Bachelorarbeit zum Thema

Entwurf und Implementierung zur dynamischen Optimierung von Liefertouren mit einem Ameisen-System

in der Fachrichtung

Informatik

an der

Fachhochschule Wedel

Eingereicht von:	Christopher Blöcker Mozartstraße 9 22941 Bargteheide Tel.: +49 4532 24834
Erarbeitet im:	8. Verwaltungssemester
Abgegeben am:	01. September 2011
Referent :	Prof. Dr. Sebastian Iwanowski Fachhochschule Wedel Feldstrasse 143 22880 Wedel Tel.: +49 4103 804863
Betreuer:	Dipl. Ing. Thorsten Klingspor implico GmbH Weidestraße 120b 22083 Hamburg Tel.: +49 40 2709360

Man versteht etwas nicht wirklich, wenn
man nicht versucht, es zu implementieren.

(Donald E. Knuth)

Vorwort

Die Entstehung dieser Arbeit war mit einigen Turbulenzen verbunden. Ursprünglich war eine Machbarkeitsanalyse angedacht, in der ich untersuchen sollte, ob es möglich ist, ein System zu entwerfen, das Tourenpläne an dynamische Ereignisse anpasst. In diesem Zusammenhang habe ich versucht, einen Vergleich zwischen Ameisen-Systemen und Neuronalen Netzen zu ziehen und ihre Vor- und Nachteile gegeneinander abzuwägen. Nach einer Zeit von leider erst 4 Monaten bin ich dann zu der Erkenntnis gelangt, dass Neuronale Netze für diese Problemstellung ungeeignet sind. Ameisen-Systeme hingegen haben sich als sehr gut geeignet präsentiert.

Da dann schon feststand, dass es unter der Verwendung von Ameisen-Systemen machbar ist, ein System zur dynamischen Tourenplanung umzusetzen, bestand keine Notwendigkeit einer Machbarkeitsanalyse mehr. Das Thema meiner Arbeit wurde umdefiniert und so geht es nun um den Entwurf und die Implementierung eines Systems zur dynamischen Planung von Liefertouren mit einem Ameisen-System. Den Großteil meiner bis dahin entstandenen, stark theoretisch angehauchten Machbarkeitsuntersuchung musste ich dann leider verwerfen.

Glücklicherweise habe ich parallel zur Machbarkeitsanalyse schon an der Implementierung der Grundlagen für die Tourenplanung gearbeitet, also an der Umsetzung sämtlicher Strukturen, die unabhängig von Optimierungsverfahren sind und die Gegebenheiten der realen Welt abbilden. So ist es mir dann auch gelungen, innerhalb von nur 2 Monaten noch ein Ameisen-System zu implementieren und diese Arbeit quasi von Grund auf neu zu schreiben. Zum Teil war ich dabei selbst überrascht, wie schnell einige Teile fertig implementiert und dokumentiert waren. Erwartete Schwierigkeiten bei der Behandlung der dynamischen Ereignisse sind vollkommen ausgeblieben und schon beim ersten Testlauf haben die Ameisen gültige Lösungen konstruiert. Durch Tests hat sich gezeigt, dass für die Zukunft noch etwas Arbeit zu erledigen ist, um die Performance zu steigern, ein paar Fehler zu beseitigen und das System zur Marktreife zu führen, aber der Grundstein ist gelegt.

Ich hoffe, für diese Arbeit einen Schreibstil gefunden zu haben, der zum einen Freude am Lesen macht und zum anderen in der Lage ist, die besprochene Thematik ausdrucksstark zu vermitteln. Nun wünsche ich viel Spaß beim Lesen.

Hamburg, den 01. September 2011 Christopher Blöcker

Inhaltsverzeichnis

Vorwort . . . iii

1. Einführung **1**
1.1. Motivation . . . 1
1.2. Problembeschreibung und Abgrenzung . . . 2
1.3. Zielsetzung . . . 3

2. Umfeld und Überblick **4**
2.1. ATOS . . . 4
2.2. TermiDe . . . 4
2.3. IcedG . . . 5
2.4. Dyonisys . . . 5
2.5. DOT . . . 6
2.6. xServer . . . 6
2.7. Java und die Java Virtual Machine . . . 7

3. Grundlagen **9**
3.1. Graphentheorie . . . 9
3.2. Komplexitätstheorie . . . 10
3.3. Das Vehicle Routing Problem . . . 11
3.4. Das Traveling Salesman Problem . . . 12
3.5. Approximationsverfahren . . . 12
3.6. Lösungsraum . . . 13

4. Ameisen-Systeme **14**
4.1. Funktionsweise . . . 14
4.2. Varianten und Erweiterungen . . . 16
4.2.1. Elite-Ameisen . . . 16
4.2.2. Ameisen-Rangfolge . . . 16
4.2.3. Pheromonbedingungen . . . 17
4.2.4. Pheromonaktualisierung . . . 17
4.2.5. Lokale Suche . . . 17
4.2.6. Abbruchkriterien . . . 17
4.3. Detaillierter Ablauf . . . 18
4.4. Eignung . . . 20

5. Entwurf und Implementierung **21**
5.1. Basis . . . 21

5.1.1. Fahrzeug . . . 22
5.1.2. Fahrzeugdepot, Ladedepot und Lieferung . . . 23
5.1.3. Anforderungen und Ausrüstung . . . 24
5.1.4. Produkte . . . 26
5.1.5. Kontrakt und Kontingente . . . 27
5.1.6. Beladung . . . 28
5.1.7. Lieferplan . . . 30
5.1.8. Parser . . . 31
5.1.9. Datenbasis und Konfiguration . . . 33
5.2. Dyonisys . . . 34
5.2.1. Kommunikation . . . 35
5.2.2. Optimierungs-Thread . . . 36
5.2.3. Ameisen-System, Ameisen und Ausführungs-Thread . . . 37
5.2.4. Pheromone . . . 39
5.2.5. Abbildung der Dynamik . . . 40

6. Test **42**
6.1. Testtool . . . 43
6.2. Dynamische Tests . . . 44
6.2.1. Fahrzeugausfall . . . 45
6.2.2. Neue Aufträge . . . 46
6.2.3. Wegkostenveränderung . . . 47
6.3. Variation der Optimierungsparameter . . . 48
6.3.1. Anzahl der Ameisen . . . 50
6.3.2. Anzahl der Zyklen . . . 51
6.3.3. Evaporationsrate . . . 52
6.3.4. Initialpheromone . . . 53
6.3.5. Gewichtung der Pheromone . . . 54
6.3.6. Gewichtung der Heuristik . . . 55
6.4. Dyonisys vs. IcedG . . . 56

7. Abschließende Betrachtungen **58**
7.1. Zusammenfassung . . . 58
7.2. Fazit . . . 59
7.3. Ausblick . . . 59

A. JSON **61**
A.1. Gemeinsame Syntax . . . 61
A.2. Für Dyonisys angepasster Teil der Syntax . . . 63

B. Eingabeparameter **64**
B.1. Basis . . . 64
B.2. Dyonisys . . . 65

Abkürzungs- und Symbolverzeichnis **66**

Abbildungsverzeichnis **67**

Tabellenverzeichnis **69**

Literaturverzeichnis **70**

Eidesstattliche Erklärung **73**

1. Einführung

In dieser Arbeit geht es um die dynamische Optimierung von *Liefertouren*. Auch wenn wir es speziell mit den Anforderungen der Öl- und Gas-Branche zu tun haben, wollen wir unsere Untersuchungen möglichst branchenunabhängig führen.

Im Einführungskapitel wird zunächst ein informeller Überblick über die Problematik gegeben, dabei wird auf mathematische Korrektheit verzichtet, da der leichte Einstieg in die Thematik im Vordergrund stehen soll.

1.1. Motivation

Versetzen wir uns einmal in die Lage eines Öl-Konzerns.

Wir haben eine Reihe von Kunden, die Bestellungen bei uns aufgeben. Diese Bestellungen müssen unter Verwendung unseres Fuhrparks ausgeliefert werden. Jedes Fahrzeug verfügt über Ausrüstung und Eigenschaften, die den Transport einer Ware ermöglichen oder verbieten können. Die Fahrzeuge starten an einem Fahrzeugdepot und müssen zunächst an einem Warenlager betankt werden, danach folgt die Belieferung der Kunden.

Bei der Planung achten wir darauf, dass unsere Liefertouren ressourcenschonend, also kostengünstig sind und dem Unternehmen somit den höchstmöglichen Gewinn bescheren. Was jedoch nicht berücksichtigt werden kann, sind Ereignisse, die in der realen Welt unvorhersehbar auftreten. Der Ausfall eines Fahrzeugs oder ein Notfall-Auftrag sind Beispiele, die den gesamten Plan unbrauchbar machen können.

Für diesen Anwendungsfall soll ein Verfahren entworfen und umgesetzt werden, welches imstande ist, die Tourenpläne *just-in-time* an neue, unerwartete Situationen anzupassen. Dieses Verfahren soll langfristig in die von der *implico GmbH* entwickelte Software zur Distributionsplanung integriert werden.

1.2. Problembeschreibung und Abgrenzung

Wir beschreiben nun das zu untersuchende Problem genauer und nehmen dabei auch einige abgrenzende Vereinfachungen vor. Dadurch wollen wir Sonderfälle ausschließen, um uns auf den Kern des Problems konzentrieren zu können, die Einbeziehung zu einem späteren Zeitpunkt schließen wir damit jedoch nicht aus.

Wie bereits erwähnt, haben wir eine Reihe von Kunden, die Bestellungen bei uns aufgeben und von Fahrzeugen aus unserem Fuhrpark beliefert werden sollen. Die Kunden haben gewisse Anforderungen an die Lieferfahrzeuge, wie z.B. eine Mindestschlauchlänge oder ein Firmenlogo, das auf dem Fahrzeug prangen soll. Somit verfügt jeder Kunde über eine Menge von Anforderungen, jedes Fahrzeug über eine Menge von Eigenschaften. Damit ein Fahrzeug für die Lieferung zu einem bestimmten Kunden eingesetzt werden kann, müssen die Anforderungen des Kunden durch das Fahrzeug erfüllt werden. Die Anforderungen lassen sich in die Kategorien *hart* und *weich* unterteilen. Die Verletzung einer harten Bedingung hat zur Folge, dass der Tourenplan ungültig wird, die Verletzung einer weichen Bedingung hingegen verursacht lediglich (fiktive) Strafkosten.

Die Liefertour eines jeden Fahrzeugs beginnt und endet an einem Fahrzeugdepot, dabei kehrt das Fahrzeug immer zu dem Depot zurück, von dem es auch gestartet ist. Zum Startzeitpunkt kann das Fahrzeug entweder voll betankt oder komplett leer sein. Im Laufe des Tages kann es erforderlich sein, mehrmals ein Tanklager anzufahren, um den Fahrzeugtank für weitere Auslieferungen zu befüllen. An den Tanklagern wird von einer mittleren Wartezeit ausgegangen, bevor mit dem Tankvorgang begonnen werden kann. Art und Menge der verfügbaren Waren können sich ebenso wie ihr Preis von Lager zu Lager unterscheiden. Getankt werden kann nur innerhalb der Öffnungszeiten des Lagers.

Jedem Kunden ist ein Zeitplan zugeordnet, dessen Zeitfenster angeben, wann die Zustellung seiner Bestellung erfolgen soll. Die Zeitfenster werden dabei derart interpretiert, dass sie angeben, wann mit der Belieferung frühestens bzw. spätestens begonnen werden darf. Wir gehen davon aus, dass die Bestellmengen jeweils in ein einziges Fahrzeug passen und eine Aufteilung auf mehrere Fahrzeuge nicht erforderlich ist.

Bei bestimmten Waren kann es vorkommen, dass ein Fahrzeug sie aufgrund technischer Gegebenheiten nicht befördern kann. Ebenso besteht die Möglichkeit, dass es Waren gibt, die nicht in Kombination transportiert werden dürfen (bestimmte Stoffe können z.B. chemisch reagieren oder den Verderb einer anderen Ware verursachen).

Wir stellen bei der Tourenplanung sicher, dass die gesetzlich vorgeschriebenen Arbeits- und Pausenzeiten eingehalten werden und setzen voraus, dass die Fahrer ihren Arbeitstag in ausgeruhtem Zustand beginnen.

1.3. Zielsetzung

Im Rahmen dieser Arbeit wird es unser Ziel sein, ein praxistaugliches Lösungsverfahren für die Tourenplanung zu entwickeln, wobei vor allem der Aspekt der dynamischen Veränderung der Probleminstanzen berücksichtigt werden soll. Die zu behandelnden Ereignisse sind das Ausfallen von Fahrzeugen, das Hinzukommen von Aufträgen und das Verändern von Wegkosten. Unsere Ziele werden die Qualität der berechneten Lösungen sowie eine niedrige Laufzeit sein, damit das Verfahren einen praktischen Nutzen bietet.

Wir werden zu Beginn einen Überblick über die notwendige Theorie geben und dabei sehen, dass das Problem der Tourenplanung schwer zu lösen ist. Danach werden wir betrachten, worum es sich bei einem Ameisen-System handelt, und prüfen, ob es den Anforderungen, die im Zusammenhang mit der Tourenplanung enstehen, gerecht werden kann. Auf der Grundlage eines Ameisen-Systems werden wir dann ein Lösungsverfahren entwerfen und implementieren.

