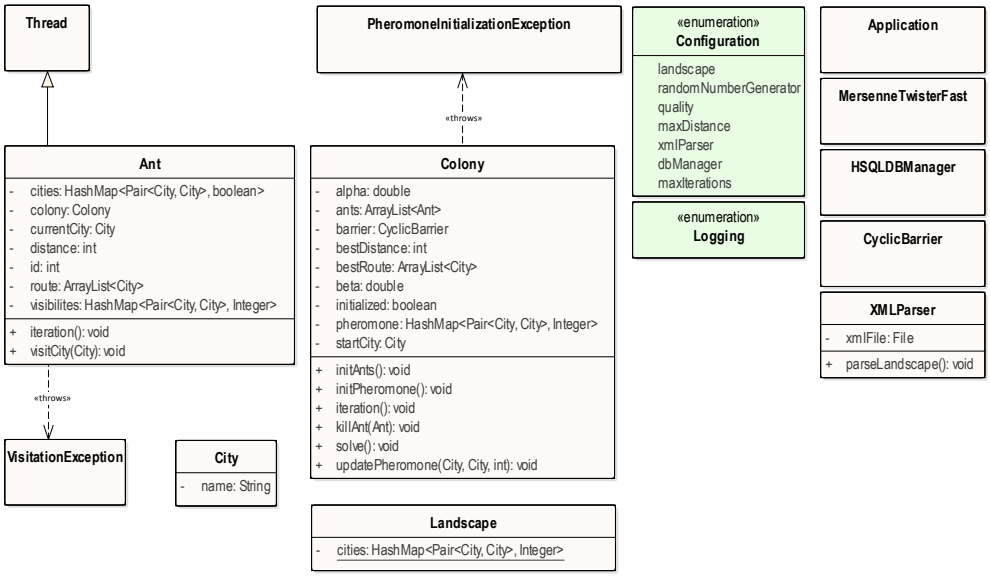

Abbildung 3.2: UML-Modellierung des Architekturentwurfs

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 moderne 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 ermöglicht in Zukunft auch andere Datei-Typen akzeptieren zu können. In der restlichen Implementierung ist eine Umsetzung nicht hilfreich oder zweckdienlich.

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 unterschiedliche 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 einem 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.

Aufgrund 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 hilfreich bzw. möglich. Dennoch sei auch hier erwähnt, dass eine Benutzung der Regel zu einfacher zu wartenden 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:

$$P(s_{ij}) = \frac{\tau_{ij}^{\alpha} * \eta_{ij}^{\beta}}{\sum_{x \in N} \tau_{ix}^{\alpha} * \eta_{ix}^{\beta}} \tag{3.13}$$

s = Strecke zwischen zwei Städten
i = Quellstadt
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

Einige der Parameter sind herleitbar, wie zum Beispiel i und j die die Städte abbilden. Ebenso is N lediglich die Menge der Städte, die eine Ameise in einer bestimmten Situation noch besuchen kann. Diese wird Menge wird über den Umstand definiert, dass bereits besuchte Städte "gesperrtßind.

Allerdings gibt es auch einige Werte, die genauer beleuchtet werden müssen. Hierbei ist von τ, η, α und β die Rede, welche den zentralen Bereich dieser Formel bilden. Alle vier Parameter zusammen sind ausschlaggebend dafür welche Stadt von der Ameise besucht wird. Im Folgenden werden diese einzeln erklärt und Ihre Funktion dargelegt.

3.5.1 Tau - τ

Tau steht in der dargestellten Formel für den Pheromonwert für die Strecke zwischen i und j. Dieser wird von der Gesamtheit der Ameisen einer Kolonie bestimmt. Die Berechnung des Pheromonwerts wird im Kapitel 3.7 noch behandelt. Von der Ameise wird also der Wert der Strecke bestimmt und mit Eta verrechnet.

3.5.2 Eta - η

Eta steht hierbei für einen Wert der nur über Konstanten bestimmt werden kann. Hierbei wird die Länge der Strecke mit einer Konstanten verrechnet, wodurch ein Wert entsteht, der für diese Strecke konstant bleibt. Es lässt sich somit sagen, dass Eta bei einer zu Beginn normalisierten Pheromonverteilung für die Ameise attraktiver erscheint.

3.5.3 Alpha - α und Beta - β

Die beiden konstanten Parameter Alpha und Beta beschreiben die prozentuale Wahrscheinlichkeit, ob eine Ameise der Pheromonspur folgt oder einen neuen Pfad erkundet. Da der Alpha-Wert in Beziehung mit dem Pheromonwert steht, kann durch das Fest-

setzen bestimmt werden ob dieser Teil der Multiplikation höher ausfällt oder geringer. Dabei werden die Werte immer so gewählt, dass $\alpha + \beta = 1$ gilt. Dies verhindert eine unnötige Berechnung großer Zahlung und stellt trotzdem die Funktionsfähigkeit sicher.

