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

13. Dezember 2017

Abstract

Zusammenfassung

Inhaltsverzeichnis

Abstract ii

Zusammenfassung iii

Abkürzungsverzeichnis vii

Abbildungsverzeichnis viii

Tabellenverzeichnis x

Formelverzeichnis xi

1 Einleitung 1

2 Stand der Technik & Forschung 2

2.1 Traveling Salesman Problem . . . 2

2.2 Ant Colony Optimization . . . 2

3 Konzeptionierung 3

3.1 Software Engineering . . . 3

3.1.1 Requirements Engineering . . . 4

3.1.2 COCOMO . . . 4

3.2 Architektur . . . 5

3.2.1 Persistenz . . . 6

3.2.2 Applikation . . . 6

3.2.3 TSP . . . 7

3.2.4 ACO . . . 7

3.2.5 Arbeitsweise der Architektur . . . 7

3.3 Umsetzung der SOLID-Prinzipien . . . 13

3.3.1 Single Responsibility . . . 13

3.3.2 Open / Closed . . . 14
3.3.3 Liskov Substitution . . . 14
3.3.4 Interface Segregation . . . 15
3.3.5 Dependency Inversion . . . 15
3.4 Parameterbeschreibung . . . 16
3.4.1 Tau - τ . . . 16
3.4.2 Eta - η . . . 17
3.4.3 Alpha - α und Beta - β . . . 17
3.5 Sensitivitätsanalyse . . . 17
3.5.1 α hoch - β niedrig . . . 18
3.5.2 α mittel - β mittel . . . 18
3.5.3 α niedrig - β hoch . . . 19
3.6 Ausgewählte Algorithmen . . . 20
3.6.1 Ant - iteration() . . . 20
3.6.2 Colony - killAnt() . . . 21
3.6.3 Colony - updatePheromone() . . . 22

4 Implementierung **23**
4.1 Beschreibung der Implementierung . . . 23
4.1.1 Klassendiagramm . . . 23
4.1.2 Persistenz . . . 24
4.1.3 Applikation . . . 25
4.1.4 TSP . . . 26
4.1.5 ACO . . . 27
4.2 Beschreibung der verwendeten Datenstrukturen . . . 29
4.2.1 BigDecimal . . . 29
4.2.2 ArrayList . . . 30
4.2.3 HashMap vs zweidimensionales Array . . . 31
4.3 Beschreibung der Testabdeckung . . . 32
4.4 Beweis der Funktionsfähigkeit . . . 32
4.4.1 numerisches Beispiel . . . 33
4.4.2 a280 drilling problem . . . 33
4.5 Performance-Analyse und -Optimierung . . . 34
4.5.1 CPU-Auslastung . . . 35
4.5.2 Heap-Nutzung . . . 35

4.5.3 Anzahl Threads pro Minute 36
4.5.4 Fazit zur Performance 37

5 Evaluation der Implementierung **39**
5.1 Betrachtung der Auswirkung der Parametergewichtung 39
5.2 Auswirkung von hoher Anzahl an parallelen Threads 39
5.3 Verhalten bei hoher Laufzeit 39

6 Proof Of Concept **40**

7 Fazit **41**

Literaturverzeichnis **42**

Anhang **43**

Abkürzungsverzeichnis

ACO Ant Colony Optimization oder Ameisenalgorithmus

TSP Traveling Salesman Problem

UML Unified Modeling Language

ER Entity Relationship

COCOMO Constructive Cost Model

Abbildungsverzeichnis

3.1 Darstellung des Traveling Salesman Problem (TSP)-Beispiels zur Darlegung der Arbeitsweise der Architektur 8
3.2 Modellierung einer Ameisenkolonie mit zwei unterschiedlichen Nahrungsquellen. 18
3.3 Modellierung der Parametereinstellung, sodass der Wert Alpha merklich größer als der von Beta ist. 19
3.4 Modellierung der Parametereinstellung, sodass der Wert Alpha genauso groß ist wie der von Beta. 20
3.5 Modellierung der Parametereinstellung, sodass der Wert Alpha merklich kleiner als der von Beta ist. 21

4.1 Klassendiagramm der vorliegenden Softwarelösung 24
4.2 Modellierung der Klassen des Persistenz-Bereichs über CRC-Karten . . . 25
4.3 Modellierung der Klassen des Applikation-Bereichs über CRC-Karten . . 26
4.4 Modellierung der Klassen des TSP-Bereichs über CRC-Karten 27
4.5 Modellierung der Klassen des Ant Colony Optimization oder Ameisenalgorithmus (ACO)-Bereichs über CRC-Karten 28
4.6 Ausschnitt der Log-Datei bei der Berechnung des numerischen Beispiels aus 3.2.5. Jede Log-Zeile steht hier für eine Generation an Ameisen. . . . 33
4.7 Ausschnitt der Log-Datei bei der Berechnung des “a280 drilling problem”. Auch hier steht jede Log-Zeile für eine Generation an Ameisen. 34
4.8 Aufzeichnung der CPU-Beanspruchung der Applikation über die gesamte Laufzeit 36
4.9 Aufzeichnung des Speicherverbrauchs der Applikation über die gesamte Laufzeit. Blau markiert ist der derzeit verwendete Speicher, in orange der allokierte Speicher. 37
4.10 Aufzeichnung der Menge an gleichzeitig aktiven Threads des Programms. In blau gezeichnet ist die Menge an Daemon-Threads, in orange die Anzahl an aktiven Live-Threads. 38

1 Paketdiagramm der vorliegenden Softwarelösung 43
2 ER-Diagramm der vorliegenden Softwarelösung 44
3 Testfälle der Klasse *Ant*, Seite 1 45
4 Testfälle der Klasse *Ant*, Seite 2 46
5 Testfälle der Klasse *Ant*, Seite 3 47
6 Testfälle der Klasse *Colony*, Seite 1 48
7 Testfälle der Klasse *Colony*, Seite 2 49
8 Testfälle der Klasse *Landscape*, Seite 1 50
9 Testfälle der Klasse *Landscape*, Seite 2 51
10 Testfälle der Klassen von *Parser* 52
11 Testfälle der Klasse *HSQLDBManager*, Seite 1 53
12 Testfälle der Klasse *HSQLDBManager*, Seite 2 54

Tabellenverzeichnis

3.1 Komplexitätseinschätzung des Projekts im Rahmen einer Abschätzung der Funktionspunkte 5
3.2 Distanzmatrix des numerischen TSP-Beispiels 8
3.3 Initiale Pheromonmatrix des TSP-Beispiels 9
3.4 Ergebnisse der drei Ameisen der ersten Generation 11
3.5 Pheromonmatrix des TSP-Beispiels nach dem ersten Durchlauf 11
3.6 Ergebnisse der drei Ameisen der zweiten Generation 12
3.7 Pheromonmatrix des TSP-Beispiels nach dem zweiten Durchlauf 12
3.8 Ergebnisse der drei Ameisen der dritten Generation 12
3.9 Pheromonmatrix des TSP-Beispiels nach dem finalen dritten Durchlauf . 13

4.1 Werte der Test-Coverage-Daten 32
4.2 Für Performance-Benchmarks verwendete relevante Hardware 35

Formelverzeichnis

3.1 COCOMO II-Formel zur Berechnung der Arbeitsaufwände bei der Entwicklung & Implementierung eines Software-Projekts 5
3.6 Formeln zum Berechnen des Verhältnisses λ von Pheromonwert und Streckenlänge 10
3.11 Formeln zum Berechnen der Wahrscheinlichkeiten ρ aus dem Verhältnis λ und der Summe aller λ 10
3.13 Formel zum Berechnen der Gewichtungen bei der Wegwahl 16

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 TSP 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 durch 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 referenziert 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.

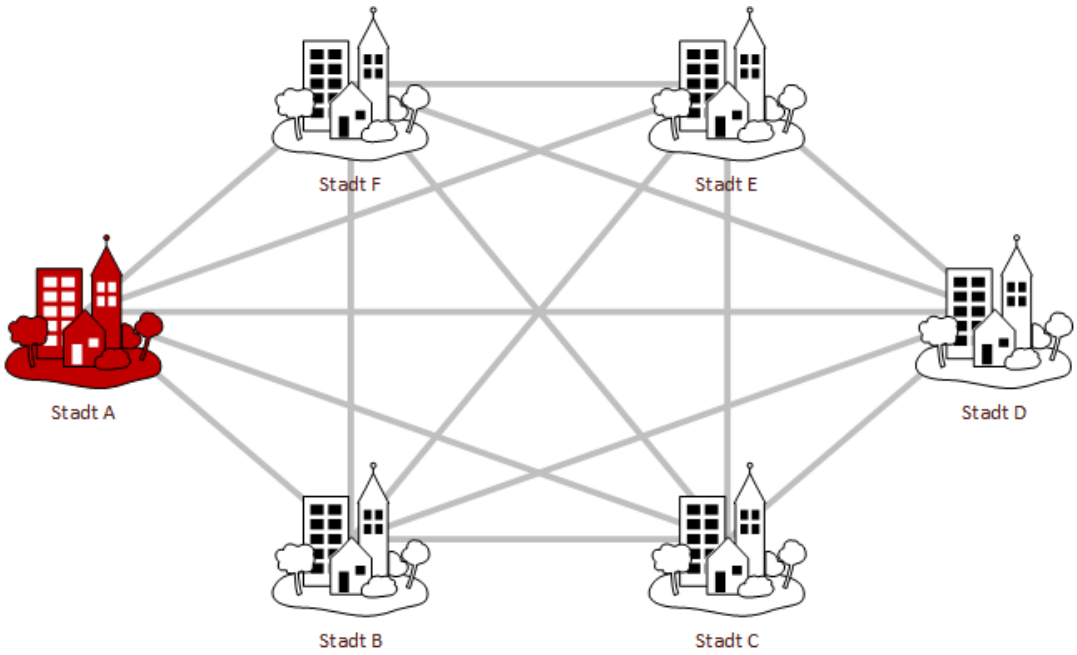

Abbildung 3.1: Darstellung des TSP-Beispiels zur Darlegung der Arbeitsweise der Architektur

	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 numerischen 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 3.4.

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

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

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

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 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.

3.3.1 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.

3.3.2 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.

3.3.3 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.

3.3.4 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.

3.3.5 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.

3.4 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.

3.4.1 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 3.6 noch behandelt. Von der Ameise wird also der Wert der Strecke bestimmt und mit Eta verrechnet.

3.4.2 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.

3.4.3 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 Festsetzen 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.

3.5 Sensitivitätsanalyse

Um den Einfluss von Alpha und Beta zu verdeutlichen, folgt ein Beispiel bezogen auf die Situation einer Ameisenkolonie. In Abbildung 3.2 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 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.