Zum Schluss werden wir einige Tests durchführen und das erarbeitete Lösungsverfahren mit einem vergleichen, das bereits in der Praxis zum Einsatz kommt und auf der Tabu-Suche basiert.

2. Umfeld und Überblick

Die *implico GmbH* entwickelt seit etwa 10 Jahren Software zur Distributionsplanung für die Öl- und Gasbranche auf der Grundlage von SAP-Systemen. Eines der entwickelten Produkte ist das *Integrated Dispatch Management* (kurz: IDM). Das Ziel von IDM ist es, Ölkonzerne wie z.B. Shell oder Aral bei der Planung ihrer Liefertouren zu unterstützen. Für die Unternehmen besteht ein großer Vorteil von IDM darin, dass es sich nahtlos in SAP-Systeme integrieren lässt und somit sämtliche Produktivdaten sowie die Planungswerkzeuge in ein und demselben System verfügbar macht. Dadurch können Geschäftsprozesse enorm vereinfacht und Kosten gesenkt werden.

Im Rahmen dieser Arbeit soll das *dynamic optimization nature-inspired system* Dyonisys entstehen, welches in Echtzeit auf Ereignisse der realen Welt reagiert und die Liefertouren an diese anpasst. Parallel werden von Timo Jürgens ein Terminvorschlags-System für den Telefonverkauf (TermiDe) und von Nicolas Woldt ein Tourenoptimierer (IcedG) entwickelt. Für die Kommunikation mit dem SAP-System wird die von André Bente entwickelte *Allgemeine Touren-Oprimierungs Schnittstelle* ATOS verwendet.

2.1. ATOS

ATOS wurde von André Bente im Rahmen seiner Bachelor-Arbeit entwickelte und kapselt sämtliche Kommunikation zwischen dem SAP-System und den Optimierungskomponenten. Durch die Verwendung von ATOS ist einerseits die zentrale Datenhaltung im SAP-System möglich, andererseits können die Optimierer transparent mit den für die Planung nötigen Daten versorgt werden. Auch die Rückerfassung verplanter und nicht-verplanter Aufträge gestaltet sich auf diese Weise einfach.

2.2. TermiDe

TermiDe soll in der Vordisposition eingesetzt werden und die Verkäufer beim Vergeben von Lieferterminen am Telefon unterstützen. Es wird dabei bereits für mehrere Wochen im Voraus ein grober Lieferplan erstellt und beim Eintreffen neuer Aufträge erweitert. TermiDe berechnet für jeden neuen Auftrag mögliche Lieferzeiten und schlägt sie dem Verkäufer vor, dieser macht dann einen Termin mit dem Kunden aus.

Das Ziel von TermiDe ist es, einen vorläufigen Plan zu erstellen, der garantiert gültig ist, also unter Verwendung der vorhandenen Fahrzeuge und Produkte ausgeführt werden kann. Es soll auf diese Weise ausgeschlossen werden, dass in der nachfolgenden Optimierung durch IcedG unlösbare Szenarien verarbeitet werden müssen. Die Forderungen bezüglich der Antwortzeit von TermiDe sind dabei sehr hoch, es soll nach bereits wenigen Sekunden angegeben werden, ob ein Kunde am gewünschten Termin beliefert werden kann, um langes Warten am Telefon zu vermeiden.

2.3. IcedG

IcedG erhält als Eingabe einen von TermiDe berechneten Plan, der garantiert gültig ist. Es sollen nun die Kosten minimiert werden, denn TermiDe macht keine Aussagen bezüglich der Qualität des Plans.

Zur Optimierung setzt IcedG die Tabu-Suche ein. Dabei werden die Aufträge des bisherigen Plans solange miteinander getauscht, bis keine guten Vertauschungen mehr möglich sind. Vertauschungen werden als gut betrachtet, solange sie keine zu hohe Kostenverschlechterung bedeuten. Bereits durchgeführte sowie zu schlechte Vertauschungen werden in eine Tabu-Liste eingetragen und dürfen nicht erneut durchgeführt werden. Die Kosten des Plans werden dabei in der Regel gesenkt, das Finden des optimalen Plans ist möglich, kann aber nicht garantiert werden.

Das Ziel von IcedG ist es, die Kosten des Plans zu minimieren. Dabei wird in Kauf genommen, dass die Berechnung sehr lange dauert und so findet die Optimierung mit IcedG jeweils in der Nacht vor dem Tag der Ausführung des Plans statt. Der durch IcedG optimierte Plan dient als Eingabe für Dyonisys.

2.4. Dyonisys

Dyonisys erhält als Eingabe den von IcedG optimierten Plan für den aktuellen Tag und wartet dann auf weitere Eingaben. Tritt ein Ereignis ein, also fällt z.B. ein Fahrzeug aus oder ein neuer, hoch priorisierter Auftrag trifft ein, so wird Dyonisys darüber informiert. Dyonisys versucht dann, den Plan an die neue Situation anzupassen und dabei die Kosten möglichst gering zu halten. Es kann passieren, dass durch die eingetretenen Ereignisse Situationen entstehen, in denen nicht alle ursprünglich geplanten Kunden beliefert werden können.

Ziel von Dyonisys ist es, innerhalb weniger Minuten einen neuen Plan zur Verfügung zu stellen. Da das Auffinden eines kostenminimalen Ersatzplans in der Kürze der Zeit nicht garantiert werden kann, werden auch weniger gute Pläne akzeptiert. Wichtig ist es, *dass* ein Plan, und somit eine Entscheidungsgrundlage, vorhanden ist.

Stell sich am Ende des Tages heraus, dass es Aufträge gibt, die hätten beliefert werden sollen, es aber nicht wurden, so wird TermiDe darüber informiert und plant die betroffenen Aufträge neu ein.

Der Name Dyonisys zeigt einerseits, was das System tut, nämlich dynamische Problemstellungen mit einem Verfahren zu optimieren, das der Natur nachempfunden ist, andererseits ist er aber auch in Anlehnung an den griechischen Gott der Freude und Ernte gewählt, lediglich die Schreibweise ist etwas abgewandelt.

2.5. DOT

DOT ist das *Disposition Optimization Toolset*, welches sich aus TermiDe, IcedG und Dyonisys zusammensetzt und die Funktionalität der drei Module zusammenfasst. Die Kommunikationswege, über die DOT mit Hilfe von ATOS einheitlich angesprochen und verwendet werden kann, sind in (Abbildung 2.1, S. 6) veranschaulicht.

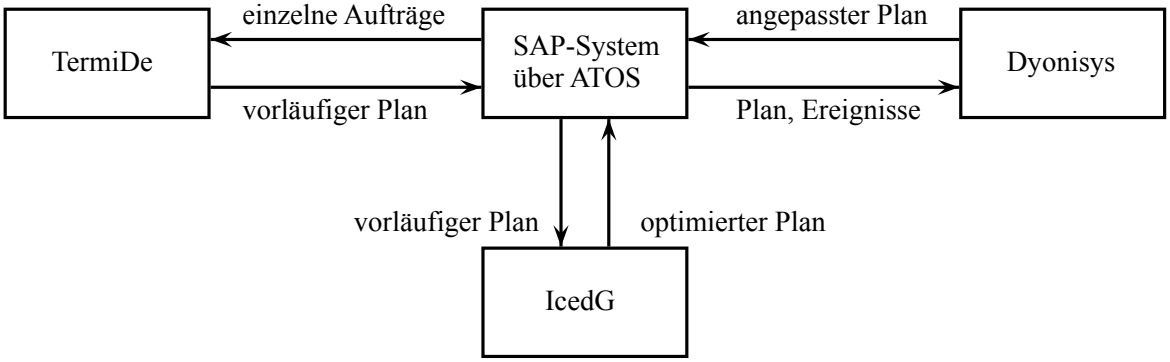


Abbildung 2.1.: Interaktion von TermiDe, IcedG und Dyonisys

2.6. xServer

Die xServer sind ein von der PTV AG hergestelltes Softwarepaket zur Verkehrsoptimierung. Sie stellen mit xMap, xLocate und xRoute verschiedene Funktionalitäten zur Verfügung.

xMap liefert Kartenmaterial und bietet die Möglichkeit, dieses anzuzeigen. Mit xLocate lässt sich die Zuordnung von Adressen auf Geokoordinaten vornehmen. Über xRoute können Wege zwischen Geokoordinaten bestimmt werden. Die für das Routing verwendeten Kriterien sind dabei konfigurierbar.

2.7. Java und die Java Virtual Machine

Als Programmiersprache für die Entwicklung von DOT soll Java verwendet werden. Ursprünglich standen auch die von der SAP entwickelte, proprietäre Programmiersprache ABAP und C++ zur Diskussion. Die Entscheidung gegen ABAP ist jedoch sehr schnell gefallen, da ABAP als Programmiersprache der 4. Generation[1] ein hohes Abstraktionsniveau aufweist und somit für rechenintensive Optimierungsaufgaben und maschinennahe Programmierung potenziell eher ungeeignet ist. Außerdem konnte in Tests, wie z.B. am Euklidischen Algorithmus, gezeigt werden, dass die Performance von Programmen unter ABAP ihrem Pendant in Java und C++ deutlich unterlegen ist. Dies lässt sich mit dem hohen Vorkommen arithmetischer Operationen und der Tatsache begründen, dass ABAP interpretiert, Java und C++ aber kompiliert werden.

Zum einen fiel die Entscheidung schlussendlich auf Java, da hier die Entwicklung u.a. durch ein konsequenteres Sprachdesign und automatische Speicherverwaltung enorm erleichtert wird. Zum anderen spielen unternehmenspolitische Grunde eine Rolle, so lassen sich auf dem Arbeitsmarkt leichter Java-Entwickler als C++-Entwickler finden.

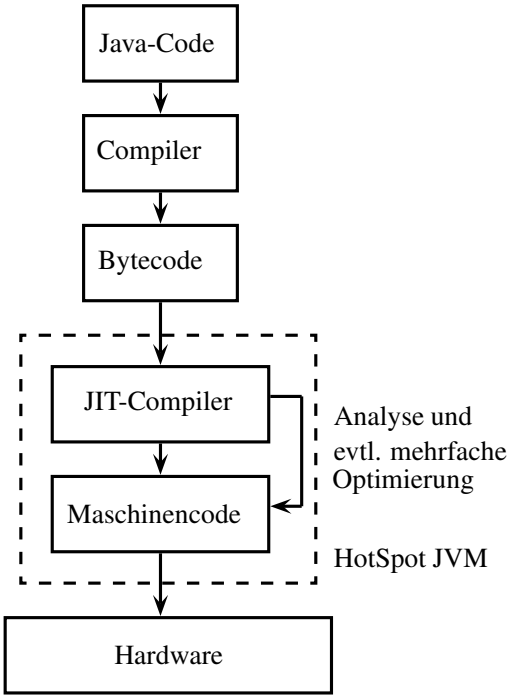


Abbildung 2.2.: Vereinfachte Darstellung der Ausführung von Java-Programmen mit der HotSpot JVM

Auch bezüglich der Performance müssen keine Bedenken bestehen, Java hat seinen schlechten Ruf in den vergangenen Jahren durch die Optimierung der Java Virtual Machine (kurz: JVM) wettmachen können. In Java geschriebene Programme werden zunächst in ein Zwischen-Format kompiliert, den sogenannten Bytecode. Die JVM nimmt zur Ausführungszeit den Bytecode her und übersetzt ihn mit Hilfe eines Just-In-Time-Compilers (JIT) in Maschinencode, der dann direkt auf der Hardware zur Ausführung kommt. Während der Ausführung kann die JVM den Codeablauf verfolgen und analysieren und wird gegebenenfalls häufig benutzte Codebereiche (sogenannte HotSpots) optimieren und den Maschinencode durch den optimierten ersetzen. Dies geschieht mit einem Verfahren, das Hot Swapping genannt wird und das Austauschen von Maschi-

[1] Als Programmiersprache der 4. Generation werden Sprachen bezeichnet, die ein hohes Abstraktionsniveau aufweisen und die Erstellung komplexer Anwendungen in wenigen Zeilen ermöglichen, es wird nicht beschrieben **wie** etwas gemacht werden soll, sondern **was** gemacht werden soll.

nencode während der Ausführung erlaubt. Das augeführte Programm kann mit dem optimierten Code unmittelbar weiterlaufen.

Außerdem bietet sich der große Vorteil, dass Java-Bytecode auf jeder Architektur ausgeführt werden kann, für die eine JVM implementiert wurde. Portierungen gestalten sich auf diese Weise sehr einfach und erfordern aufgrund der Standardisierung der JVM (prinzipell) keine Anpassungen.

3. Grundlagen

Dieses Kapitel gibt einen Überblick über die Grundlagen, die zum Verständnis der weiteren Arbeit erforderlich sind. Wir werden einige Begriffe aus der Theorie betrachten und sie in Zusammenhang mit unserer Problemstellung bringen.

3.1. Graphentheorie

Die Graphentheorie bildet ein Teilgebiet der Mathematik, sie befasst sich mit der Untersuchung der Eigenschaften sogenannter Graphen. Graphen bestehen aus einer Menge von Objekten, auch *Knoten* genannt, und einer Menge von Verbindungen zwischen den Objekten, auch *Kanten* genannt. Die Kanten können *gerichtet* oder *ungerichtet* sein, d.h. eine Richtung besitzen oder nicht. Gerichtete Kanten symbolisieren wir durch einen Pfeil.