3.6 Sensitivitätsanalyse

Um den Einfluss von Alpha und Beta zu verdeutlichen, folgt ein Beispiel bezogen auf die Situation einer Ameisenkolonie. In Abbildung 3.3 zu sehen ist eine Ameisenkolonie, die neu aufgebaut wurde. Diese Kolonie hat Zugang zu zwei Nahrungsquellen: Einer Wasserquelle und einer Zuckerwasserquelle. Für die Ameisen deutlich wertvoller ist die Zuckerwasserquelle, allerdings ist diese auch weiter entfernt. Zu sehen sind auch bereits Ameisen, die auf die Nahrungssuche gehen. Hierbei ist zu beachten, dass die Ameisen im derzeitigen Zustand gleich verteilt sind. Je länger die Nahrungssuche abläuft, desto

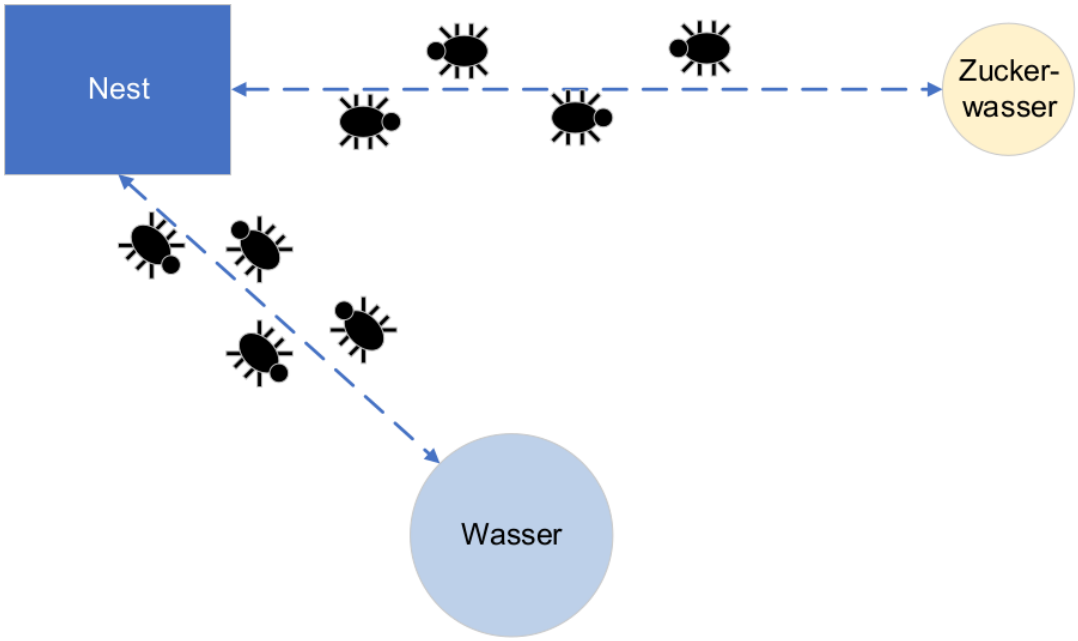

Abbildung 3.3: Modellierung einer Ameisenkolonie mit zwei unterschiedlichen Nahrungsquellen.

höher wird der Pheromonwert der Strecke, an deren Ende das Zuckerwasser zu finden ist. In der im vorherigen Kapitel vorgestellten Formel sind Alpha und Beta die beiden Parameter, über die bestimmt wird wie sehr der Pheromonwert gewichtet wird.

Die folgenden Beispiele basieren auf der Annahme, dass genügend Zeit vergangen ist,

damit die Ameisenkolonie die Strecken ausreichend mit Pheromonen belegen konnte. Zusätzlich zu beachten ist, dass Alpha und Beta laut Definition der vorliegenden Architektur gemeinsam 1 ergeben müssen.

3.6.1 α hoch - β niedrig

Wird Alpha maßgeblich größer gewählt als Beta, so wird der Pheromonwert der Strecke höher gewichtet als die Länge. Auf das Beispiel angewendet bedeutet das, dass die Ameisen - wie in Abbildung 3.4 - deutlich wahrscheinlicher das Zuckerwasser einsammeln. Diese Situation entspricht der Realität und ist die logischste. Denn der Pheromonwert soll ein Leitwert für die restliche Kolonie sein, welche Strecke wertvoll ist.