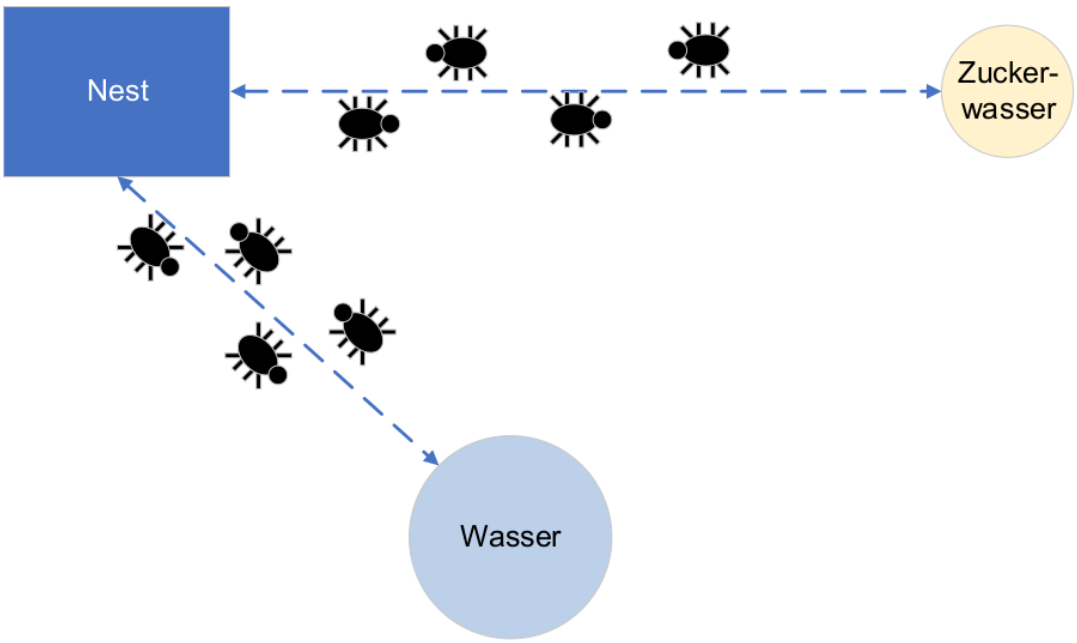

Abbildung 3.2: Modellierung einer Ameisenkolonie mit zwei unterschiedlichen Nahrungsquellen.

3.5.1 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 3.3 - 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.

3.5.2 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 3.4 ergeben. Im Gegensatz zu Abbildung 3.3 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 3.2 - sowie auch der späteren implementierten Software - diesem Zustand stark ähnelt. In beiden Fällen ist die Gewichtung ausgeglichen. Im Fall einer normalen

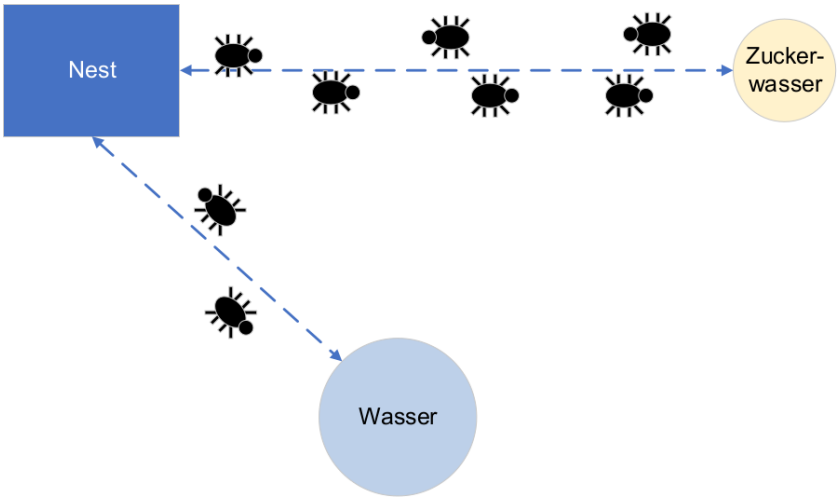

Abbildung 3.3: Modellierung der Parametereinstellung, sodass der Wert Alpha merklich größer als der von Beta ist.

Ameisenkolonie liegt dies aber an der noch nicht gewichteten Pheromonmatrix, die sich erst mit der Zeit aufbaut.

3.5.3 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 3.5 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.

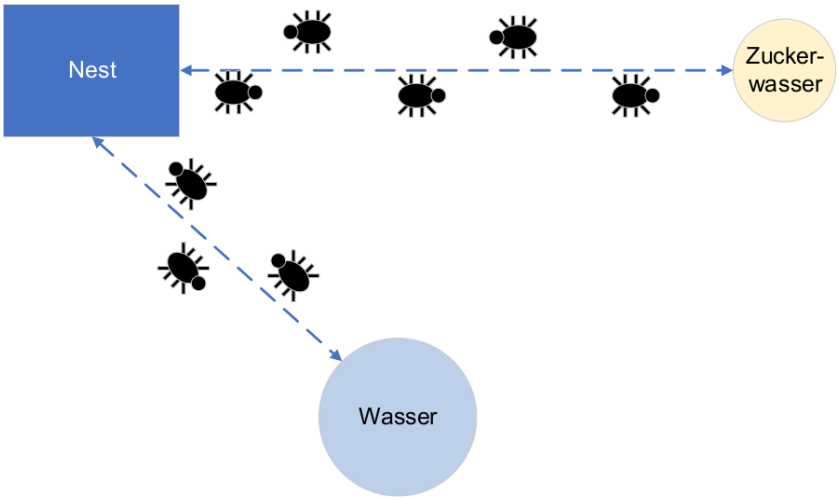

Abbildung 3.4: Modellierung der Parametereinstellung, sodass der Wert Alpha genauso groß ist wie der von Beta.

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()

In dem 3.4 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

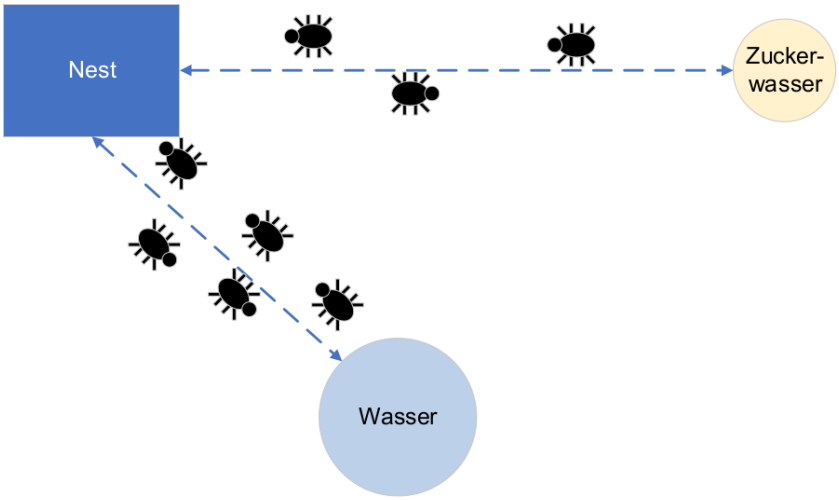

Abbildung 3.5: Modellierung der Parametereinstellung, sodass der Wert Alpha merklich kleiner als der von Beta ist.

bisherigen addiert werden muss.

3.6.2 Colony - killAnt()

Ebenfalls in dem in Abbildung **??** 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.

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

3.6.3 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.

4 Implementierung

Bei der Konzeptionierung wurden Punkte wie die Parametereinstellungen, die Ameisenkolonie als zentraler Punkt der Applikation, sowie die Ameise als einzelner Akteur beschrieben. Auch wurden Punkte wie die SOLID-Prinzipien aufgezeigt und die Umsetzung beschrieben. Auch wurde für ein Beispiel des Ablaufs der Applikation ein numerisches Beispiel aufgezeigt.

Im Folgenden wird auf die wirkliche Umsetzung der einzelnen Punkte eingegangen, sowie auch der Arbeitsablauf der letzendlichen Implementierung beschrieben. Als Beweis der Funktionsfähigkeit werden zwei Beispiele berechnet:

- eigenes numerische Beispiel aus Kapitel 3.2.5
- die Problemstellung des "a280 drilling problem"

4.1 Beschreibung der Implementierung

Um die Funktionsweise der Softwarelösung zu verstehen, ist eine klare Übersicht über die einzelnen Bestandteile notwendig. Diese wird im Folgenden über ein Klassendiagramm inkl. Beschreibung gegeben. Danach werden noch die einzelnen Abschnitte in ihrer Funktion beschrieben.

4.1.1 Klassendiagramm

Bevor auf den genauen Funktionsablauf der Applikation eingegangen wird, sollte zuerst ein Überblick auf die Implementierung gegeben werden. Hierzu ist in Abbildung 4.1 der komplette Umfang aller Klassen aufgezeigt [1]. Leicht erkennbar ist, dass der größte

[1] Aus Gründen der Übersichtlichkeit wurden die getter- und setter-Methoden weggelassen, um die Abbildung nicht über zu dimensionieren

Teil der Logik innerhalb des aco-Pakets - welches unter anderem Ant und Colony enthält - stattfindet. Landscape, also die Klasse welche das TSP beschreibt, dient nur als Schnittstelle zur Abfrage der Distanzen zwischen den Städten.