Wir können Graphen zur Modellierung verschiedener Dinge verwenden, z.B. für ein Straßennetz. Die Knoten entsprechen dann den Kreuzungen, die Kanten den Straßen zwischen den Kreuzungen. Auf ungerichteten Kanten kann in beide Richtungen verkehrt werden, während gerichtete Kanten Einbahnstraßen darstellen.

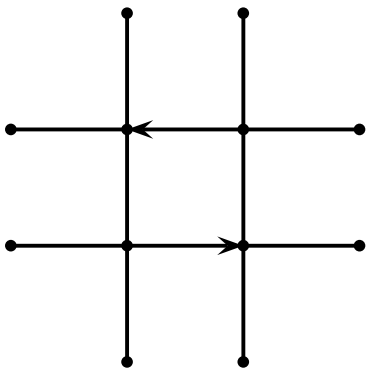


Abbildung 3.1.: Ausschnitt des Straßennetzes von Manhattan mit Einbahnstraßen, modelliert als Graph.

Es ist auch möglich, dass die Kanten und Knoten mit einem Wert versehen sind, in diesem Fall sprechen wir von einem *kanten-* bzw. *knotenbewerteten* Graphen. In einem Straßennetzes kann der Wert einer Kante z.B. für ihre Länge stehen, der Wert eines Knoten für die Zeit, die gewartet werden muss, ehe die Kreuzung passiert werden kann.

Einen Graphen, bei dem jedes Paar von Knoten miteinander verbunden ist, bezeichnen wir als *vollständig*. Wenn es zwischen zwei Knoten eine Kante gibt, dann bezeichnen wir diese Knoten als *adjazent*. Über eine Kante sagen wir, dass sie mit einem Knoten *inzident* ist, wenn er einer der Endpunkt dieser Kante ist.

Definition 3.1. ***Graph***
Ein Graph $G = (V, E)$ ist ein Paar, bestehend aus einer Menge V von Knoten und einer Menge E von Kanten, die je zwei Knoten miteinander verbinden. Zusätzlich können Bewertungsfunktionen für die Kanten bzw. Knoten gegeben sein.

In den nachfolgenden Kapiteln werden wir Graphen verwenden, um das Problem der Tourenplanung zu modellieren. Für weitere Informationen zum Thema Graphentheorie sei auf [Beu07] und [Tu09] verwiesen.

3.2. Komplexitätstheorie

Die Komplexitätstheorie untersucht die Schwierigkeit von Problemen und trifft Aussagen darüber, wie viel *Zeit* und *Speicherplatz* zur Lösung eines Problems mindestens notwendig sind. Um die Komplexität eines konkreten Problems festzustellen, werden die Algorithmen betrachtet, die es lösen. Der *beste bekannte* Algorithmus gibt dann die Komplexität des Problems an. Es ist durchaus möglich, dass es noch bessere Algorithmen gibt, diese aber bisher nicht bekannt sind und somit die Komplexität des Problems geringer ist als vermutet.

Die Komplexität eines Problems wird in der *Anzahl an Rechenschritten* gemessen, die der beste bekannte Algorithmus machen muss, um es zu lösen. Es ist klar, dass wir für den praktischen Einsatz Algorithmen suchen, die schnell zu einem Ergebnis kommen. Da es im allgemeinen sehr schwierig ist, exakt anzugeben, wieviele Schritte ein Algorithmus benötigt, wird hierbei mit *oberen Schranken* gerechnet. Diese Schranken werden in Abhängigkeit von der Eingabelänge in Form von Funktionen unter Verwendung der sogenannten $\mathcal{O}$-Notation[1] beschrieben.

Die *maximale* Anzahl an Rechenschritten, die ein Algorithmus $\mathcal{A}$ benötigt, um für Eingaben der Länge n die Lösung zu berechnen, bezeichnen wir als $t_{\mathcal{A}}(n)$. Wenn es eine Funktion f gibt, die für jedes $n \in \mathbb{N}$ größer ist als $t_{\mathcal{A}}$, dann schreiben wir dafür $t_{\mathcal{A}} \in \mathcal{O}(f)$[2]. Wenn es sich bei f um ein Polynom handelt, f also geschrieben werden kann als

$$f(n) = a_i \cdot n^{c_i} + \ldots + a_0 \cdot n^{c_0}, \tag{3.1}$$

dann sagen wir, dass $\mathcal{A}$ in polynomieller Zeit läuft. Für gewöhnlich werden dabei Konstanten ignoriert und nur der größte Exponent betrachtet. So würden wir z.B. nicht $t_{\mathcal{A}} \in \mathcal{O}(2 \cdot n^3 + n^2)$ schreiben, sondern $t_{\mathcal{A}} \in \mathcal{O}(n^3)$.

Die Menge aller Algorithmen mit polynomieller Laufzeit bezeichnen wir als $\mathcal{P}$, dies sind genau die Algorithmen, die auf einem Computer in praxistauglicher Zeit zu einem Ergebnis kommen. Ist jedoch von einem Problem bekannt, dass zu dessen Lösung *mindestens exponentielle* Laufzeit vonnöten ist, dann können wir es nicht in akzeptabler

[1] gesprochen: "Groß-O-Notation".
[2] gesprochen: "$t_{\mathcal{A}}$ ist von der Größenordnung f".

Zeit lösen. Probleme mit dieser Eigenschaft werden als $\mathcal{NP}$-Vollständig bezeichnet. Es ist gezeigt worden, dass alle Probleme mit dieser Eigenschaft gleich schwer sind [Ka72].

Bisher ist es nicht bekannt, ob sich $\mathcal{NP}$-Vollständige Probleme überhaupt mit einem Polynomialzeit-Algorithmus lösen lassen. Es wird jedoch weitgehend vermutet, dass dies nicht der Fall ist [Co00]. Umfangreiche Informationen zum Thema Komplexitätstheorie können in [GJ79] und [Sip06] gefunden werden.

3.3. Das Vehicle Routing Problem

Das Problem der Tourenplanung ist in der Theorie bekannt als Vehicle Routing Problem (kurz: VRP) und wird mit Hilfe von Graphen beschrieben.

Gegeben ist ein kanten- und knotenbewerteter, vollständiger Graph $G = (V, E)$ mit Funktionen zur Bewertung der Kanten und Knoten

- $\phi\colon E \to \mathbb{N}$: Die Bewertungsfunktion der Kanten (Reisekosten)
- $\delta\colon V \to \mathbb{N}$: Die Bewertungsfunktion der Knoten (Warennachfrage)

Zusätzlich gibt es n_f Fahrzeuge $f_1, \ldots, f_{n_f}$ der Kapazität k. v_0 ist das Waren- und Fahrzeugdepot[3], die restlichen Knoten sind Kunden.

Für jedes Fahrzeug ist eine Tour gesucht, sodass insgesamt möglichst viele der Aufträge beliefert werden und die Gesamtkosten aller Touren minimal sind. Bei der Beladung der Fahrzeuge darf ihre Kapazität nicht überschritten werden.

Das VRP ist $\mathcal{NP}$-Vollständig und somit liegt die Vermutung nahe, dass es nicht in Polynomialzeit gelöst werden kann. Weitere Informationen hierzu können in [GJ79] und [LLRS85] gefunden werden.

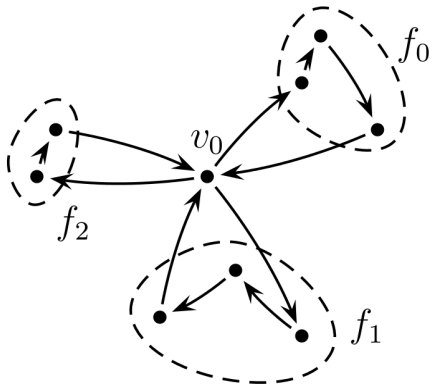


Abbildung 3.2.: Eine Instanz des VRP mit 3 Fahrzeugen f_0, f_1, f_2, den zugeordneten Touren und dem Depot v_0.

[3] Wenn nötig durch Umbenennung der Knoten.

3.4. Das Traveling Salesman Problem

Das Traveling Salesman Problem (kurz: TSP) ist eng mit dem VRP verwandt und stellt eine ähnliche Problemstellung dar. Je nach Betrachtungsweise ist das TSP ein Sonderfall des VRP bzw. das VRP die Verallgemeinerung vom TSP. Wir werden das TSP im folgenden Kapitel heranziehen, um die Funktionsweise von Ameisen-Systemen zu erklären.

Beim TSP geht es darum, dass ein Verkäufer eine Menge von Städten besuchen muss, um danach wieder zu seinem Startort zurückzukehren. Es gilt, die Reihenfolge der zu besuchenden Städte so zu bestimmen, dass die Reisekosten minimiert werden.

Das TSP wird, genau wie das VRP, als Graph modelliert, bei dem die Knoten den Städten entsprechen und die Kanten den Wegen zwischen ihnen. Üblicherweise ist der Graph *vollständig*.

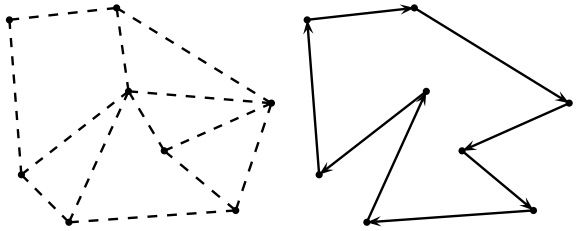


Abbildung 3.3.: Links eine Instanz des TSP und rechts eine mögliche Lösung dazu.

Das TSP ist, wie das VRP, ein $\mathcal{NP}$-vollständiges Problem, für das die optimale Lösung vermutlich nicht in akzeptabler Zeit bestimmt werden kann. Ein Buch, das sich fast ausschließlich mit dem TSP befasst, ist [LLRS85]. Weitere Informationen können aber auch in [GJ79] und [Wan06] gefunden werden.

3.5. Approximationsverfahren

Mit einem Approximationsverfahren[4] meinen wir ein Verfahren, das zu einem gegebenen Problem eine *beliebige* Lösung berechnet. Es muss sich dabei *nicht* um die beste Lösung handeln, sie muss lediglich gültig sein.

Verzichtet man auf das Finden der optimalen Lösung, so kann die Laufzeit häufig enorm gesenkt werden und sogar für $\mathcal{NP}$-vollständige Probleme eine Lösung im Polynomialzeit gefunden werden. In der Regel wird der Ansatz verfolgt, dass, ausgehend von einer Startlösung, Schritt für Schritt eine Verbesserung erzielt werden soll und somit eine Annäherung an das Optimum erfolgt[5].

Für viele Approximationsverfahren ist bekannt, wie nahe sie der optimalen Lösung

[4] Lateinisch von *proximus*: "der Nächste".
[5] Es handelt sich dabei um sogenannte "iterative Verfahren".

kommen, es kann also eine Vorhersage über die Abweichung im schlimmsten Fall gemacht werden. Wir sprechen dabei von einer Gütegarantie. Ebenso gibt es aber auch Approximationsverfahren, für die eine derartige Vorhersage nicht bekannt oder nicht möglich ist. Ameisen-Systeme zählen zu Approximationsverfahren *ohne* Gütegarantie.

Zwei gute Bücher, die sich mit Approximationsverfahren beschäftigen sind [Vaz01] und [Wan06].

3.6. Lösungsraum

Als Lösungsraum eines Problems bezeichnen wir die Gesamtheit aller prinzipiell denkbarer Lösungen. Häufig sind viele der Lösungen aus dem Lösungsraum nicht gültig, da sie geforderte Eigenschaften nicht aufweisen. Im Fall der Suche nach einem Weg zwischen zwei Orten gibt es in der Regel viele Möglichkeiten, davon werden aber häufig nur wenige eine geforderte Maximallänge nicht überschreiten. Im Fall des TSP besteht der Lösungsraum aus allen möglichen Rundreisen.

Intuitiv scheint es gegeben zu sein, dass ein Problem umso schwieriger ist, je größer der zugehörige Lösungsraum ist, da es dann auch länger dauert, ihn zu durchsuchen. Dies ist jedoch nicht der Fall. Die Schwierigkeit eines Problems hängt nicht von der Größe des Lösungsraumes ab, wie [VV86] gezeigt haben. Sie ist eine *dem Problem immanente* Eigenschaft.

4. Ameisen-Systeme

Wie der Name bereits vermuten lässt, ist die Idee der Ameisen-Systeme dem Verhalten von Ameisen in der Natur nachempfunden.

Bei der Suche nach Futter markieren die Ameisen den von ihnen zurückgelegten Weg durch die Ausschüttung von *Pheromonen*[1]. Andere Ameisen nehmen diese Markierungen wahr und folgen bevorzugt den Wegen, die eine hohe *Pheromonkonzentration* aufweisen. Hat eine Ameise Futter gefunden, so kehrt sie auf demselben Weg zum Nest zurück, den sie gekommen ist, und legt dabei erneut *Pheromone* ab, sie hat ihren Weg dann zweimal markiert. Durch das Zusammenwirken vieler Ameisen, bildet sich mit der Zeit der Weg heraus, der zur Futterquelle führt. Ein Großteil der Ameisen verwendet ihn, während ein kleiner Teil neue Wege erkundet, um weitere Futterquellen zu erschließen. Der Effekt der *indirekten* Kommunikation über die Pheromone wird als *Emergenz* bezeichnet. Die Gesamtheit aller Pheromone lässt sich als *gemeinsames, verteiltes Gedächtnis* der Ameisen darüber interpretieren, welche Wege auf der Futtersuche Erfolg versprechen.