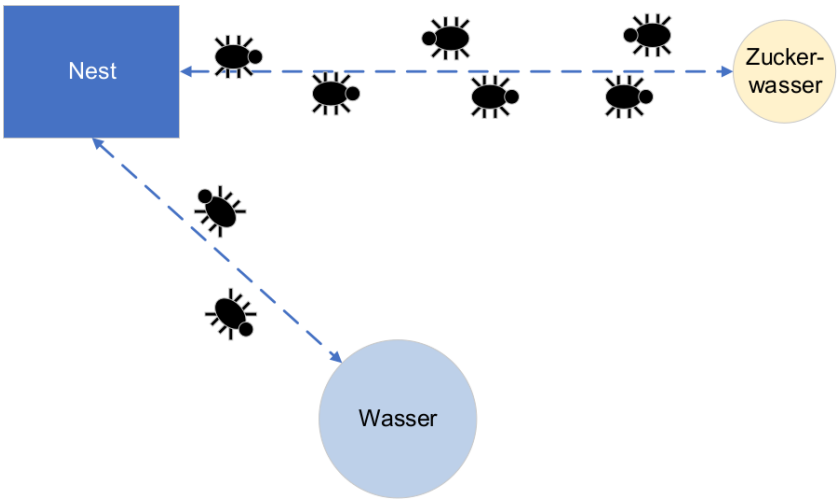

Abbildung 3.4: Modellierung der Parametereinstellung, sodass der Wert Alpha merklich größer als der von Beta ist.

3.6.2 α mittel - β mittel

Ein Fall, der ebenso möglich ist und in einem gewissen Rahmen Sinn ergibt, ist dass die beiden Parameter Alpha und Beta gleich groß gewählt werden. Hierbei würde sich das in Abbildung 3.5 ergeben. Im Gegensatz zu Abbildung 3.4 ist zu erkennen, dass mehr Ameisen den kürzeren Weg zur Wasserquelle wählen. Dies liegt daran, dass der Weg dorthin kürzer ist, was durch die neue Parameterverteilung höher gewichtet wird. Hier lässt sich die Aussage treffen, dass das Verhalten der neuen Ameisenkolonie aus Abbildung 3.3 - sowie auch der späteren implementierten Software - diesem Zustand

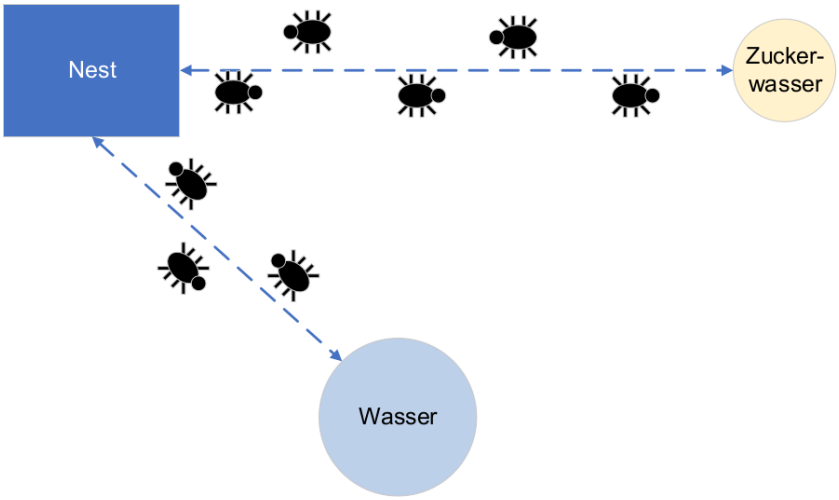

Abbildung 3.5: Modellierung der Parametereinstellung, sodass der Wert Alpha genauso groß ist wie der von Beta.

stark ähnelt. In beiden Fällen ist die Gewichtung ausgeglichen. Im Fall einer normalen Ameisenkolonie liegt dies aber an der noch nicht gewichteten Pheromonmatrix, die sich erst mit der Zeit aufbaut.

3.6.3 α niedrig - β hoch

Als letztes Beispiel - und auch letzten Abschnitt - folgt das Beispiel, dass die Länge der zu wählenden Strecke deutlich höher gewichtet ist als der Pheromonwert. Dies führt, wie in Abbildung 3.6 zu sehen ist, dazu, dass nur einzelne Ameisen den Weg zum wertvolleren Zuckerwasser wählen. Denn die Strecke ist hierbei länger und somit weniger attraktiv. Dies hat wieder für die Ameisenkolonie eine allgemeine Folge, die sich auch auf die Software übertragen würde. Denn hier würde nur mit minimaler Wahrscheinlichkeit eine Verbesserung eintreten. Zum Einen weil die Pheromongewichtung auf dem besseren Pfad nie höher aufgebaut werden kann.[6] Zum Anderen da eine Auswahl sowieso nur mit geringer Wahrscheinlichkeit zugunsten der Pheromonmatrix ausfallen würden.