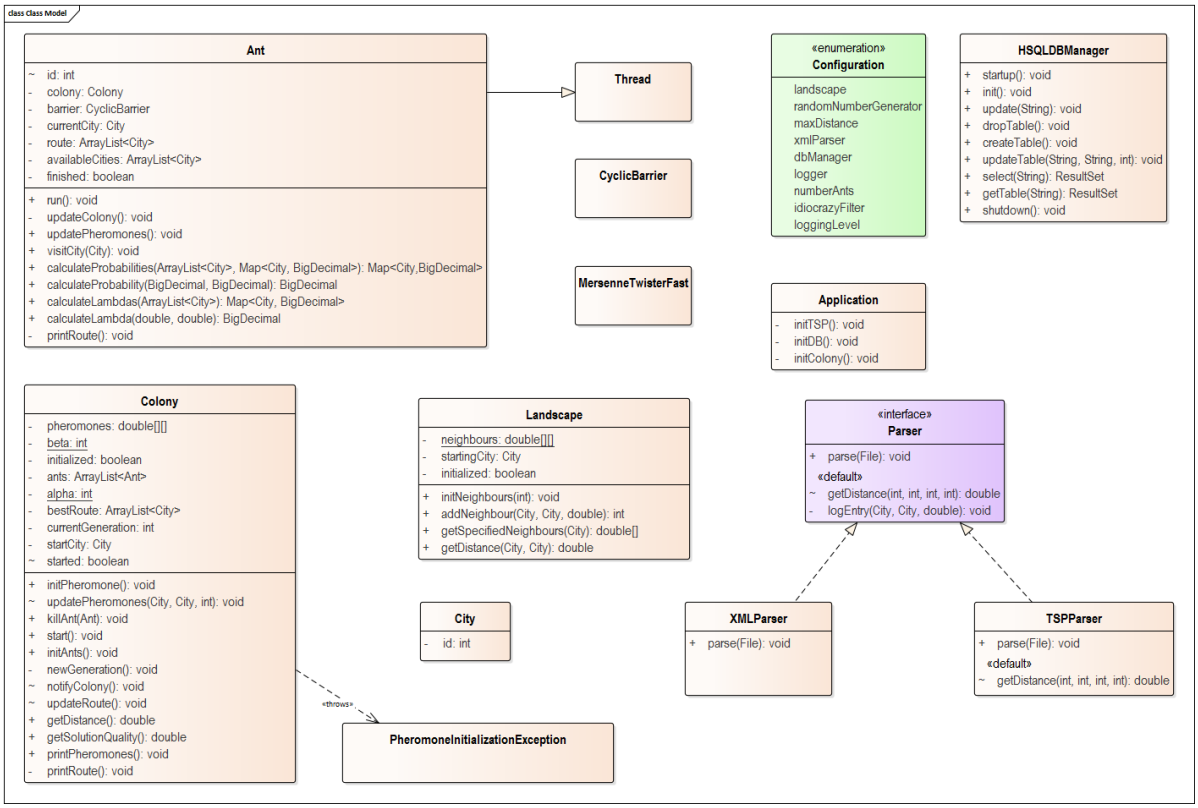

Abbildung 4.1: Klassendiagramm der vorliegenden Softwarelösung

4.1.2 Persistenz

Um die Ergebnisse der einzelnen Generationen persistent abspeichern zu können, ohne den Speicherverbrauch des Programms ansteigen zu lassen, wird eine Datenbank-Anbindung an eine HSQLDB-Datenbank verwendet. Um die Performance möglichst hoch und den Datenbankumfang möglichst klein zu halten, werden in dieser allerdings nur zwei Tabellen erstellt und verwaltet.

Zum Einen wird von jeder Generation, welche den Algorithmus durchlaufen hat, die beste Route erfasst und abgespeichert. Hierbei besteht ein Tabelleneintrag lediglich aus einer Zufalls-ID, der Route als String und der Distanz als Double.

Zum Anderen wird eine Tabelle zur Verfolgung der Verbesserung angelegt. In der

Klasse: HSQLDBManager	
Super-Klasse: -	
Verantwortlichkeiten	**Abhängigkeiten**
Datenbank verwalten Daten in Datenbank speichern	Configuration

Abbildung 4.2: Modellierung der Klassen des Persistenz-Bereichs über CRC-Karten

Historie wird eine Route nur dann abgespeichert, wenn diese sich im Vergleich zur vorherigen auch verbessert hat. Somit besteht diese aus den gleichen Attributen, wie die Generationen-Tabelle, mit dem Zusatz, dass auch die Generation, welche die Verbesserung verursacht hat, mit einer Ganzzahl abgespeichert wird.

4.1.3 Applikation

Dem Bereich der Applikation sind die Klassen zugeordnet, welche sich um die Initialisierung bzw. die Verwaltung von zentralen Parametern kümmern. Somit sind hier die Klassen Application, Configuration, sowie alle Implementierungen des Parser-Interfaces zu finden. In Abbildung 4.3 ist eine Modellierung dieser Klassen über CRC-Karten gezeigt.

Application ist beim Start des Programms dafür verantwortlich, dass alle wichtigen Instanzen korrekt initialisiert werden. Darunter fällt das Starten des Parsers, des HSQLDB-Managers, sowie der Ameisenkolonie. Sobald die Kolonie gestartet ist, ist die letzte Aufgabe der Application auf das Ende der Berechnung zu warten und dann die Datenbank wieder abbauen zu lassen.

Configuration ist als zentrale Anlaufstelle für alle Instanzen, die nur einmal benötigt werden, und alle Parameter, die mehrfach benötigt werden, gedacht. So verwaltet diese unter anderem die Gewichtung der Parameter bei der Berechnung der Wahrscheinlichkeiten. Auch wird über die Configuration der zentrale Logger bereitgestellt. Über diesen werden die Ergebnisse der Generationen erfasst bzw. auch Debug-Informationen gesammelt.

Alle Implementierungen des Parser-Interfaces haben einen gemeinsamen Zweck: Die Landscape-Klasse - welche auch von Configuration verwaltet wird - zu konfigurieren und die Problemstellung zu generieren. Hierzu gibt es zwei Varianten: Es kann entwe-

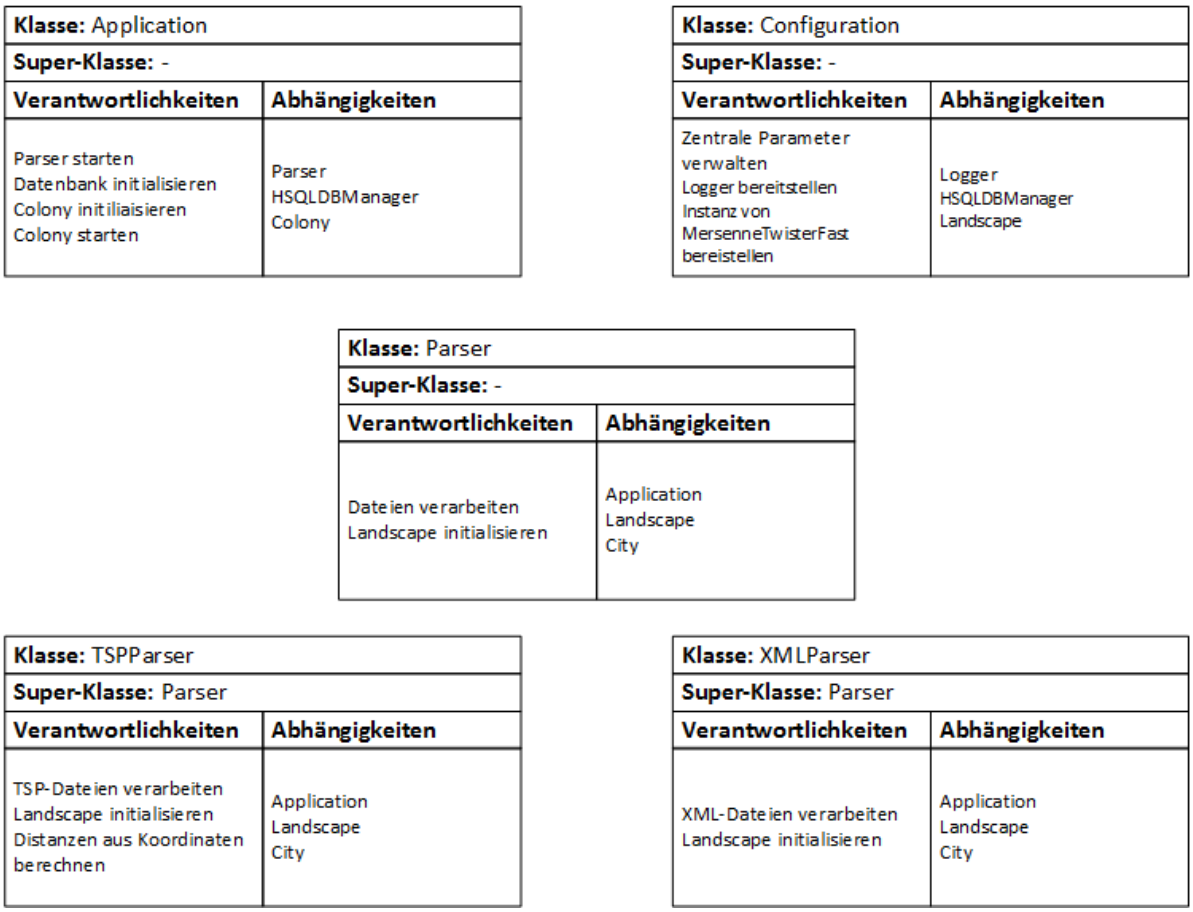

Abbildung 4.3: Modellierung der Klassen des Applikation-Bereichs über CRC-Karten

der eine XML-Datei erstellt werden, welche eine bestimmte Formatierung erfüllen muss. Oder aber es kann eine TSP-Datei beigefügt werden, welche dem Standard der TSP-Problemstellungen folgt. In beiden Fällen wird die Datei ausgelesen und die Problemstellung in die Applikation importiert, sodass die Ameisenkolonie auf dieser aufbauen kann.

4.1.4 TSP

Der TSP-Bereich der Architektur dient zu aller erst der Verdeutlichung des Aufbaus. So soll durch die Verwendung der City-Klasse statt einfachen Integern das Verständnis gefördert werden. Innerhalb der Landscape-Instanz - welche in der Konfiguration zentral initialisiert wird - wird die zweidimensionale Matrix der Distanzen zwischen den Städten bereit gehalten.

Über diese Matrix kann immer zentralisiert der Wert von den Ameisen nachgefragt werden, ohne eine unnötige Berechnung durchzuführen. Ebenso fördert dieser Aufbau die Sicherung der Funktionsfähigkeit, da einfacher sichergestellt werden kann, dass die Distanzen richtig berechnet werden. Ein weiterer Vorteil ist, dass die Ameisen hier eine Liste von Nachbarn der derzeitigen Position nachfragen können, indem sie ihren derzeitigen Standort übergeben. Dies vereinfacht das Konzept bei der Berechnung innerhalb der Ameisen-Klasse.

Klasse: Landscape	
Super-Klasse: -	
Verantwortlichkeiten	**Abhängigkeiten**
Distanzmatrix verwalten Nachbarn einer Stadt berechnen Distanzen liefern	City

Klasse: City	
Super-Klasse: -	
Verantwortlichkeiten	**Abhängigkeiten**
Verständnis für Struktur erhöhen	-

Abbildung 4.4: Modellierung der Klassen des TSP-Bereichs über CRC-Karten

4.1.5 ACO

Der hauptsächliche Teil der Berechnung wird innerhalb des ACO-Bereichs durchgeführt. Denn hier befindet sich die Ameisenkolonie, als Verwaltungsorgan, und die Ameisen, welche als Threads gleichzeitig auf die Suche nach der bestmöglichen Lösung gehen.

Von der Kolonie werden die Ameisen insofern verwaltet, dass die Threads über diese Klasse gestartet werden, sowie hier auch die Ergebnisse abgeliefert werden. Nach jeder Generation wird innerhalb der Kolonie für jede Ameise eine Auswertung gestartet, ob die Route besser war als die derzeit beste. Sollte dies der Fall sein, wird die alte Route mit der neuen Route überschrieben.