Bei Ameisen-Systemen handelt es sich um *metaheuristische Optimierungsverfahren.* Unter einer *Metaheuristik* verstehen wir ein allgemeines Lösungsverfahren, das unter Verwendung einer problemspezifischen Heuristik auf eine Vielzahl von Problemen angewendet werden kann. Außerdem handelt es sich bei Ameisen-Systemen um Approximationsverfahren, wie wir sie in (Kapitel 3.5, S. 12) kennengelernt haben, sie berechnen also stets nur eine Näherung und können nicht garantieren, die optimale Lösung zu finden. Ein umfangreiches Werk, das sich ausschließlich mit Ameisen-Systemen befasst und unterschiedliche Varianten vorstellt, ist das von Dorigo und Stützle verfasste Buch *"Ant Colony Optimization"*, siehe dazu [DS04].

4.1. Funktionsweise

Zur Erklärung der Funktionsweise von Ameisen-Systemen werden wir das TSP heranziehen. Die Eingabe besteht aus einem *kantenbewerteten, vollständigen* Graphen[2]. Als Lösung suchen wir eine Rundreise *minimaler* Länge. Zur besseren Veranschaulichung stellen wir uns die Knoten des Graphen als Kreuzungen und die Kanten als Wege vor.

[1] Bei *Pheromonen* handelt es sich um Duftstoffe, die zur Markierung von Wegen eingesetzt werden.
[2] siehe (Kapitel 3.1, S. 9)

Die Berechnung der Lösung findet in verschiedenen Phasen statt, zu Beginn läuft einmalig die *Initialisierungsphase* ab, danach schließen sich abwechselnd mehrere Durchläufe von *Konstruktionsphase* und *Verwaltungsphase* an. Je ein Ablauf von Konstruktionsphase, gefolgt von der Verwaltungsphase wird als *Zyklus* bezeichnet. Vor Start der Berechnungen muss durch den Anwender festgelegt werden, wieviele Zyklen durchlaufen werden sollen.

In der Initialisierungsphase werden zunächst *Initialpheromone* auf allen Wegen abgelegt, danach wird einer der Knoten des Graphen als Nest ausgewählt. Alle Ameisen werden im Nest platziert und beginnen hier in jedem Durchlauf der Konstruktionsphase mit dem Berechnen einer Lösung. Alternativ kann in jedem Zyklus pro Ameise ein zufälliger Startknoten ausgewählt werden.

In der Konstruktionsphase konstruiert jede Ameise einen Lösungskandidaten[3]. Die Konstruktion beginnt stets am Startknoten. Jede Ameise wählt solange aus den von ihr *unbesuchten* Knoten einen aus und besucht ihn, bis sie an jedem Knoten *genau einmal* gewesen ist, dann kehrt sie zum Nest zurück. Hierfür ordnet sie jedem unbesuchten möglichen Folgeknoten einen Wert zu, der angibt, wie wahrscheinlich es ist, dass sie ihn als nächsten besuchen wird. Zur Berechnung dieser *Besuchswahrscheinlichkeiten* werden eine *Heuristik*[4] und die *Pheromonkonzentration* auf den Kanten verwendet. Die Heuristik bewertet Kanten anhand ihrer Länge, wobei kurze Kanten eine bessere Bewertung erhalten als lange. Kanten, die einen guten heuristischen Wert und eine hohe Pheromonkonzentration aufweisen, präsentieren sich als besonders attraktiv für die Ameisen. Die Auswahl des nächsten Knotens findet zufallsbasiert statt, d.h. es ist durchaus *möglich*, dass ein unattrativer Knoten, also einer mit einer geringen Besuchswahrscheinlichkeit, ausgewählt wird.

Wenn alle Ameisen eine Rundreise bestimmt haben, schließt sich die Verwaltungsphase an, in der die *Pheromonupdates* durchgeführt werden. Zuerst verdampft ein Teil der bisher vorhandenen Pheromone, dieser Effekt ist ebenfalls der Natur nachempfunden und wird *Evaporation* genannt. Danach werden entlang der durch die Ameisen bestimmten Rundreisen auf den benutzten Kanten neue Pheromone abgelegt. Die Menge der neu ausgeschütteten Pheromone hängt von der Qualität der jeweiligen Lösung, also der Länge der Tour, ab: je kürzer sie ist, desto höher ist die Konzentration der neu gelegten Pheromone.

Der Ablauf des Ameisen-Systems wird vorzeitig abgebrochen, falls *Stagnation* eintritt. Als Stagnation wird eine Situation bezeichnet, in der alle Ameisen dieselbe Lösung konstruieren. Ist das System mit der Berechnung fertig, so wird die insgesamt beste gefundene Lösung ausgegeben. Ameisen-Systeme bieten jedoch den immensen Vorteil, dass zu *jedem* Zeitpunkt die bisher beste bekannte Lösung erfragt werden kann, es

[3] Die Bezeichnung als "Lösungskandidat" rührt daher, dass jede konstruierte Lösung als Kandidat für die Auswahl der besten Lösung infrage kommt.

[4] Altgriechisch von $\epsilon\nu\rho\iota\sigma\kappa\omega$: "ich finde". Die Verwendung einer Heuristik soll mit begrenztem Wissen zu guten Ergebnissen führen und kann als "kluges Raten" aufgefasst werden.

muss also nicht notwendigerweise bis zum Ende der Berechnung gewartet werden. Bei den Lösungen, die vor dem Feststehen des Endergebnisses abgefragt werden, muss aber beachtet werden, dass sie diesem *deutlich* unterlegen sein *können*. In einigen Anwendungsfällen wird dies jedoch in Kauf genommen, um überhaupt über eine Lösung, und somit eine Entscheidungsgrundlage, zu verfügen.

In Versuchen konnte gezeigt werden, dass sich Ameisen-Systeme zur Lösung einer Vielzahl von Problemen verwenden lassen und dabei sehr gute Ergebnisse liefern können, es kann jedoch nicht davon ausgegangen werden, dass die optimale Lösung gefunden wird. Wir müssen stets davon ausgehen, nur eine Näherung zu erhalten.

Hauptsächlich werden Ameisen-Systeme zum Lösen $\mathcal{NP}$-vollständiger Probleme verwendet, einige Beispiele sind Anwendungen zur *Protein-Sequenzierung*, zum *Scheduling* und zum *Routing*. Dorigo und Stützle stellen in ihrem Buch [DS04] eine Reihe von Anwendungsfällen vor.

4.2. Varianten und Erweiterungen

Die in [DS04] beschriebenen Ameisen-Systeme laufen im groben alle nach dem in (Kapitel 4.1, S. 14) beschriebenen Prinzip ab. Es existiert jedoch eine Reihe von Varianten und Erweiterungen, deren Zweck es ist, die Konfigurierbarkeit zu steigern und bessere Ergebnisse zu erzielen. Einige, prinzipiell in jedem Ameisen-System anwendbare, wollen wir hier kurz betrachten.

4.2.1. Elite-Ameisen

Es werden Elite-Ameisen hinzugefügt, die die Pheromonkonzentration entlang der besten Lösung oder der besten Lösungen verstärken. Dadurch soll erreicht werden, dass sich die restlichen Ameisen bevorzugt an diese Lösungen halten und in der Nähe dieser nach einer Verbesserung suchen.

Üblicherweise wird über einen Parameter gesteuert, wieviele der besten Lösungen durch Elite-Ameisen verstärkt werden sollen.

4.2.2. Ameisen-Rangfolge

Nach der Lösungsberechnung dürfen nicht alle, sondern nur die besten Ameisen neue Pheromone legen. Es wird eine Rangfolge der Ameisen entsprechend der Qualität der gefundenen Lösungen bestimmt. Anhand eines Parameters, der festlegt, bis zu welchem Rang die Ameisen Pheromone legen dürfen, findet dann die Ausschüttung statt. Auf diese Weise soll die Ausschüttung von Pheromonen auf Strecken, die zu einer der schlechteren Lösungen gehören, verhindert werden.

4.2.3. Pheromonbedingungen

Durch das Festlegen von Minimal- sowie Maximalkonzentrationen für die Pheromone soll verhindert werden, dass einige Kanten bei der Konstruktion einer Lösung gar nicht mehr in Betracht gezogen werden. Dieses Problem tritt auf, wenn eine Kante sehr selten verwendet wird. Die Pheromonkonzentration auf ihr strebt dann gegen 0 und die Kante wird nicht mehr verwendet. Eine ähnliche Situation ist gegeben, wenn es eine Kante gibt, die sehr häufig verwendet wird: die Pheromonkonzentration auf ihr steigt im Gegensatz zu den auf anderen stark an. Als Folge werden die Kanten mit geringerer Pheromonkonzentration nicht mehr oder kaum gewählt. Durch eine obere Grenze bezüglich der Pheromonkonzentration kann dem entgegengewirkt werden.

4.2.4. Pheromonaktualisierung

Eine Variante der Pheromonaktualisierung besteht darin, dass die Pheromone nicht in der Verwaltungsphase aktualisiert werden, sondern in der Konstruktionsphase durch die Ameisen selbst. Betritt eine Ameise eine Kante, so "verbraucht" sie einen Teil der dort abgelegten Pheromone, die Konzentration sinkt. Für nachfolgende Ameisen ist die gerade verwendete Kante dann weniger attraktiv und es werden andere Wege ausprobiert. Dadurch wird erreicht, dass die konstruierten Lösungen sich stärker voneinander unterscheiden und der Lösungsraum somit breiter durchsucht wird.

4.2.5. Lokale Suche

Durch die Anwendung eines Verfahrens, das auf lokaler Suche basiert, lassen sich die von den Ameisen berechneten Lösungen weiter verbessern. Die lokale Suche läuft so ab, dass Teile einer Lösung vertauscht werden, um dann zu prüfen, ob dadurch eine Verbesserung stattgefunden hat. Eine derartige Vertauschung könnte so aussehen, dass die Reihenfolge zweier aufeinanderfolgender Städte einer Rundreise vertauscht werden.

Ein Verfahren, das auf der lokalen Suche basiert, ist die Tabu-Suche.

4.2.6. Abbruchkriterien

Es gibt eine Reihe von Abbruchkriterien, die dafür sorgen, dass Ameisen-Systeme vorzeitig gestoppt werden. Dazu zählen:

- Die maximale Anzahl durchzuführender Zyklen ist erreicht.
- Alle Ameisen berechnen dieselbe Lösung, es liegt Stagnation vor.
- Die maximale Berechnungsdauer wird überschritten.
- Die maximale Kostengrenze für die Lösungen wird unterschritten.

4.3. Detaillierter Ablauf

Wir werden nun den Ablauf der Lösungsberechnung mit einem Ameisen-System in detaillierter, formalisierter Form betrachten. Durch die mathematische Beschreibungsweise wollen wir einerseits die Abläufe konkretisieren und mögliche Missverständnisse ausräumen, aber auch eine Grundlage für die nachfolgende Implementierung legen. Dabei werden wir auch aufzeigen, welche Parameter dem Ameisen-System für die Optimierung übergeben werden müssen. Inhaltlich wird sich in diesem Kapitel kein Unterschied zu (Kapitel 4.1, S. 14) ergeben, dies ist aber beabsichtigt, da wir davon ausgehen, dass durch das bereits vorhandene Wissen das Verständnis der formalen Erläuterungen leichter fällt.

Die Eingabe für ein Ameisen-System besteht aus einem Graphen $G = (V, E)$ und einer Reihe von Parametern, die (Tabelle 4.1, S. 18) entnommen werden können. G muss eine Kostenfunktion aufweisen, also kantenbewertet sein, da den Ameisen für ihre Arbeit die Maximierung bzw. Minimierung einer Kostenfunktion als Ziel gesetzt wird, die Kostenfunktion bezeichnen wir mit ϕ, den Wert der Kante (i, j) mit $\phi((i, j))$.

Parameter	Bedeutung
τ_0	Konzentration der Initialpheromone
α	Exponent zur Gewichtung der Pheromone
β	Exponent zur Gewichtung der Heuristik
ρ	Faktor für die Evaporation der Pheromone
n_{ant}	Anzahl zu verwendender Ameisen
n_{run}	Anzahl zu durchlaufender Zyklen

Tabelle 4.1.: Parameter für die Optimierung mit einem Ameisen-System

In der Initialisierungsphase werden n_{ant} Ameisen erzeugt, ins zuvor ausgewählte Nest gesetzt und auf allen Kanten des Graphen Initialpheromone der Konzentration τ_0 gelegt. Dann werden in n_{run} Zyklen jeweils Konstruktions- und Verwaltungsphase durchlaufen.