[6]Hier sei noch erwähnt, dass alle Ameisen ihre Pheromone abgeben, sodass selbst die Ameisen auf dem Weg zum Wasser diesen Weg mit Pheromonen belegen. Dies hat den Effekt, dass durch die schiere Zahl an Ameisen dieser Weg einen höheren Pheromonwert besitzt.

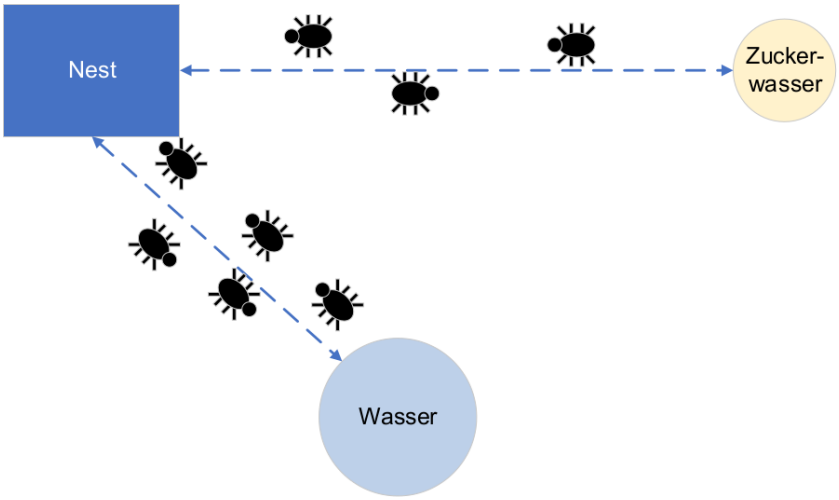

Abbildung 3.6: Modellierung der Parametereinstellung, sodass der Wert Alpha merklich kleiner als der von Beta ist.

3.7 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.

3.7.1 Ant - iteration()

In dem 3.5 wurde bereits die Berechnung beschrieben, die für jede neue Streckenauswahl von den Ameisen durchgeführt werden muss. Diese Berechnung wird in der Implementierung in der Iteration der Ameisen umgesetzt. Pro Durchgang bzw. Iteration wird also jede besuchbare, angrenzende Stadt betrachtet und auf Ihre Attraktivität untersucht. Überwiegt der Pheromonwert τ, welcher über Alpha gewichtet ist, über dem heuristischen Faktor η, der mit Beta verrechnet wird, so wird die bisher von der Kolonie gewählten Strecke gewählt. Diese Berechnung wird für alle Städte berechnet und dann verglichen. Die im Vergleich attraktivste Strecke wird in diesem Zusammenhang dann gewählt. Nachdem eine Strecke gewählt wird, verteilt die Ameise auf der Strecke ihre Pheromone indem der Kolonie ein Pheromonwert mitgeteilt wird, welcher auf den

bisherigen addiert werden muss.

3.7.2 Colony - killAnt()

Ebenfalls in dem in Abbildung 3.2 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.[7]

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.

3.7.3 Colony - updatePheromone()

Schon mehrfach erwähnt wurden die Pheromonwerte, die Pheromonmatrix, sowie die Pheromonverteilung. All diese Begriffe sind auf die Ameisenkolonie zurückzuführen. Denn diese enthält die Pheromonmatrix, die von den Ameisen zur Berechnung der Iteration benutzt wird. Um diese aktuell zu halten wird diese von jeder Ameise nach jeder Iteration upgedatet.

Dabei wird die Funktion *updatePheromone* der Kolonie aufgerufen und ein neuer Wert übergeben. Die Kolonie schreibt diesen dann in die zentrale Matrix, sodass der neue Wert bei der nächsten Iteration allen Ameisen ersichtlich ist. Durch diesen gleichzeitig und unübersichtlichen Zugriff auf die Matrix, muss diese Funktion synchronisiert ablaufen um einen Datenverlust zu verhindern.

[7] vgl. [**bibid**]

4 Implementierung

4.1 Klassendiagramm

4.2 Beschreibung der Implementierung

4.2.1 Persistenz

4.2.2 Applikation

4.2.3 Präsentation

4.3 ER-Diagramm

4.4 Komponenten-Diagramm

4.5 Paket-Diagramm

4.6 Perforamce-Analyse und -Optimierung

5 Fazit