Unabhängig davon ob eine neue beste Route gefunden wurd oder nicht, wird nach jeder Generation von Ameisen ein Update auf die Pheromonmatrix durchgeführt. Hierbei meldet jede Ameise für jeden Weg, den sie zwischen zwei Städten gegangen ist, einen Wert, welcher auf den derzeitigen Pheromonwert addiert werden soll. Hierfür iteriert eine Ameise über die Route, welche sie sich gemerkt hat, und ruft die updatePheromone-Methode der Kolonie auf, mit dem Wert $1/distance$, wobei $distance$ der Distanz zwischen der Stadt, über welche gerade iteriert wird, und der Folgestadt beträgt.

Klasse: Colony	
Super-Klasse: -	
Verantwortlichkeiten	**Abhängigkeiten**
Pheromonmatrix verwalten Generationen (Ameisen) erzeugen & starten SolutionQuality berechnen & vergleichen	Configuration Landscape City Ant HSQLDBManager

Klasse: Ant	
Super-Klasse: -	
Verantwortlichkeiten	**Abhängigkeiten**
Routen berechnen Route an Colony melden	Configuration CyclicBarrier Colony City Landscape MersenneTwisterFast

Klasse: MersenneTwisterFast	
Super-Klasse: -	
Verantwortlichkeiten	**Abhängigkeiten**
Zufallszahlen berechnen	-

Abbildung 4.5: Modellierung der Klassen des ACO-Bereichs über CRC-Karten

Die restliche Berechnungsarbeit wird von den einzelnen Ameisen bzw. Threads bewältigt. Diese berechnen ab dem Startpunkt für alle möglichen Nachbarn [2] einen λ-Wert[3]. Als nächstes werden alle λs aufsummiert, um mit den einzelnen λs der Städte dividiert durch die Summe die Wahrscheinlichkeit zu berechnen. Hierdurch wird beschrieben, dass Städtepaare, welche eine kurze Distanz besitzen, sowie einen hohen Pheromonwert besitzen, deutlich wahrscheinlicher besucht werden. Hierbei kann eine unterschiedliche Gewichtung vorgenommen werden, wie in Kapitel 3.4 und Kapitel 3.5 beschrieben wurde.

Nachdem für alle erreichbaren Städte die Wahrscheinlichkeiten berechnet wurden, wird eine Zufallszahl bestimmt. Sollte eine Wahrscheinlichkeit für eine Stadt höher sein, als die Zufallszahl so wird die Berechnung an dieser Stelle beendet und die Ameise besucht diese Stadt. Sollte diese Bedingung für keine einzelne Stadt erfüllt werden, so werden die Wahrscheinlichkeiten solange aufsummiert bis die Summe größer ist als die Zufallszahl. Die Stadt, bei welcher diese Schwelle überschritten wird, wird dann von der Ameise aufgesucht.

Sobald jede Ameise ihre Route beendet hat und wieder bei der Anfangsstadt angekommen ist, wird über die CyclicBarrier zentral eine Methode innerhalb der Kolonie angestoßen. Diese fordert nach und nach jede Ameise auf, das Ergebnis zu melden, um

[2] Ein Nachbar ist dann erreichbar, wenn dieser noch nicht in der Route vorgekommen ist, also noch nicht besucht wurde.

[3] Beschrieben wurde diese Berechnung bereits in Kapitel 3.4

die Pheromonmatrix zu aktualisieren. Nachdem die Kolonie vollständig aktualisiert wurde, ist die derzeitige Generation beendet und es wird eine neue generiert und gestartet.

4.2 Beschreibung der verwendeten Datenstrukturen

Nachdem nun die logischen Schritte der Implementierung aufgezeigt wurden, folgt ein kurzer Ausblick auf die Auswahl der verwendeten Datenstrukturen. Hierbei wird auch eine Einschätzung auf die Effizienz dieser Datenstrukturen gegeben und die Entscheidung, welche Datenstruktur verwendet wird, begründet.

4.2.1 BigDecimal

Eine Besonderheit der vorliegenden Implementierung ist die Verwendung von BigDecimal statt Double als zentrale Zahlenwerte. Hierbei werden BigDecimal vor allem dann verwendet, wenn die Berchnung der Lambda- bzw. Wahrscheinlichkeitswerte durchgeführt werden muss. Hier hat sich schnell gezeigt, dass der Zahlenraum von Double[4] bzw. auch die Implementierung von Double selbst nicht genau genug ist, um mit extrem kleinen Werten arbeiten zu können. Diese extrem kleinen Werte werden in der Applikation dann erreicht, wenn ein recht hoher Gewichtungswert gewählt wird, welcher zu einer hohen Potenzierung führt. Da die Applikation aber unabhängig der Gewichtung immer zuverlässig und genau arbeiten soll und muss, ist dieser Zustand nicht haltbar.

Um dieses Problem zu lösen, wurden BigDecimal als Ausweg gewählt. BigDecimal-Zahlen werden nicht zwingend als einzelner Speicher allokiert. Dies hat den Vorteil, dass eine beliebig genaue Genauigkeit erzeugt werden kann, da die Nachkommastellen nicht begrenzt werden durch einen maximalen Wert, wie bei primitiven Datentypen. Der Nachteil dieser Vorgehensweise ist ein Overhead, den das System verwalten muss, welcher bei primitiven Datentypen nicht vorhanden ist. Allerdings ist dieser Overhead, falls Genauigkeit wichtig ist, immer einer ungenauen Arbeitsweise vorzuziehen.

[4] 4,94065645841246544E-324 bis 1,79769131486231570E+308 [2]

4.2.2 ArrayList

ArrayList ist eine gängige Datenstruktur in Java, wie auch in wie vielen anderen Programmiersprachen[5] Sie hat den Vorteil, dass bei Erstellung der Liste nicht klar sein muss, wie lang die Liste werden wird. Dadurch kann die Liste dynamische bearbeitet werden, indem immer Objekte hinzugefügt und entfernt werden können. Wie immer gibt es aber nicht nur Vorteile.

Ein großer Nachteil von ArrayList ist der unperformante Speichervorgang der Daten. Da nicht bekannt ist, wie lange die Liste wird, kann auch kein zusammenhängeneder Speicher allokiert werden. Dies bedeutet, dass die Daten in der Liste meist fragmentiert abgespeichert werden müssen. Dadurch wird die Performance eingeschränkt, da zwischen den Daten immer die Pointer-Referenz aufgelöst werden muss. Bei Arrays ist dies nicht der Fall. Denn hier würde ein einziger Block an Daten belegt werden, über den in einem Zug iteriert werden kann. Somit ist eine Nutzung von Arrays generell der einer ArrayList vorzuziehen, solange die Größe bekannt ist.

Ein weiterer Nachteil ist der Struktur einer ArrayList geschuldet. Wenn eine ArrayList erstellt wird, belegt diese ein Array in welchem die zu speichernden Objekte abgelegt werden. Sollte das Array während der Laufzeit zu klein werden, muss ein neues Array belegt werden und das alte kopiert werden. Dies verursacht wiederum einen Performance-Einbruch bei der eigentlichen Applikation.[6]

Im vorliegenden Fall wurden ArrayList vor allem benutzt, um die möglichen erreichbaren Nachbarn abzuspeichern. Man könnte über einen Zähler und die Länge der Distanzmatrix zwar einen Algorithmus implementieren, welcher die nötige Länge berechnet und damit ein Array erzeugt. Allerdings wäre dies zum Einen umständlich und zum Anderen unschön. Auch würde der Performance-Gewinn relativ klein ausfallen, da bei jedem Schritt die Berechnung durchgeführt werden muss. Generell würde die Wartbarkeit des Codes darunter leiden. Um diese Punkte zu vermeiden wurde ArrayList als standardisierte Datenstruktur verwendet, um trotz allem eine möglichst performante und speichereffiziente Arbeitsweise in allen Methoden zu gewährleisten.

[5] In anderen Sprachen heißt die Datenstruktur zwar anders, hat aber meist die selbe Arbeitsweise.
[6] vgl. [1]

4.2.3 HashMap vs zweidimensionales Array

Zu Beginn der Implementierung wurden die Möglichkeiten zur Darstellung der Distanzen und Pheromonwerte betrachtet. Da alle Städte untereinander vernetzt sind, gibt es für jeden Pfad jeweils einen Distanzwert und einen Pheromonwert. Somit ergibt sich immer eine zweidimensionale Matrix. Der logische Schritt wäre diese Matrix über ein zweidimensionales Array zu implementieren. Allerdings wäre auch eine Implementierung über eine HashMap möglich. Um die beiden Möglichkeiten auszuloten wurden mehrere Aspekte beachtet:

- Performance
- Übersichtlichkeit
- Umständlichkeit

Der Punkt der Performance ist relativ schnell abgehandelt. Denn ähnlich zur ArrayList arbeitet auch eine HashMap mit einem Array als "Backend". Somit kann eine HashMap nicht schneller als ein Array sein. Dennoch kann es Anwendungsfälle geben, in denen es sinnvoller ist eine HashMap zu verwenden.

Ein möglicher Anwendungsfall wäre zum Beispiel, wenn durch eine HashMap eine bessere Übersichtlichkeit gewährleistet wäre. Denn in einem Array können immer nur Integer-Werte als Referenzen gewählt werden, bei einer HashMap aber auch ein beliebiges Objekt. Dadurch ist im objektorientierten Umfeld ein besseres Verständnis gewährleistet. Dies ist auch der Grund, warum zu Beginn die HashMap ausgewählt wurde, um Distanz- und Pheromon-Werte abzuspeichern. Da diese Werte immer in Beziehung zwischen zwei Städten stehen, wäre es auch sinnvoll diese beiden Städte als Referenz zu benutzen. Dies wäre bei einem Array nicht ohne weiteres möglich.

Der letzte Punkt der beachtet wurde war die Umständlichkeit. Auch wenn eine HashMap deutlich sinnvoller und übersichtlicher erscheint gibt es wieder einen Nachteil. Man kann nicht ohne weiteres der Reihe nach über alle Einträge iterieren. Auch kann nicht ohne weiteres ein Ausschnitt aus einem Array übergeben werden. Bei einem Array kann eine einzelne Spalte zurückgegeben werden, was zum Beispiel bei der Abfrage der Nachbarn einer Stadt sinnvoll ist. Im Falle einer HashMap müssten alle Einträge durchlaufen werden und geprüft werden, ob die derzeitige Stadt in dem Key-Paar als erste Stadt eingetragen ist. Dies führt zum Einen zu schlechterer Code-Qualität und zum Anderen zu Performance-Verlusten, da die ganze HashMap durchlaufen werden muss.

Vor allem wegen der Umständlichkeit wurde relativ schnell von einer Struktur mittels HashMaps auf einen, auf zweidimensionalen Arrays basierenden, Aufbau gewechselt.