Während der Konstruktionsphase konstruiert jede Ameise einen Lösungskandidaten. Dazu führt sie solange Schritte aus, bis sie alle Knoten besucht hat. Ein Schritt besteht aus mehreren Teilen:

1. Den unbesuchten möglichen Folgeknoten wird nach (Gleichung 4.1, S. 19) eine Besuchswahrscheinlichkeit zugeordnet.
2. Anhand der Besuchswahrscheinlichkeiten wird zufallsbasiert ein Folgeknoten ausgewählt.
3. Die Ameise bewegt sich zum Folgeknoten und markiert ihn als besucht.

Befindet sich eine Ameise am Knoten i, so wird jedem unbesuchten möglichen Folgeknoten j eine Besuchswahrscheinlichkeit p_j zugeordnet. Dabei bezeichne $N(i)$ die Menge der unbesuchten Nachbarknoten von i, $\tau_{(i,j)}$ die Pheromonkonzentration auf der Kante von i nach j und $\eta_{(i,j)}$ den heuristischen Wert derselben Kante. Alle bereits besuchten Knoten und diejenigen, die nicht als Folgeknoten infrage kommen, erhalten eine Besuchswahrscheinlichkeit von 0.

$$p_j = \frac{\tau_{(i,j)}^{\alpha} \cdot \eta_{(i,j)}^{\beta}}{\sum\limits_{N(i)} \tau_{(i,j)}^{\alpha} \cdot \eta_{(i,j)}^{\beta}} \tag{4.1}$$

Die heuristische Information wird anhand der Kosten bestimmt, die durch die Benutzung einer Kante entstehen, also aus der Bewertungsfunktion der Kanten abgeleitet. Da der heuristische Wert umso besser, also größer sein soll, je kürzer die Kante ist, wird der reziproke Wert der Kantenbewertung hierfür verwendet. Es ergibt sich

$$\eta_{(i,j)} = \frac{1}{\phi((i,j))}. \tag{4.2}$$

Hat jede Ameise eine Lösung konstruiert, schließt sich die Verwaltungsphase an. Im ersten Schritt findet Evaporation statt, dabei verdampft auf jeder Kante ein Teil der dort abgelegten Pheromone. Der neue Pheromonwert ergibt sich aus

$$\tau_{(i,j)} \leftarrow (1-\rho) \cdot \tau_{(i,j)}. \tag{4.3}$$

Der Parameter ρ gibt an, welcher Anteil der Pheromone nach der Evaporation noch vorhanden sein soll. Danach legt jede Ameise auf den in ihrer Lösung benutzten Kanten neue Pheromone ab. Die dabei verwendete Konzentration ergibt sich aus den Kosten der konstruierten Lösung. Wenn c die Kosten der Lösung sind und auf der Kante (i,j) neue Pheromone gelegt werden sollen, dann ergibt sich die neue Konzentration zu

$$\tau_{(i,j)} \leftarrow \tau_{(i,j)} + \frac{1}{c}. \tag{4.4}$$

Nachdem die Verwaltungsphase abgeschlossen ist, schließen sich evtl. weitere Zyklen an. Während der Berechnung wird die beste gefundene Lösung stets aktuell gehalten und zum Schluss als Ergebnis ausgegeben. Dies geschieht, indem nach jedem Zyklus die Lösungskandidaten aller Ameisen betrachtet und die beste dieser Lösungen gespeichert wird, falls sie besser ist als die beste Lösung aus vorherigen Durchläufen.

4.4. Eignung

Ameisen-Systeme sind für die vorliegende Problemstellung der Tourenplanung sehr gut geeignet. Sie bieten einerseits die Möglichkeit, zu jedem beliebigen Zeitpunkt die bisher beste Lösung zu erfragen und stellen somit stets eine Handlungsgrundlage für die Praxis zur Verfügung. Andererseits laufen alle Schritte während der Berechnung in polynomieller Zeit ab, weshalb auch das Ameisen-System in polynomieller Zeit zu einem Endergebnis kommt, wie auch in [DS04] angemerkt wird. Es sei aber noch einmal daran erinnert, dass Ameisen-Systeme Approximationsverfahren sind, also nur eine Näherung bestimmen. Dies ist auch der Grund dafür, dass eine Lösung in Polynomialzeit gefunden werden kann.

Den dynamischen Anforderungen der Probleminstanzen können Ameisen-Systeme ebenfalls gerecht werden. Trifft ein neuer Auftrag ein, fällt ein Fahrzeug aus oder verschlechtern sich Wegkosten, so muss bis zur Verwaltungsphase gewartet werden, dann können die Änderungen sofort in die Problemstellung übernommen werden. Bei der Konstruktion der Lösung ist es für die Ameisen nicht von Bedeutung, ob weitere Aufträge hinzugekommen sind. Da stets mit der Menge noch zu besuchender Knoten gearbeitet wird und die Behandlung jedes Knotens vollkommen gleich abläuft, merkt die Ameise nicht einmal, dass sich am Problem etwas verändert hat. Modifizierte Wegkosten werden ebenfalls transparent übernommen, da sie während der Lösungskonstruktion in die Berechnung der Besuchswahrscheinlichkeiten einfließen, eine gesonderte Handhabung ist nicht notwendig. Ausgefallene Fahrzeuge werden in der Planung nicht weiter berücksichtigt, die Ameisen verteilen die dadurch betroffenen Aufträge auf die restlichen Fahrzeuge.

Wie wir sehen, erfüllen Ameisen-Systeme alle der Anforderungen, die wir in der Zielsetzung in (Kapitel 1.3, S. 3) formuliert haben. Sie sind in der Lage, in praxistauglicher Zeit Lösungen zu bestimmen und können gute Ergebnisse liefern, wie [DS04] in empirischen Untersuchungen nachgewiesen haben. Sogar der Problematik der dynamischen Veränderungen werden sie gerecht.

5. Entwurf und Implementierung

Der Entwurf und die Implementierung von DOT läuft in zwei Phasen ab. Da die von Dyonisys, TermiDe und IcedG zu verrichtende Arbeit sehr ähnlich ist und auf denselben Daten basiert, wird zunächst eine gemeinsame Basis in Teamarbeit von Christopher Blöcker, Timo Jürgens und Nicolas Woldt erstellt. Anschließend werden auf dieser Grundlage die DOT-Optimierer entwickelt.

Bei der Erläuterung von Entwurf und Implementierung nehmen wir bewusst keine Trennung vor, sondern behandeln beides gemeinsam. Zum einen wäre die Unterscheidung in Entwurf und Implementierung künstlich herbeigeführt, da Entwicklung und Entwurf in einem iterativen Prozess stattgefunden haben, bei dem mehrfaches Redesign eine wichtige Rolle gespielt hat. Einige Problematiken und Notwendigkeiten sind erst mit der Zeit klar geworden. Zum anderen kann so direkt aufgezeigt werden, wie die Umsetzung stattgefunden hat und es werden identisch strukturierte Kapitel vermieden. Zur Veranschaulichung verwenden wir UML-Diagramme und machen Gebrauch von Entwurfsmustern, die bei richtiger Anwendung das Entstehen modularer, wiederverwendbarer und erweiterbarer Software fördern. Dabei orientieren wir uns an [GHJV04] und kennzeichnen die verwendeten Entwurfsmuster im folgenden mit dem $\star$-Symbol, statt jedes Mal explizit auf [GHJV04] zu verweisen.

An dieser Stelle sei noch einmal darauf hingewiesen, dass das entstehende System branchenunabhängig sein soll, um so die Möglichkeit offen zu halten, es nicht nur in der Öl- und Gasbranche einzusetzen, sondern auch den Transport von z.B. Tiefkühlkost oder Blumen planen zu können.

5.1. Basis

In der Basis wird die Logik, die in Zusammenhang mit der Erstellung und Verwaltung von Liefertouren steht, umgesetzt. Durch die gemeinsame Entwicklung können sowohl die Zeit für die Entwicklung als auch der Wartungsaufwand von DOT reduziert werden. Die Validierung der Eingaben muss nicht durch die Optimierer, sondern kann in der Basis stattfinden. Werden Fehler entdeckt, so muss nicht jeder der Optimierer sondern nur die zugrundeliegende Basis angepasst werden. Dasselbe gilt für Änderungen, die an der Logik und Verarbeitung der Daten vorgenommen werden.

Wir zeigen zunächst die wesentlichen Bestandteile der Basis auf und heben dabei

die von Christopher Blöcker implementierte Funktionalität besonders hervor. Dazu zählen das Matching der Fahrzeugausrüstung mit den Kundenanforderungen (Kapitel 5.1.3, S. 24), die Verwaltung der Kontingente (Kapitel 5.1.5, S. 27), die Beladung der Fahrzeuge (Kapitel 5.1.6, S. 28) und die Entwicklung des Parsers (Kapitel 5.1.8, S. 31).

Danach folgt das Kernstück dieser Arbeit, der Entwurf und die Implementierung von Dyonisys, welches auf der gemeinsamen Basis aufbaut.

5.1.1. Fahrzeug

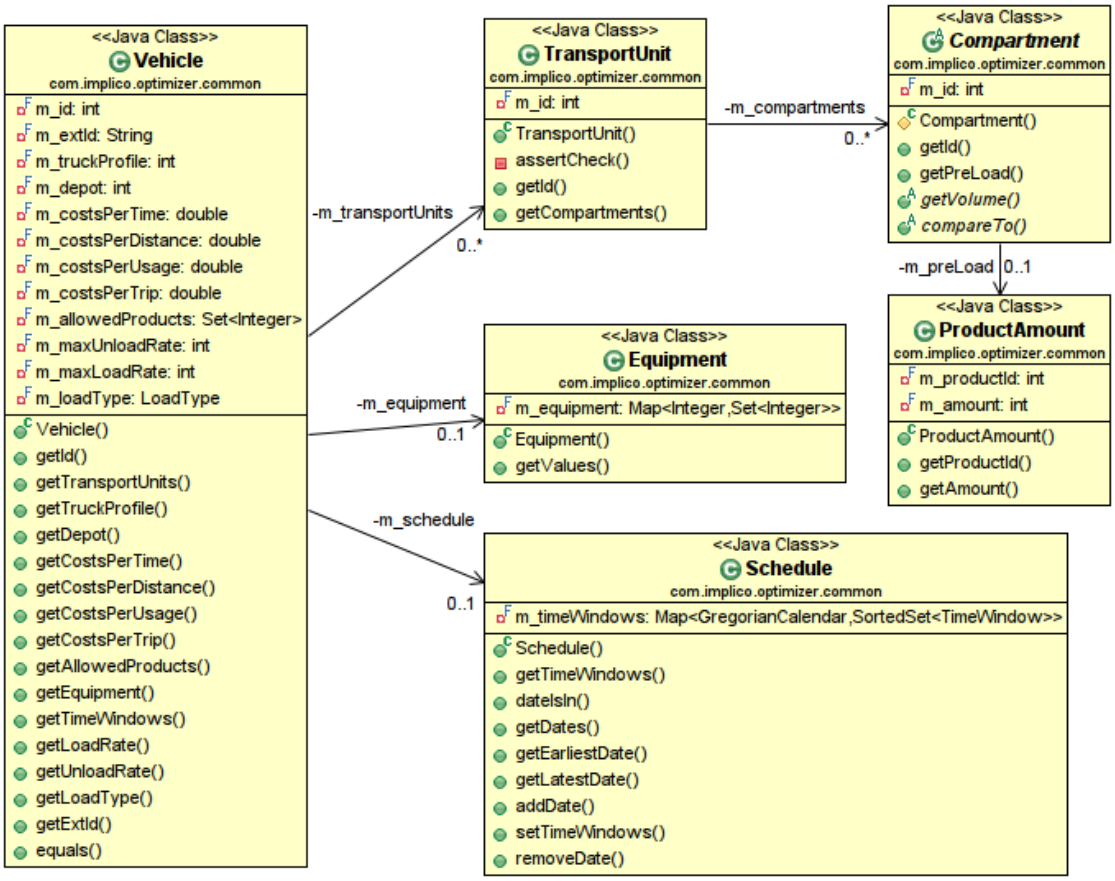


Abbildung 5.1.: UML-Diagramm zum Fahrzeug

Fahrzeuge werden für die Auslieferung von Waren an die Kunden verwendet. Ein Fahrzeug (Vehicle) verfügt über eine Reihe von Transporteinheiten (TransportUnit), die als Anhänger betrachtet werden können und jeweils über Transportkammern (Compartment) verfügen. Die Kapazität kann für jedes Fahrzeug unterschiedlich sein. In den Kammern werden die Produkte transportiert. Jedes Fahrzeug ist mit einer Ausrüstung (Equipment) ausgestattet, die benötigt wird, um Kunden beliefern zu können. Die Benutzung des Fahrzeugs ist nur innerhalb der im Zeitplan (Schedule) angegebenen Arbeitszeiten möglich.

Fahrzeuge verfügen über Informationen darüber, wie teuer ihr Einsatz ist, dabei wird unterschieden in z.B. Kosten pro Strecke (m_costsPerDistance) oder pro Zeit (m_costsPerTime). Außerdem sind ihnen ein Heimatdepot und ein Truckprofil in Form einer Identifikationsnummer zugeordnet. Über das Truckprofil kann festgestellt werden, wie schnell ein Fahrzeug auf welcher Art von Straßen verkehren darf und es dient zum Abrufen der Routing-Informationen vom xServer[1]. Anhand der maximalen Be- und Entladerate des Fahrzeugs kann festgestellt werden, wie schnell Betankungen und Lieferungen durchgeführt werden können.