4.3 Beschreibung der Testabdeckung

Bevor die Applikation im Gesamten getestet wird, sollte immer ein Komponenttest durchgeführt werden. Hierbei sollen die einzelnen Methoden und Algorithmen der Software unabhängig von einander einzeln geprüft werden. In dieser Arbeit wurde der Ansatz des TDD verfolgt, wodurch ein Code-Coverage-Wert von 100 Prozent angestrebt wird. Zum Zeitpunkt dieses Kapitels ergaben sich folgende Coverage-Daten:

	Klassenabdeckung	Methodenabdeckung	Zeilenabdeckung
aco	100%	92%	82%
tsp	100%	100%	100%
parser	100%	80%	83%
util.HSQLDBManager	100%	100%	94%

Tabelle 4.1: Werte der Test-Coverage-Daten

In den Coverage-Daten nicht abgebildet ist das Paket *main*[7], da innerhalb von *Application* nur Methoden anderer Klassen aufgerufen werden, welche bereits getestet wurde. Ebenfalls nicht getestet wurde das Paket *util*[8] mit Ausnahme des $HSQLDBManager$, welcher im Gegensatz zu $MersenneTwisterFast$ selbst implementiert wurde.

4.4 Beweis der Funktionsfähigkeit

In der Theorie funktionierte dieses Konstrukt bereits in Kapitel 3.2. Im Folgenden soll gezeigt werden, dass das entworfene Konstrukt aber auch in der Praxis relevante Werte liefert. Hierzu wird das in Kapitel 3.2.5 herangezogen, sowie ein standardisiertes Beispiel aus dem Themenbereich des TSP, nämlich das a280-Problem - also eine Problemstellung mit 280 Städten.

[7] s. Abbildung 1
[8] s. Abbildung 1

4.4.1 numerisches Beispiel

In Abbildung 3.1 in Kapitel 3.2.5 wurde der Aufbau des numerischen Beispiels schon gezeigt. An der Ausgabe, welche beispielhaft in Abbildung 4.6 als Ausschnitt der Log-Datei gezeigt ist, ist gut zu erkennen, dass schnell eine optimale Strecke gefunden werden konnte.

Als Erinnerung: Bei der manuellen Berechnung des numerischen Beispiels belief sich die optimale Route auf eine Distanz von 29. Diese trat bei drei Ameisen nach drei Generationen auf. Zu Testzwecken lief das Programm nur mit einer Ameise, da bei einer zu hohen Anzahl an Ameisen bereits die erste Generation die optimale Strecke fand. Es lässt sich hiermit ableiten, dass die implementierte Architektur genauso abläuft wie das theoretische Konzept, wodurch eine fehlerfreie Implementierung bewiesen ist.

```
1  Dez 06, 2017 12:40:50 AM main.Configuration log
2  INFORMATION:  Distance: 33.0 Best Route: 0,1,4,3,2,5,0
3  Dez 06, 2017 12:40:50 AM main.Configuration log
4  INFORMATION:  Distance: 33.0 Best Route: 0,1,4,3,2,5,0
5  Dez 06, 2017 12:40:50 AM main.Configuration log
6  INFORMATION:  Distance: 33.0 Best Route: 0,1,4,3,2,5,0
7  Dez 06, 2017 12:40:50 AM main.Configuration log
8  INFORMATION:  Distance: 33.0 Best Route: 0,1,4,3,2,5,0
9  Dez 06, 2017 12:40:50 AM main.Configuration log
10 INFORMATION:  Distance: 33.0 Best Route: 0,1,4,3,2,5,0
11 Dez 06, 2017 12:40:50 AM main.Configuration log
12 INFORMATION:  Distance: 29.0 Best Route: 0,1,2,3,4,5,0
13 Dez 06, 2017 12:40:50 AM main.Configuration log
14 INFORMATION:  Distance: 29.0 Best Route: 0,1,2,3,4,5,0
15 Dez 06, 2017 12:40:50 AM main.Configuration log
16 INFORMATION:  Distance: 29.0 Best Route: 0,1,2,3,4,5,0
17 Dez 06, 2017 12:40:50 AM main.Configuration log
18 INFORMATION:  Distance: 29.0 Best Route: 0,1,2,3,4,5,0
```

Abbildung 4.6: Ausschnitt der Log-Datei bei der Berechnung des numerischen Beispiels aus 3.2.5. Jede Log-Zeile steht hier für eine Generation an Ameisen.

4.4.2 a280 drilling problem

Aber nicht nur die fehlerfreie Implementierung soll bewiesen werden, sondern auch die Anwendungsmöglichkeiten auf komplexe Problemstellungen. Ohne diese Möglichkeit wäre das Programm darauf beschränkt die optimale Strecke innerhalb des numerischen Beispiels zu finden. Durch Abbildung 4.7 lässt sich aber beweisen, dass auch deutlich größere Probleme sich berechnen lassen. Erkennbar sind mehrere Punkte:

- Die Berechnungszeit wird etwas höher, erkennbar an dem Log-Zeitraum, der nun drei Sekunden umfasst

- Die Berechnungszeit beträgt immer noch deutlich weniger als eine Sekunde
- Es findet eine stetige Verbesserung der Distanz statt

Zusätzliche zu den Punkten lässt sich noch eine weitere Aussage ableiten: Das Problem lässt sich berechnen. Hierdurch ist der Punkt, der eigentlich bewiesen werden sollte, nachweislich belegt. Noch nicht bewiesen ist, ob der Algorithmus auch in kurzer Zeit ein Optimum finden kann. Da dieser Beweis allerdings mehr Hintergrundwissen und mehr Nachweise erfordert, wird dieser im späteren Kapitel 6 behandelt.

```
100 INFORMATION:  Distance: 5243.4454025453915 Best Route: 0,279,1,242,241,
101 Dez 06, 2017 12:29:39 PM main.Configuration log
102 INFORMATION:  Distance: 5237.278462371953 Best Route: 0,279,1,228,229,2
103 Dez 06, 2017 12:29:39 PM main.Configuration log
104 INFORMATION:  Distance: 5237.278462371953 Best Route: 0,279,1,228,229,2
105 Dez 06, 2017 12:29:40 PM main.Configuration log
106 INFORMATION:  Distance: 5237.278462371953 Best Route: 0,279,1,228,229,2
107 Dez 06, 2017 12:29:40 PM main.Configuration log
108 INFORMATION:  Distance: 5213.666372966497 Best Route: 0,279,1,255,256,2
109 Dez 06, 2017 12:29:40 PM main.Configuration log
110 INFORMATION:  Distance: 5188.778763253458 Best Route: 0,279,1,241,240,2
111 Dez 06, 2017 12:29:40 PM main.Configuration log
112 INFORMATION:  Distance: 5188.778763253458 Best Route: 0,279,1,241,240,2
113 Dez 06, 2017 12:29:40 PM main.Configuration log
114 INFORMATION:  Distance: 5188.778763253458 Best Route: 0,279,1,241,240,2
115 Dez 06, 2017 12:29:40 PM main.Configuration log
116 INFORMATION:  Distance: 5188.778763253458 Best Route: 0,279,1,241,240,2
117 Dez 06, 2017 12:29:40 PM main.Configuration log
118 INFORMATION:  Distance: 4749.54806217864 Best Route: 0,279,1,2,278,277,
119 Dez 06, 2017 12:29:40 PM main.Configuration log
120 INFORMATION:  Distance: 4749.54806217864 Best Route: 0,279,1,2,278,277,
121 Dez 06, 2017 12:29:40 PM main.Configuration log
122 INFORMATION:  Distance: 4749.54806217864 Best Route: 0,279,1,2,278,277,
123 Dez 06, 2017 12:29:41 PM main.Configuration log
124 INFORMATION:  Distance: 4749.54806217864 Best Route: 0,279,1,2,278,277,
```

Abbildung 4.7: Ausschnitt der Log-Datei bei der Berechnung des “a280 drilling problem”. Auch hier steht jede Log-Zeile für eine Generation an Ameisen.

4.5 Performance-Analyse und -Optimierung

Selbstverständlich ist für eine richtige Implementierung nicht nur wichtig, dass diese funktioniert. Sie muss dies auch möglichst performant und effizient tun. Hierzu werden mehrere Testläufe in Hinblick auf die Performance durchgeführt. Im Folgenden werden diese Testfälle beschrieben, sowie auch die Hardware, auf welcher die Tests durchgeführt werden. Ohne die Referenz zur verwendeten Hardware wäre eine Aussage über die Performance nur sehr bedingt verwendbar. In Abbildung 4.2 sind die beiden relevanten Komponenten aufgelistet. Zu beachten ist hierbei, dass AMD-Prozessoren allgemein eine höhere Multithreading-Performance besitzen, welche in der vorliegenden Software genutzt wird.

Komponente	Name	Technische Daten
CPU	AMD FX8350	8 Kerne, 4.2 GHz
RAM	Kingston 99U5471	24 GB DDR3, 666 MHz

Tabelle 4.2: Für Performance-Benchmarks verwendete relevante Hardware

Als Grundlage der Benchmarks wird das a280-TSP verwendet um zum Einen eine standardisierte Grundlage zu erhalten und zum Anderen um eine genügend große Problemstellung zu erzeugen, die bezogen auf den Rechenaufwand auch relevant ist. Es wurden 8 parallele Threads gestartet, um eine hohe Auslastung der Hardware zu gewährleisten, ohne durch zu umfangreiches Threadhandling Performance einzubüßen. Die Applikation wurde genau 60 Sekunden lang betrieben, um einen einfachen Vergleichswert zu erhalten.

4.5.1 CPU-Auslastung

Wie in Abbildung 4.8 zu sehen ist, beansprucht die Applikation durchgehend zwischen 60 und 70 Prozent. Dass keine 100 Prozent genutzt werden können, liegt daran, dass die Test-Umgebung nicht optimal ist. So wurden die Benchmarks auf einem Desktop-PC in normalen Betrieb mit paralleler Heim-Nutzung durchgeführt. Da für das Betriebssystem keine Priorisierung der Applikation vorliegt, wird diese mit anderen Programmen gleichgestellt und die Ressourcen auch dementsprechend verteilt.

Würde die Applikation auf einer einwandfreien Test-Umgebung betrieben werden, würden bis 100 Prozent der CPU genutzt werden, was zu einer Verbesserung der Performance führen würde. Aber eine hohe Nutzung der CPU alleine kann nicht beschreiben, wie effizient ein Programm arbeitet. Hierzu sind noch andere Parameter notwendig, wie zum Beispiel die Nutzung des Hauptspeichers.

4.5.2 Heap-Nutzung

In Abbildung 4.9 ist eine Statistik des benötigten Heaps der Applikation über die Laufzeit zu sehen. In blau markiert ist der benötigte Speicher im Heap zu einem bestimmten Zeitpunkt. In orange hinterlegt ist die Menge an Speicher, welche vom Programm allokiert wurde. Es lassen sich hierbei mehrere Aussagen ableiten.

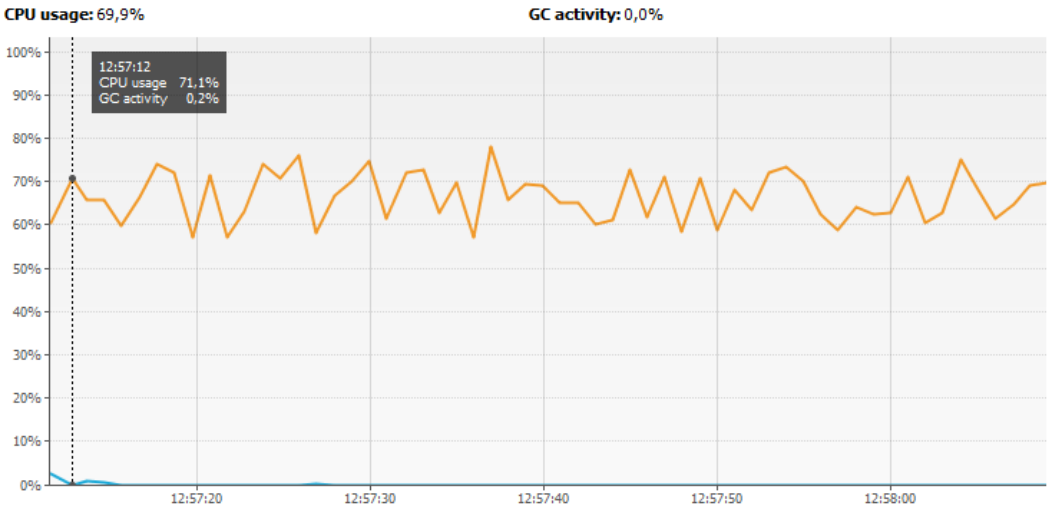

Abbildung 4.8: Aufzeichnung der CPU-Beanspruchung der Applikation über die gesamte Laufzeit

Zum Einen ist die Speicherallokierung konstant. Es werden vom Programm immer 2,4 GB an RAM belegt. Zum Anderen ist der Unterschied zwischen belegtem und allokierten Speicher teilweise enorm, was aber nicht auf die Arbeitsweise des Programms zurückzuführen ist. Zusätzlich lässt sich über die Grafik auch noch bestätigen, dass der GarbageCollector von Java zuverlässig nicht mehr benötigte Strukturen im Heap wieder frei gibt, was durch die lokalen Tiefpunkte im Heap-Verbrauch ersichtlich wird.

Als möglichen Optimierungspunkt lässt sich hier der überdimensionierte Heap nennen. Mit 2,4 GB ist der Verbrauch zwar nicht so hoch, dass er eine Verwendung auf einer heutigen Hardware verhindert. Allerdings ist bei einem Anspruch einer möglichst effizienten Software dieser unnötige Verbrauch nicht haltbar.

4.5.3 Anzahl Threads pro Minute

Schlussendlich ist nicht die Auslastung der Hardware ausschlaggebend für die Performance eines Algorithmus, sondern wie effizient dieser rechnet. Als Richtwert kann in diesem Beispiel die Anzahl an Threads pro Minute herangezogen werden, da ein Thread einem Rechenschritt entspricht bzw. über die Anzahl an Threads auch die Anzahl an Generationen errechnet werden können. Die Anzahl an Generationen ist hierbei ein direkter Richtwert wie viele Routen abgelaufen werden konnten.

In Abbildung 4.10 ist eine Statistik der in der Applikation aktiven Threads aufgezeigt.

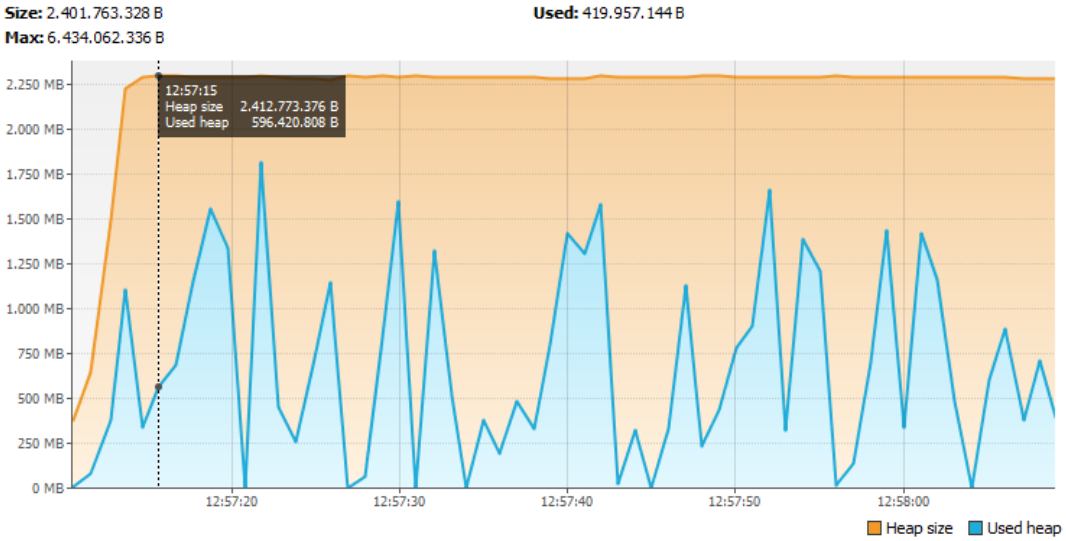

Abbildung 4.9: Aufzeichnung des Speicherverbrauchs der Applikation über die gesamte Laufzeit. Blau markiert ist der derzeit verwendete Speicher, in orange der allokierte Speicher.

In blau werden die Daemon-Threads - hauptsächlich Systemthreads, wie beispielsweise der GarbageCollector - und in rot die Live-Threads gezeigt. Die Menge an aktiven Live-Threads beläuft sich im Durchschnitt auf 19. Für die Berechnung werden pro Generation acht Threads gestartet, welche nach der Generation auch wieder abgebaut werden. Somit ist auch hier wieder ein Overhead vom System vorhanden - in Summe 11 Live-Threads und 10 Daemon-Threads.

Mithilfe der Datenerfassung lässt sich auch die genaue Anzahl an gestarteten Threads innerhalb der Applikation ermitteln. Innerhalb von 60 Sekunden wurden 2170 Threads bzw. Ameisen gestartet. Somit wurden ca 36,1 Ameisen pro Sekunde generiert. Teilt man diese Menge durch die Anzahl an Ameisen pro Generation (also acht) erhält man die Menge an Generationen pro Sekunde - in diesem Fall ca. 4,5.

4.5.4 Fazit zur Performance

Nun wurden im Detail die relevanten Daten zur Performance genannt und analysiert. Als Fazit können einige Punkte genannt werden:

- Die CPU wird fast vollständig genutzt
- Der Heap-Verbrauch ist akzeptabel, aber nicht optimal

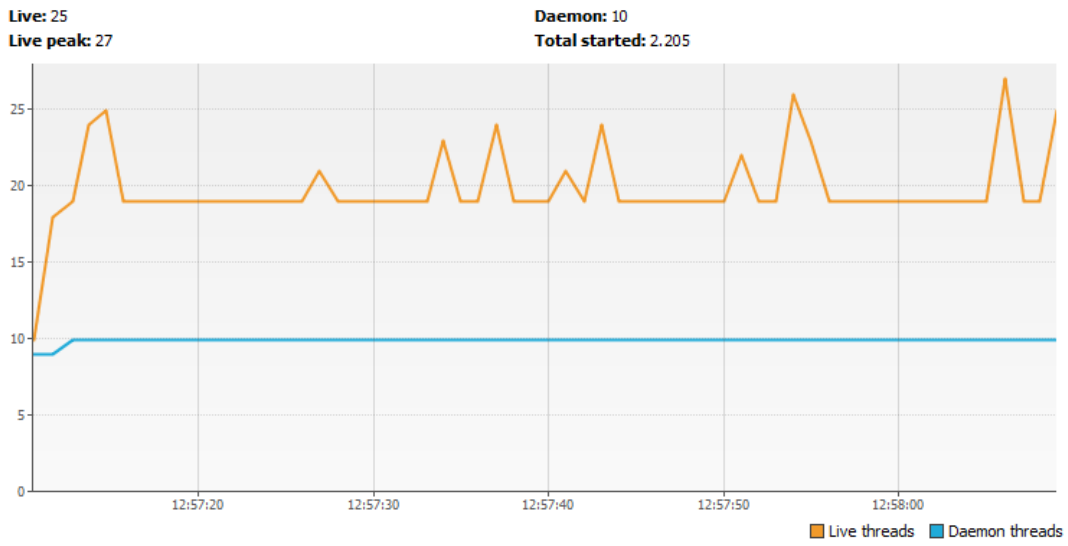

Abbildung 4.10: Aufzeichnung der Menge an gleichzeitig aktiven Threads des Programms. In blau gezeichnet ist die Menge an Daemon-Threads, in orange die Anzahl an aktiven Live-Threads.

- Es werden bei acht gleichzeitigen Ameisen 4,5 Generationen des a280-Problem berechnet

Diese Aussagen ergeben eine relativ gute Bewertung der Applikation in Hinblick auf die Performance. Allerdings wurden bereits Schwachstellen erkannt. So ist die Belegung des Speichers im Vergleich zum wirklichen Verbrauch teilweise sehr hoch. Auch entsteht durch System-Threads ein recht großer Overhead. Im weiteren Entwicklungsverlauf dieser Applikation sollte in diesen Punkten noch eine Optimierung vorgenommen werden.

5 Evaluation der Implementierung

5.1 Betrachtung der Auswirkung der Parametergewichtung

5.2 Auswirkung von hoher Anzahl an parallelen Threads

5.3 Verhalten bei hoher Laufzeit

6 Proof Of Concept

7 Fazit

Literaturverzeichnis

[1] Tamara Smyth. *CMPT 126: Lecture 11 Collections: Abstract Data Types and Dynamic Representation*. Techn. Ber. Simon Fraser University, 30. Okt. 2007.

[2] Christian Ullenboom. *Java ist auch eine Insel. Das umfassende Handbuch*. 10. Aufl. Galileo Press, 2011.