Die Kammern der Transporteinheiten können über eine Anfangsbeladung verfügen, die über eine Produktmenge (ProductAmount) abgebildet wird. Die Festlegung, welche Produkte in einem Fahrzeug transportiert werden dürfen, findet über eine Whitelist statt, die durch eine Menge (m_allowedProducts) umgesetzt wird.

Die Branchenunabhängigkeit wird hier dadurch erreicht, dass ein Fahrzeug mit genau der Ausprägung von TransportUnit ausgestattet werden kann, die für den jeweiligen Anwendungsfall erforderlich ist.

5.1.2. Fahrzeugdepot, Ladedepot und Lieferung

Fahrzeugdepots (TruckDepot), Ladedepots (LoadDepot) und Lieferungen (Delivery) werden als Orte (Location) modelliert. Dadurch können über Vererbung gemeinsame Funktionalität und Datenfelder bereits in der beerbten Klasse Location implementiert werden. In TruckDepot, LoadDepot und Delivery werden weitere spezifische Attribute und Methoden ergänzt.

LoadDepot und Delivery verfügen über Anforderungen (Requirement), die ein Fahrzeug erfüllen muss, um sie anfahren zu können[2], sie können mit der Methode getRequirements erfragt werden. Außerdem ist stets in einem Zeitplan (Schedule) angegeben, zu welchen Zeiten die Orte "geöffnet" sind. Über Methoden wie getServiceTime und getWaitingTime kann bestimmt werden, wie lange ein Fahrzeug an einem Ort verweilt.

Jede Delivery hat eine Menge von Bestellungen (DeliveryPosition), die durch die Fahrzeuge zu liefern sind. Es werden jeweils das geforderte Produkt und die dazugehörige Menge in einer DeliveryPosition gespeichert. DeliveryPosition und ProductAmount sind zwar strukturell identisch, jedoch wird aufgrund von möglichen zukünftigen Erweiterungen und aufgrund der logistischen Betrachtungsweise diese Unterscheidung getroffen.

Ein LoadDepot verfügt über eine Menge von Kontingenten (Contingent), in denen verfügbare Warenmengen verwaltet werden. Dasselbe Kontingent kann an mehreren

[1] siehe (Kapitel 2.6, S. 6)
[2] mehr dazu in (Kapitel 5.1.3, S. 24)

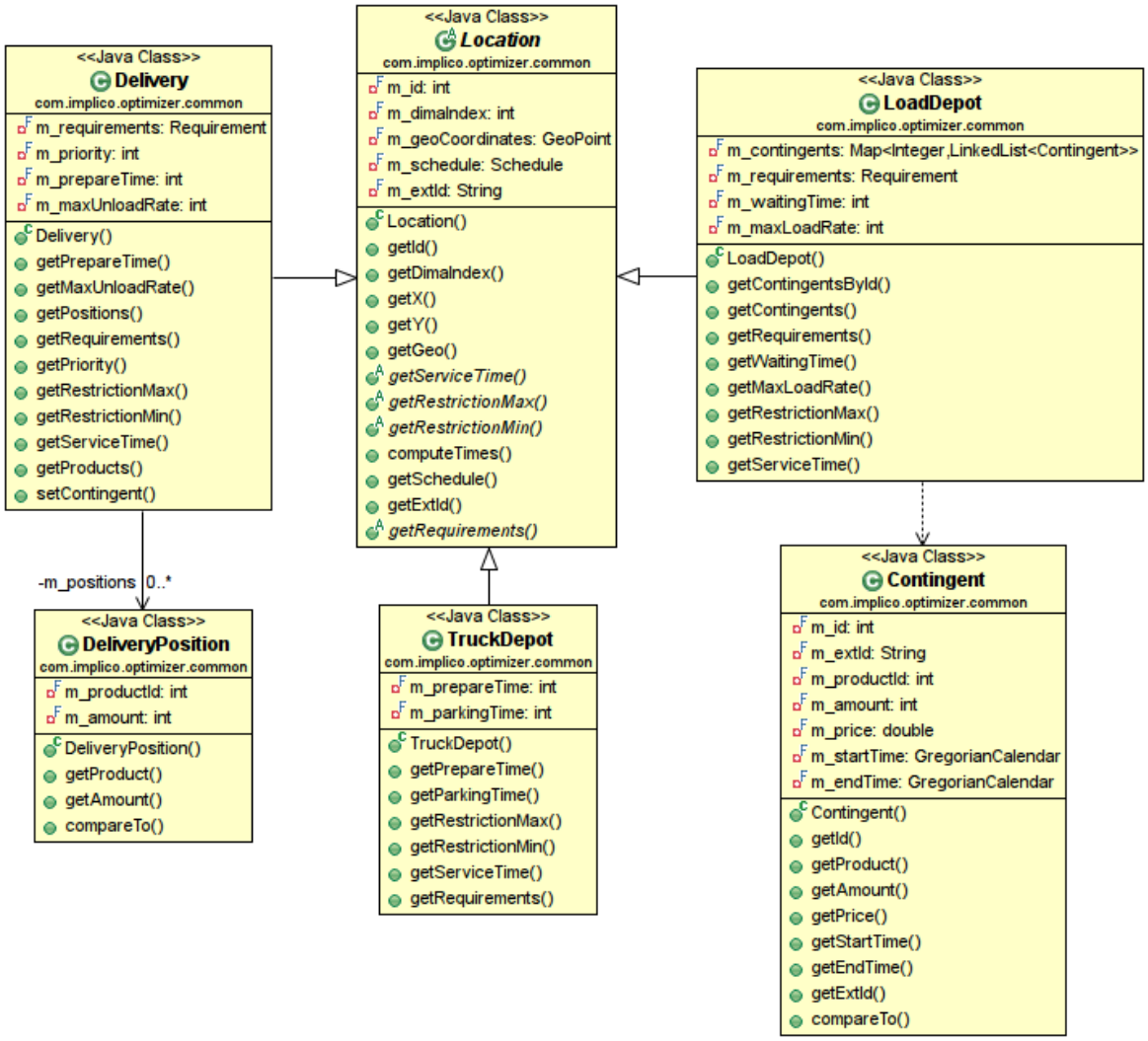


Abbildung 5.2.: UML-Diagramm zu den Orten

Depots vorkommen, sodass die Betankung an einem Depot auch Kontingente an anderen Depots beeinflussen kann. Die maximale Beladerate m_maxLoadRate gibt an, wie schnell Fahrzeuge am jeweiligen Depot beladen werden können. Für die Berechnung der Beladedauer wird dabei das Minimum von maximaler Beladerate vom LoadDepot und maximaler Beladerate vom Fahrzeug gebildet.

5.1.3. Anforderungen und Ausrüstung

Wie wir bereits gesehen haben, verfügt jedes Fahrzeug über eine Ausrüstung (Equipment) und jede Location über Anforderungen (Requirement).

Um das Equipment abzubilden, wird jedem Ausrüstungsgegenstand eine Identifikationsnummer zugewiesen. Dem Programm ist nicht bekannt, welcher Ausrüstungsgegenstand sich hinter einer Nummer verbirgt. In eine Tabelle (Map) wird eingetragen,

zu welchem Ausrüstungsgegenstand (also zu welcher Identifikationsnummer) welche Ausprägung vorhanden ist. Verfügt ein Fahrzeug z.B. über Schlauchlängen von 5m und 10m und ist 1 die Identifikationsnummer für die Schlauchlänge, so werden in die Tabelle für den Schlüssel 1 die Werte 5 und 10 eingetragen, siehe (Tabelle 5.1, S. 25).

Ausrüstung	Ausprägung
1	$\{5, 10\}$

Tabelle 5.1.: Beispielausrüstung eines Fahrzeugs mit den Schlauchlängen 5m und 10m

Anforderungen (Requirement) bestehen aus einer Menge von Einträgen (RequirementEntry), die in ihrer Gesamtheit definieren, welches Equipment vonnöten ist, um sie zu erfüllen. Zu jedem RequirementEntry sind eine Identifikationsnummer, eine Menge von Werten und ein EquipmentOperator gegeben. Die Identifikationsnummer gibt an, welchem Ausrüstungsgegenstand diese Anforderung entspricht. Die Werte geben an, welche Ausprägung seitens des Equipment verfügbar sein muss und der EquipmentOperator gibt an, welche Relation zwischen Ausrüstungsgegenstand und Anforderung bestehen soll. So lässt sich z.B. unter Verwendung des Operators EquipmentOperatorGreaterOrEqual ausdrücken, dass eine Mindestschlauchlänge gefordert ist, siehe (Tabelle 5.2, S. 25).

Anforderung	Ausprägung	Operator
1	$\{5\}$	$\geq$

Tabelle 5.2.: Anforderung an die Schlauchlänge: Sie soll mindestens 5m betragen.

Die Erzeugung der EquipmentOperatoren kann nicht direkt stattfinden, da die Konstruktoren versteckt sind, sondern muss über die EquipmentOperatorFactory stattfinden, die als *Singleton** implementiert ist. Die Operatoren selbst sind *Fliegengewichte**, da sie keinen intrinsischen Zustand aufweisen. Das Interface EquipmentOperator gibt vor, welche Funktionalität, nämlich das Matchen von Requirement und Equipment, durch die Operatoren zu implementieren ist. Die Operatoren können transparent gegeneinander ausgetaucht werden und somit unterschiedliche Funktionalität zur Verfügung stellen, es liegt eine *Strategie** vor.

Das Abgleichen von Anforderungen und Ausrüstung wird als *Matching* bezeichnet und liefert als Ergebnis die entstehenden Strafkosten, die durch die Auslieferung mit dem jeweiligen Fahrzeug entstehen. Beim Matching werden die Anforderungen nacheinander überprüft, dabei werden jeweils die Einträge der Ausrüstung, die Anforderung und der jeweilige Operator herangezogen. Für weiche Restriktionen werden die durch die Ausrüstung verursachten Strafkosten aufsummiert. Wird eine harte Restriktion verletzt, bricht das Matching ab. Das Ergebnis wird in Form eines Objektes der Klasse Match geliefert. Über das Feld m_matches kann geprüft werden, ob die Anforderungen erfüllt werden können, m_costs gibt die Höhe der Strafkosten an.

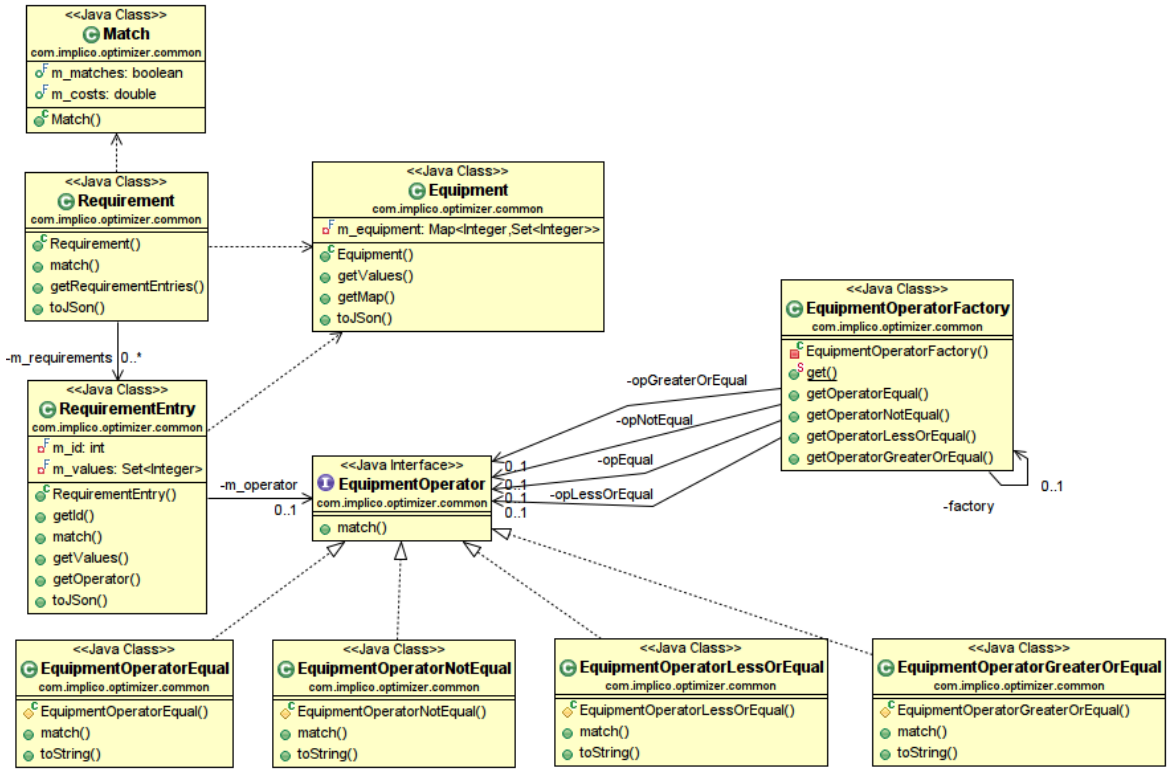


Abbildung 5.3.: UML-Diagramm zu den Anforderungen und der Fahrzeugausrüstung

5.1.4. Produkte

Bei den Produkten (Product) handelt es sich um den Gegenstand, dessen Auslieferung zu planen ist. Jedem Produkt ist eine eindeutige Identifikationsnummer sowie ein externer Identifikator, der durch ATOS erforderlich ist, zugeordnet.