Anhang

Abbildung 1: Paketdiagramm der vorliegenden Softwarelösung

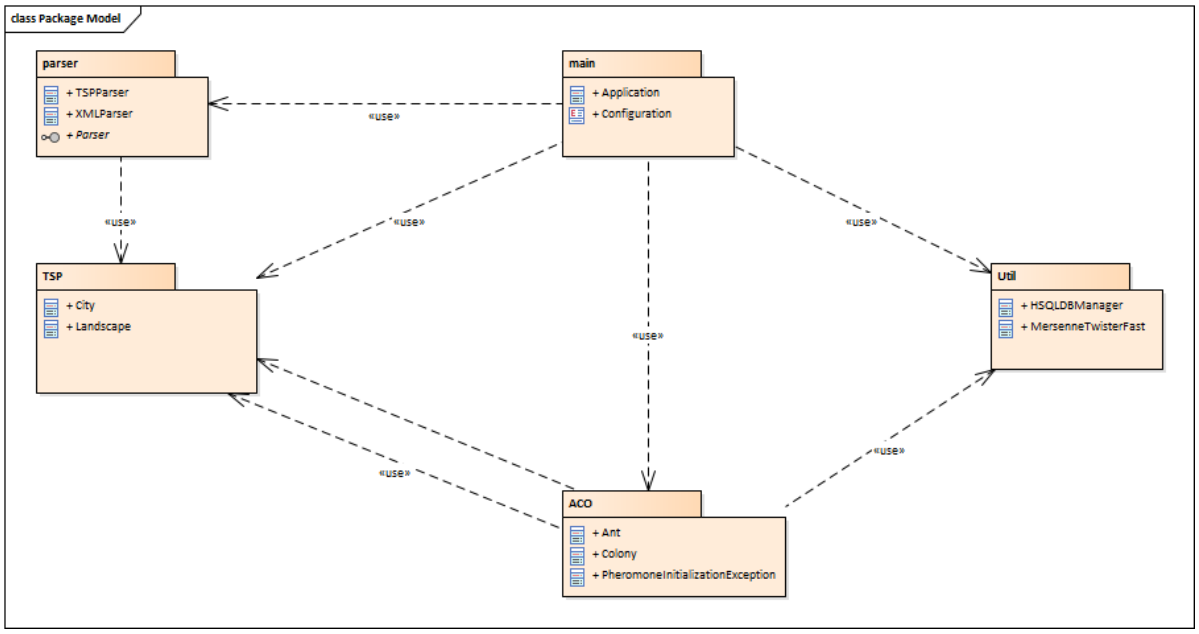

Abbildung 2: ER-Diagramm der vorliegenden Softwarelösung

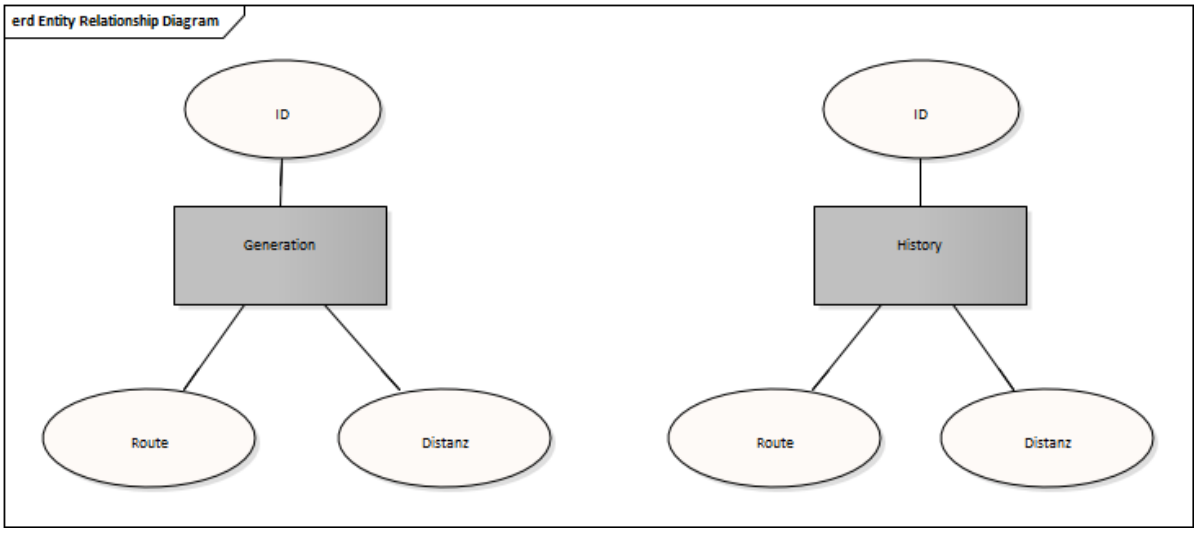

Abbildung 3: Testfälle der Klasse *Ant*, Seite 1

ID	1
Verfahren	Parametrisierter JUnit-Test
Klasse	Ant
Methoden	calculateLambda(double distance, double pheromone): BigDecimal Berechnung des Lambda-Wertes einer Strecke zwischen zwei Städten
Vorbedingung	-
Eingaben	3, 1
Erwartetess Ergebnis	$\frac{1}{3}$
Ergebnis	$\frac{1}{3}$ **Test erfolgreich**

ID	2
Verfahren	Parametrisierter JUnit-Test
Klasse	Ant
Methoden	calculateLambdas(ArrayList<City> cities): Map<City, BigDecimal> Berechnung der Menge an Lambda-Werten aller Nachbarn
Vorbedingung	City(1), City(2), City(3) (Stadt 2 und 3 haben 3 als Distanz zu Stadt 1)
Eingaben	ArrayList<City>(City(2), City(3))
Erwartetes Ergebnis	Map(Stadt(2), $\frac{1}{3}$; Stadt(3), $\frac{1}{3}$)
Ergebnis	Map(Stadt(2), $\frac{1}{3}$; Stadt(3), $\frac{1}{3}$) **Test erfolgreich**

ID	3
Verfahren	Parametrisierter JUnit-Test
Klasse	Ant
Methoden	calculateProbability(BigDecimal lambda, BigDecimal sum): BigDecimal Berechnung der Wahrscheinlichkeit, ob eine bestimmte Stadt besucht wird
Vorbedingung	-
Eingaben	$\frac{1}{3}$, 3
Erwartetes Ergebnis	$\frac{1}{9}$
Ergebnis	$\frac{1}{9}$ **Test erfolgreich**

ID	4
Verfahren	Parametrisierter JUnit-Test
Klasse	Ant
Methoden	calculateProbabilities(ArrayList<City> cities): Map<City, BigDecimal> Berechnung der Wahrscheinlichkeiten aller erreichbaren Nachbarn
Vorbedingung	City(1), City(2), City(3) (Stadt 2 und 3 haben 3 als Distanz zu Stadt 1)
Eingaben	ArrayList<City>(City(2), City(3))
Erwartetes Ergebnis	Map(Stadt(2), 0.5; Stadt(3), 0.5)
Ergebnis	Map(Stadt(2), 0.5; Stadt(3), 0.5) **Test erfolgreich**

Abbildung 4: Testfälle der Klasse *Ant*, Seite 2

ID	5
Verfahren	Parametrisierter JUnit-Test
Klasse	Ant
Methoden	run() Hauptalgorithmus einer Ameise: Berechnen der möglichen Wegstrecke
Vorbedingung	City(1), City(2), City(3) (Städte haben untereinander die Distanz 2) (Zwischen allen Städten liegt ein Pheromonwert von jeweils 1.5)
Eingaben	-
Erwartetes Ergebnis	Route der Ameise > 0 Aktuelle Stadt der Ameise == City(1)
Ergebnis	Route der Ameise == 4 Aktuelle Stadt der Ameise == City(1) **Test erfolgreich**

ID	6
Verfahren	Parametrisierter JUnit-Test
Klasse	Ant
Methoden	updatePheromones() Aktualisieren der Pheromonmatrix der Kolonie
Vorbedingung	City(1), City(2), City(3) (Städte haben untereinander die Distanz 3) Ameise hat eine Route von City(1), City(2), City(3), City(1)
Eingaben	-
Erwartetes Ergebnis	$\{\{1, 1+\frac{1}{3}, 1\}, \{1, 1, 1+\frac{1}{3}\}, \{1+\frac{1}{3}, 1, 1\}\}$
Ergebnis	$\{\{1, 1+\frac{1}{3}, 1\}, \{1, 1, 1+\frac{1}{3}\}, \{1+\frac{1}{3}, 1, 1\}\}$ **Test erfolgreich**

ID	7
Verfahren	Parametrisierter JUnit-Test
Klasse	Ant
Methoden	visitCity(City b) Wechsel der aktuellen Stadt sowie Aktualisierung der Route
Vorbedingung	City(1), City(2) (Städte haben untereinander die Distanz 2)
Eingaben	City(2)
Erwartetes Ergebnis	Aktuelle Stadt der Ameise == City(2)
Ergebnis	Aktuelle Stadt der Ameise == City(2) **Test erfolgreich**

Abbildung 5: Testfälle der Klasse *Ant*, Seite 3

ID	8
Verfahren	Parametrisierter JUnit-Test
Klasse	Ant
Methoden	visitCity(City b) Wechsel der aktuellen Stadt sowie Aktualisierung der Route
Vorbedingung	City(1), City(2), City(3) (Städte haben untereinander die Distanz 2) Ameise kann nur City(1) erreichen
Eingaben	City(2)
Erwartetes Ergebnis	Ameise bleibt in City(1)
Ergebnis	Ameise bleibt in City(1) **Test erfolgreich**

Abbildung 6: Testfälle der Klasse *Colony*, Seite 1

ID	9
Verfahren	Parametrisierter JUnit-Test
Klasse	Colony
Methoden	updatePheromone(City a, City b, double pLevel) Adds the given value to the value in the matrix between a and b
Vorbedingung	Pheromonmatrix ist mit 1 normalisiert
Eingaben	City(1), City(2), 5
Erwartetes Ergebnis	Pheromon zwischen a und b == 6
Ergebnis	Pheromon zwischen a und b == 6 **Test erfolgreich**

ID	10
Verfahren	Parametrisierter JUnit-Test
Klasse	Colony
Methoden	initPheromone() Die Pheromonmatrix muss in der Größe der Distanzmatrix von Landscape mit je 1 als Pheromonwert initialisiert werden
Vorbedingung	City(1), City(2)
Eingaben	-
Erwartetes Ergebnis	Größe der Pheromonmatrix == 2 Kolonie erkennt Pheromonmatrix als initialisiert Pheromonwert zwischen City(1) und City(2) == 1
Ergebnis	Größe der Pheromonmatrix == 2 Kolonie erkennt Pheromonmatrix als initialisiert Pheromonwert zwischen City(1) und City(2) == 1 **Test erfolgreich**

ID	11
Verfahren	Parametrisierter JUnit-Test
Klasse	Colony
Methoden	initPheromone() Die Pheromonmatrix muss in der Größe der Distanzmatrix von Landscape mit je 1 als Pheromonwert initialisiert werden
Vorbedingung	-
Eingaben	-
Erwartetes Ergebnis	PheromoneInitializationException wird geworfen
Ergebnis	PheromoneInitializationException wird geworfen **Test erfolgreich**

ID	12
Verfahren	Parametrisierter JUnit-Test
Klasse	Colony
Methoden	initAnts() Die Threads müssen zu Beginn jeder Generation erstellt werden
Vorbedingung	-
Eingaben	-
Erwartetes Ergebnis	Anzahl der Threads stimmt mit der konfigurierten Anzahl überein
Ergebnis	Anzahl der Threads stimmt mit der konfigurierten Anzahl überein **Test erfolgreich**

Abbildung 7: Testfälle der Klasse *Colony*, Seite 2

ID	13
Verfahren	Parametrisierter JUnit-Test
Klasse	Colony
Methoden	killAnt(Ant a) Sollte eine Ameise auf einen schwerwiegenden Fehler stoßen, kann sie sich selbst beenden und der Kolonie mitteilen die Referenz zu löschen
Vorbedingung	Kolonie hat derzeit eine Liste *ants* von Ameisen
Eingabenμ	Erste Ameise aus *ants*
Erwartetess Ergebnis	*ants* enthält nicht mehr die zuvor erste Ameise
Ergebnis	**Test erfolgreich**

ID	14
Verfahren	Parametrisierter JUnit-Test
Klasse	Colony
Methoden	notifyColony() Sobald eine Generation beendet ist, wird eine neue erzeugt
Vorbedingung	City(1), City(2) (Städte haben untereinander die Distanz 2)
Eingaben	-
Erwartetes Ergebnis	Liste der Ameisen der Kolonie ist größer 0 Generationenzähler steht auf 2
Ergebnis	Liste der Ameisen der Kolonie ist größer 0 Generationenzähler steht auf 2 **Test erfolgreich**

Abbildung 8: Testfälle der Klasse *Landscape*, Seite 1

ID	18
Verfahren	Parametrisierter JUnit-Test
Klasse	Landscape
Methoden	getDistance(City city, City neighbour): double Rückgabe der Distanz zweier Städte zueinander
Vorbedingung	City(1), City(2) City(2) hat zu City(1) eine Distanz von 5
Eingaben	City(1), City(2)
Erwartetes Ergebnis	5
Ergebnis	5 **Test erfolgreich**

ID	19
Verfahren	Parametrisierter JUnit-Test
Klasse	Landscape
Methoden	getNeighboursSize(): int Rückgabe der Größe der Distanzmatrix
Vorbedingung	City(1), City(2) City(2) hat zu City(1) eine Distanz von 5
Eingaben	Distanzmatrix hat eine Größe von 2
Erwartetes Ergebnis	Distanzmatrix hat eine Größe von 2
Ergebnis	**Test erfolgreich**

ID	20
Verfahren	Parametrisierter JUnit-Test
Klasse	Landscape
Methoden	getSpecifiedNeighbours(City currentCity): double[] Rückgabe der Distanzen zu allen erreichbaren Städten ausgehend von der übergebenen Stadt
Vorbedingung	City(1), City(2), City(3) Städte haben untereinander eine Distanz von je 1
Eingaben	City(1)
Erwartetes Ergebnis	{1,1}
Ergebnis	{1,1} **Test erfolgreich**

ID	21
Verfahren	Parametrisierter JUnit-Test
Klasse	Landscape
Methoden	getNeighbours(): double[][] Rückgabe der kompletten Distanzmatrix
Vorbedingung	City(1), City(2), City(3) Städte haben untereinander eine Distanz von je 1
Eingaben	-
Erwartetes Ergebnis	Distanzmatrix hat eine Länge von 3
Ergebnis	Distanzmatrix hat eine Länge von 3 **Test erfolgreich**

Abbildung 9: Testfälle der Klasse *Landscape*, Seite 2

ID	22
Verfahren	Parametrisierter JUnit-Test
Klasse	Landscape
Methoden	addNeighbours(City a, City b, double distance): int Abspeichern der übergebenen Distanz zwischen a und b in die Pheromonmatrix
Vorbedingung	-
Eingaben	City(1), City(2), 1
Erwartetes Ergebnis	Checksumme ist 1 Distanzmatrix hat eine Länge von 2 Distanz zwischen City(1) und City(2) beträgt 1
Ergebnis	Checksumme ist 1 Distanzmatrix hat eine Länge von 2 Distanz zwischen City(1) und City(2) beträgt 1 **Test erfolgreich**

ID	23
Verfahren	Parametrisierter JUnit-Test
Klasse	Landscape
Methoden	addNeighbours(City a, City b, double distance): int Abspeichern der übergebenen Distanz zwischen a und b in die Pheromonmatrix
Vorbedingung	City(1), City(2) City(1) hat eine Distanz von 1 zu City(2)
Eingaben	City(1), City(2), 1
Erwartetes Ergebnis	Checksumme ist -1 Distanzmatrix hat eine Länge von 2
Ergebnis	Checksumme ist -1 Distanzmatrix hat eine Länge von 2 **Test erfolgreich**

Abbildung 10: Testfälle der Klassen von *Parser*

ID	15
Verfahren	Parametrisierter JUnit-Test
Klasse	TSPParser
Methoden	getDistance(int xSource, int ySource, int xDestination, int yDestination): double Zwischen zwei Städten mit Koordinaten muss die Distanz über den Satz des Pythagoras ausgerechnet werden
Vorbedingung	-
Eingaben	10, 15, 32, 35
Erwartetes Ergebnis	29.7321
Ergebnis	29.7321 **Test erfolgreich**

ID	16
Verfahren	Parametrisierter JUnit-Test
Klasse	TSPParser
Methoden	parse(File file) Aus einer .tsp-Datei im Standardformat muss *Landscape* initiliasiert werden
Vorbedingung	.tsp-Datei nach Standardformat
Eingaben	Vorbereitete Datei
Erwartetes Ergebnis	Größer der Distanzmatrix von *Landscape* ist größer 0
Ergebnis	Größer der Distanzmatrix von *Landscape* ist größer 0 **Test erfolgreich**

ID	17
Verfahren	Parametrisierter JUnit-Test
Klasse	XMLParser
Methoden	parse(File file) Aus einer .xml-Datei im eigenen Format muss *Landscape* initiliasiert werden
Vorbedingung	,xml-Datei nach vorgegebenen Format
Eingaben	Vorbereitete Datei
Erwartetes Ergebnis	Größer der Distanzmatrix von *Landscape* ist größer 0
Ergebnis	Größer der Distanzmatrix von *Landscape* ist größer 0 **Test erfolgreich**

Abbildung 11: Testfälle der Klasse *HSQLDBManager*, Seite 1

ID	24
Verfahren	Parametrisierter JUnit-Test
Klasse	HSQLDBManager
Methoden	init() Alle vorhandenen Tabellen werden gelöscht und neu erstellt
Vorbedingung	Datenbank ist aktiv
Eingaben	-
Erwartetes Ergebnis	Tabelleninhalt von der Tabelle *GENERATIONS* kann abgerufen werden
Ergebnis	Tabelleninhalt von der Tabelle *GENERATIONS* kann abgerufen werden **Test erfolgreich**

ID	25
Verfahren	Parametrisierter JUnit-Test
Klasse	HSQLDBManager
Methoden	update(String sqlStatement) Ein gegebenes SQL-Statement wird ausgeführt
Vorbedingung	Datenbank ist aktiv
Eingaben	"INSERT INTO GENERATIONS (id , generation, route , distance) " + "VALUES ("+ 1 + "," + 1 + "," + "'0,1,0'" + "," + 2 + ") "
Erwartetes Ergebnis	Tabelle *Generations* hat eine Zeile
Ergebnis	Tabelle *Generations* hat eine Zeile **Test erfolgreich**

ID	26
Verfahren	Parametrisierter JUnit-Test
Klasse	HSQLDBManager
Methoden	select(String sqlStatement): ResultSet Ein gegebenes SQL-Statement wird ausgeführt und das Resultat zurückgegeben
Vorbedingung	Datenbank ist aktiv Tabelle *Generations* hat eine Zeile mit vordefinierten Daten
Eingaben	"SELECT * FROM GENERATIONS"
Erwartetes Ergebnis	Integer in der Spalte *generation* ist 1 String in der Spalte *route* ist 0,1,0 Integer in der Spalte *distance* ist 2
Ergebnis	Integer in der Spalte *generation* ist 1 String in der Spalte *route* ist 0,1,0 Integer in der Spalte *distance* ist 2 **Test erfolgreich**

ID	27
Verfahren	Parametrisierter JUnit-Test
Klasse	HSQLDBManager
Methoden	getTable(String table): ResultSet Eine komplette Tabelle wird abgerufen und zurückgegeben
Vorbedingung	Datenbank ist aktiv
Eingaben	"GENERATIONS"
Erwartetes Ergebnis	Tabelle *GENERATIONS* hat 4 Spalten
Ergebnis	Tabelle *GENERATIONS* hat 4 Spalten **Test erfolgreich**

Abbildung 12: Testfälle der Klasse *HSQLDBManager*, Seite 2

ID	28
Verfahren	Parametrisierter JUnit-Test
Klasse	HSQLDBManager
Methoden	dropTable() Alle vorhandenen Tabellen werden gelöscht
Vorbedingung	Datenbank ist aktiv Tabelle *GENERATIONS* ist vorhanden
Eingaben	-
Erwartetes Ergebnis	Tabelle *GENERATIONS* ist nicht mehr vorhanden (NullPointerException muss geworfen werden)
Ergebnis	**Test erfolgreich**

ID	29
Verfahren	Parametrisierter JUnit-Test
Klasse	HSQLDBManager
Methoden	createTable() Es werden alle Tabellen neu erstellt
Vorbedingung	Datenbank ist aktiv
Eingaben	-
Erwartetes Ergebnis	Tabelle *GENERATIONS* hat 4 Spalten Tabelle *HISTORY* hat 4 Spalten Tabelle *TEST* existiert nicht
Ergebnis	Tabelle *GENERATIONS* hat 4 Spalten Tabelle *HISTORY* hat 4 Spalten Tabelle *TEST* existiert nicht **Test erfolgreich**

ID	30
Verfahren	Parametrisierter JUnit-Test
Klasse	HSQLDBManager
Methoden	updateTable(String table, int generation, String route, int distance) Auf eine gegebene Tabelle wird ein INSERT mit den gegebenen Daten ausgeführt
Vorbedingung	Datenbank ist aktiv
Eingaben	"GENERATIONS", 1, „0,1,0“, 2
Erwartetes Ergebnis	Tabelle *GENERATIONS* hat eine Zeile
Ergebnis	Tabelle *GENERATIONS* hat eine Zeile **Test erfolgreich**