Produkte können aus mehreren anderen Produkten zusammengesetzt sein. In einer Tabelle (m_baseProducts) wird eingetragen, welche Mengen anderer Produkte benötigt werden, um eine Einheit des betroffenen Produktes herzustellen. Dabei wird davon ausgegangen, dass ein zusammengesetztes Produkt nur aus Basisprodukten, also aus solchen, die selbst nicht zusammengesetzt sind, besteht.

Zu jedem Produkt gibt es eine Menge von Produkten (m_allowedProducts), die gemeinsam mit ihm transportiert werden dürfen, die Produkte werden dabei anhand ihrer Identifikationsnummern in einer Whitelist gespeichert.

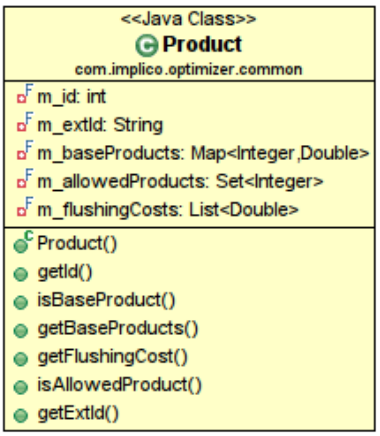


Abbildung 5.4.: UML-Diagramm zur Struktur der Produkte

Bei der Beladung eines Fahrzeugs kann es vorkommen, dass Kammern aufgrund einer vorherigen Beladung zunächst gereinigt werden müssen. Abhängig davon, welches Produkt eingeladen werden soll und welches Produkt zuvor geladen war, fallen dabei Kosten unterschiedlicher Höhe an. Je Produkt werden in einer Liste Informationen darüber verwaltet, wie teuer die Reinigung in Abhängigkeit zuvor geladener Produkte ist.

5.1.5. Kontrakt und Kontingente

Die Verwaltung sämtlicher Kontingente (Contingent), also der Verfügbaren Warenmengen an den Depots wird im Kontrakt (Contract) zusammengefasst. Da die Kontingente tagesübergreifend sein können, sich also ihre Verfügbarkeit über mehrere Tage erstrecken kann, gilt dasselbe für den Kontrakt. Der Kontrakt ist stets dem Plan zugeordnet.

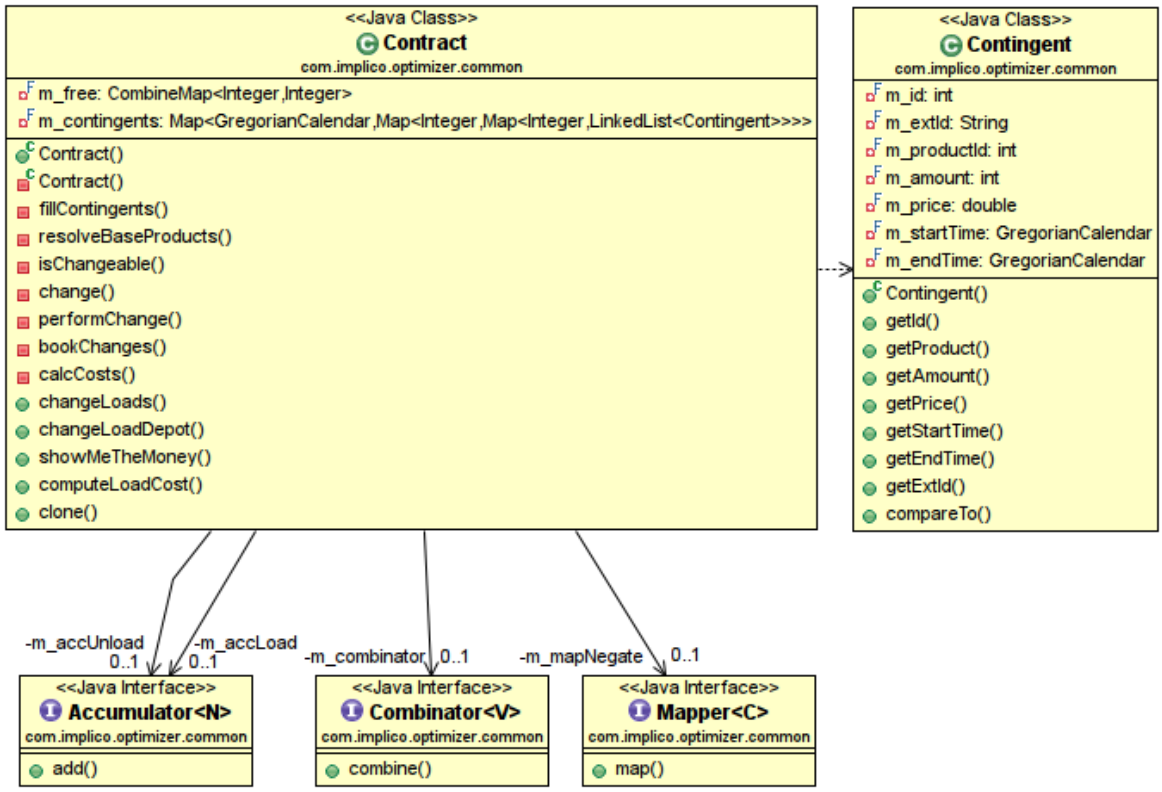


Abbildung 5.5.: UML-Diagramm zum Kontrakt und den Kontingenten

Soll die Beladung eines Fahrzeugs an einem bestimmten Depot stattfinden, so muss dies über den Kontrakt geschehen. Es werden dann die am angegebenen Depot verfügbaren Kontingente in einen Cache (m_contingents) geladen, die Auswertung findet also nur bei Bedarf (*lazy*) statt. Dann wird geprüft, ob die geforderten Warenmengen unter

Verwendung der verfügbaren Kontingente bereitgestellt werden können. Die Kontingente werden dafür aufsteigend nach ihrem Preis pro Liter sortiert. Es wird sequenziell nach dem ersten passenden Kontingent gesucht, mit dem die Beladung durchgeführt werden kann, hierbei kann es vorkommen, dass aus mehreren Kontingenten jeweils Teilmengen verwendet werden müssen. In der Tabelle m_free wird eingetragen, welche Restmengen nach der Beladung in den jeweiligen Kontingenten noch verfügbar sind.

Die Implementierung von Contract ist threadsicher, d.h. dass bei der Arbeit mit mehreren Threads stets ein gültiger Zustand garantiert wird. Dies wird dadurch erreicht, dass sämtliche öffentlichen Methoden synchronisiert sind und somit immer nur ein Thread zur Zeit Zugriff auf den Contract haben kann.

Bei den Interfaces Accumulator, Combinator und Mapper handelt es sich um die Nachbildung anonymer Funktionen, inspiriert durch die funktionale Programmierung. Algorithmen zum Akkumulieren und zum Kombinieren von Werten wurden in Form von Objekten definiert, die einer gegebenen Schnittstelle gehorchen, es liegen hier $Befehle^{\star}$ vor. Der Mapper wird verwendet, um auf jedes Element einer Java Collection dieselbe Funktion anzuwenden und aus den daraus entstehenden Werten eine neue Collection zu erzeugen. Ein Combinator kapselt eine Funktion zum Verknüpfen zweier beliebiger, aber gleichtypiger Werte zu einem und ein Accumulator akkumuliert eine Reihe gleichtypiger, numerischer Werte nach einem durch ihn festgelegten Verfahren.

5.1.6. Beladung

Die Beladung der Fahrzeuge (Load) findet pro Trip statt und kann nach unterschiedlichen Verfahren erfolgen. Gemeinsam ist den möglichen Fällen, dass eine Menge von eingeplanten Aufträgen (m_deliveries) verwaltet wird. Über eine Whitelist (m_allowedProducts) wird festgelegt, welche Produkte transportiert werden dürfen, diese Information wird aus dem für den Trip verwendeten Fahrzeug ausgelesen. Jede Beladung verfügt über Informationen darüber, welche Produktrestmengen (m_initial) zum Beladungszeitpunkt noch in den Kammern vorhanden sind und welche Beladung (m_load) nach dem Tanken am Depot m_loadDepot vorhanden sein wird. Über die Methoden add und remove können Aufträge, und somit Produktmengen, zu der Beladung hinzugefügt bzw. von aus ihr entfernt werden. Die Methode fits prüft, ob das Hinzufügen eines Auftrags überhaupt noch möglich ist und mit der Invarianten inv kann geprüft werden, ob der Ladezustand gültig ist.

Liegt der Fall der Vorbeladung (LoadPreload) vor, so ist von vornherein festgelegt, welche Produkte in welche Kammern einzuladen sind. Bei jedem Nachtanken muss dieser Lade-Zustand wiederhergestellt werden. Die Belieferung der Aufträge findet dann in Abhängigkeit davon statt, welche Produkte geladen wurden. Die Begründung für die Verwendung dieser Beladeart können z.B. betriebliche Vorgaben sein.

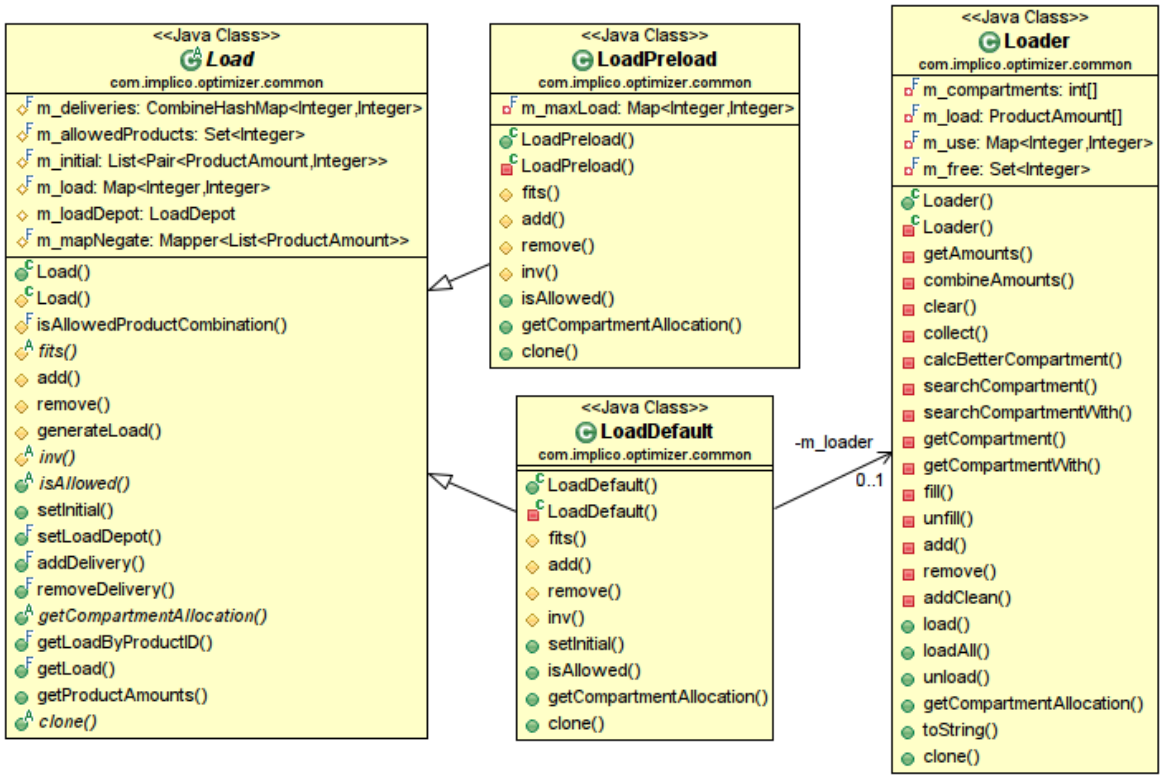


Abbildung 5.6.: UML-Diagramm zur Beladung der Fahrzeuge

Beim Standard-Beladeverfahren (LoadDefault) wird abhängig von den zu beliefernden Aufträgen die Beladung bestimmt. Die tatsächliche Arbeit wird dabei an den Loader delegiert, der die Einplanung der Produktmengen auf Kammer-Ebene durchführt.

Dem Loader werden die einzuplanenden Produktmengen übergeben und er versucht, sie nach dem *Best Fit* Prinzip einzuplanen. Dabei werden die einzuplanenden Produktmengen absteigend sortiert und in die am besten passende Kammer eingeplant, es wird mit der größten Menge begonnen. Als beste Kammer wird stets zunächst diejenige gewählt, die bereits das jeweilige Produkt geladen hat, aber noch nicht voll ist. Gibt es keine solche Kammer, wird diejenige gewählt, die die Menge komplett aufnehmen kann und nach der Beladung die kleinste freie Restmenge hat. Ist die Produktmenge für jede Kammer zu groß, so wird die größte Kammer ausgewählt. Der Ablauf des Verfahrens ist wie folgt:

1. Wenn es eine Kammer gibt, die das jeweilige Produkt bereits enthält und noch freien Platz bietet, wird hier so viel wie möglich eingeladen.
2. Es wird solange die am besten passende Kammer ausgewählt und so viel wie möglich eingeladen bis alles eingeladen wurde oder es keine freien Kammern mehr gibt.
3. Wenn alle Mengen erfolgreich eingeladen wurden, dann stoppt das Verfahren mit Erfolg. Wenn es aber Mengen gibt, die nicht eingeladen werden konnten, dann

wird die Beladung hergenommen und mit den zu ladenden Produktmengen zusammengefasst. Als nächstes wird die gesamte Beladung entfernt und es wird versucht, die zusammengefassten Mengen in die nun freien Kammern einzuplanen, dies erfolgt ebenfalls absteigend nach den Mengen sortiert. Scheitert dies auch, so können die zusätzlichen Mengen nicht eingeplant werden. Das Verfahren weist die Mengen zurück und stellt den vorherigen Beladezustand wieder her.

Die hier aufgezeigten Beladeverfahren, LoadPreload und LoadDefault, sind bereits auf die Anwendung in der Öl- und Gasbranche ausgelegt und somit nicht branchenunabhängig. Im Bezug auf die Beladung wird es auch nicht möglich sein, eine allumfassende Lösung zu bieten, da es in jeder Branche Spezialanforderungen gibt. Um das System dennoch für andere Branchen verwenden zu können, wäre jeweils eine branchenspezifische Implementierung der Load-Klasse erforderlich.

5.1.7. Lieferplan

Der Lieferplan (Plan) stellt das Ergebnis der Berechnungen dar und wird sukzessive aufgebaut. Er besteht aus mehreren Touren (Tour), die jeweils von genau einem Fahrzeug gefahren werden. Jedes Fahrzeug startet seine Tour an dem ihm zugeordneten Fahrzeugdepot (TruckDepot) und beendet sie auch dort. Eine Tour setzt sich aus mehreren Trips (Trip) zusammen, die aus einer Reihe von Stops (Stop) bestehen. Jeder Trip beginnt mit einem Stop zur Beladung (StopLoad), gefolgt von einer Reihe von Lieferungen (StopDelivery).

Die Erzeugung des Plans sowie der Touren und Trips geschieht über die jeweilige Implementierung der *Abstrakten Fabrik*[⋆] Factory, die für jeden der Optimierer zu erstellen ist. Beim Hinzufügen einer neuen Tour zum Plan mittels add bzw. eines neuen Trips zur Tour mittels appendNewTrip werden im Hintergrund die Methoden createTour bzw. createTrip der Fabrik aufgerufen. Durch das Bereitstellen dieser Schnittstelle und Verstecken der Konstruktoren von Plan, Tour, Trip und der Stops wird verhindert, dass diese Klassen direkt instanziiert werden. So wird die Möglichkeit der Fehlbenutzung minimiert und ein höheres Abstraktionsniveau bei der Umsetzung der Optimierer erreicht.

Um dem Plan eine neue Tour hinzuzufügen, muss angegeben werden, welches Fahrzeug dazu zu verwenden ist. Die Tour wird dann anhand der Identifikationsnummer des Fahrzeugs verwaltet, so ist ein unmittelbarer Bezug zwischen Fahrzeug und Tour hergestellt. Das Erstellen eines neuen Trips läuft über die Methode appendNewTrip von Tour ab, die im Hintergrund die Factory aufruft. Wird ein Stop an einen Trip angehängt, so werden sofort die dadurch entstehenden Kosten aktualisiert, die Beladung mit Hilfe des Loaders durchgeführt und die Änderung der Warenmengen am Kontrakt gebucht.

Die Funktionalität des Plans und der zugehörigen Klassen, die die Verwaltung der

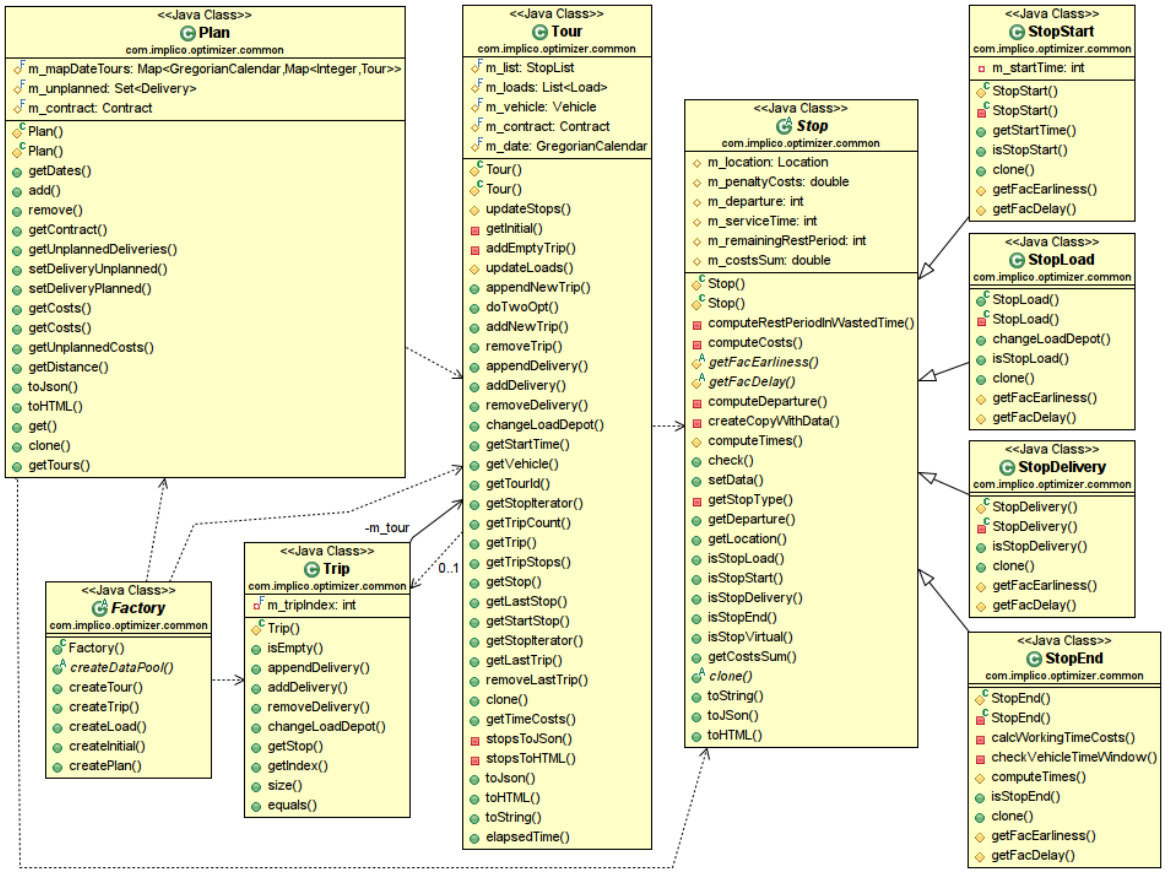


Abbildung 5.7.: UML-Diagramm zum Plan

Liefertouren bieten, ist so implementiert, dass niemals ein ungültiger Zustand entstehen kann. Würde das Einfügen eines weiteren Stops harte Bedingungen verletzen, so wird die Aktion verweigert. Zu jeden Zeitpunkt wird gewährleistet, dass der Plan in einem konsistenten Zustand und gültig ist.

5.1.8. Parser

Die Eingaben, die aus dem SAP-System über ATOS an DOT übergeben werden, sind in Form von JSON codiert und werden mit Hilfe eines Parsers verarbeitet. Die Syntax, der die Eingaben für den Parser entsprechen müssen, kann der in (Anhang A, S. 61) definierten Grammatik entnommen werden. Die Verwendung von JSON begründet sich damit, dass ein einheitliches Kommunikationsformat verwendet werden soll und seitens des SAP-Systems auch ein JSON-Parser zur Verfügung steht. Außerdem bietet JSON im Vergleich zu XML, welches als Alternative in Betracht gezogen wurde, den Vorteil, dass es durch den Menschen wesentlich leichter zu lesen ist und ein weitaus geringeres Datenvolumen erzeugt.

Wenn eine Eingabe von ATOS an DOT übergeben wird, analysiert zunächst der

Scanner die Eingabe und zerlegt sie in Token. Der Scanner ist nicht von Hand implementiert, sondern mit jflex generiert worden. So kann eine spätere Anpassung von Token oder die Erweiterung der Sprachdefinition schnell und unkompliziert erfolgen.

Der durch den Scanner aus der Eingabe generierte Token-Stream wird als nächstes vom Parser[3] verarbeitet. Der Parser ist direkt aus der Definition der Eingabesprache entwickelt worden. Bereits bei der Definition der Eingabesprache wurde darauf geachtet, eine LL(0) Sprache zu konstruieren, um so eine leichte Umsetzung des Parsers zu ermöglichen.

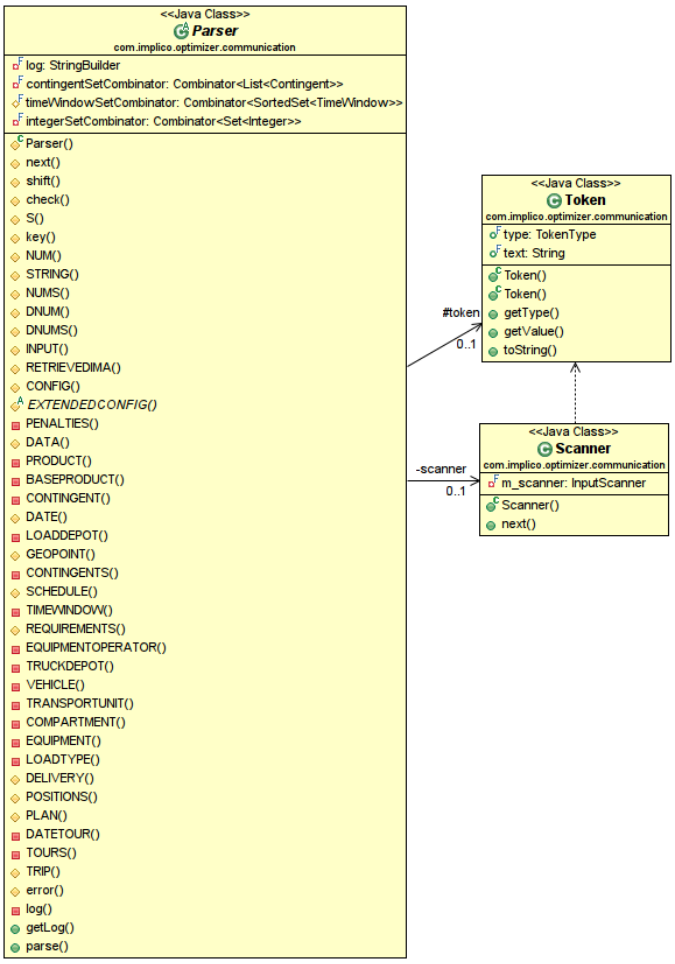


Abbildung 5.8.: UML-Diagramm zum Parser

Der Parser baut direkt beim Verarbeiten der Eingabe alle für die Arbeit mit den Daten nötigen Datenstrukturen auf. Da es Abhängigkeiten zwischen einzelnen Teilen der Eingabe gibt, ist die in (Anhang A, S. 61) festgelegte Sequenz der Eingabe signifikant. Für jeden syntaktischen Typ weist der Parser eine eigene Erkennunsprozedur auf, so werden z.B. Fahrzeuge mit der Methode VEHICLE verarbeitet.

Die Struktur der Eingabedaten kann in drei voneinander abhängige Teile unterschieden werden:

1. **Konfiguration**
   Die Konfiguration enthält Parameter für die spätere Optimierung. Es werden beispielsweise die Höhe von Strafkosten, maximale Arbeitsdauer der Fahrer sowie die Anzahl von Fahrzeugen, Lieferungen etc. des Szenarios festgelegt. Eine vollständige Auflistung der Parameter für die Basis und für Dyonisys ist in (Anhang B, S. 64) zu finden.

[3]Genau genommen handelt es sich beim Parser um einen *Interpretierer**. Prinzipiell hätte der Parser aus der JSON-Definition generiert werden können, jedoch wäre dann auch das Entwickeln weiterer Zwischenstrukturen nötig und die direkte Nutzung als Interpretierer nicht möglich gewesen.

2. **Daten**
   Die Daten zur Optimierung umfassen die Informationen über die Produkte, Kontingente, Ladedepots, Fahrzeugdepots, Fahrzeuge und Lieferungen. Dabei ist u.a. enthalten, an welchen Geokoordinaten sich die Orte befinden. Sind die Daten fertig gelesen und verarbeitet, wird die Distanzmatrix für die eingegebenen Orte vom xServer abgerufen.

3. **Plan**
   Es kann ein Ausgangsplan vorliegen, der als Grundlage für die Berechnungen bei der Optimierung dient. Der Plan *darf* jedoch leer sein.

5.1.9. Datenbasis und Konfiguration

Die vom Parser eingelesenen Daten werden in der Datenbasis (DataPool) und der Konfiguration (Config) verwaltet, die beide abstrakt sind, da sie spezifisch für die jeweiligen Optimierer angepasst werden müssen.

Während bei IcedG zur Laufzeit keine Veränderungen der Daten auftreten, die Anzahl der Fahrzeuge und Aufträge etc. also immer gleich bleibt, ist es für TermiDe und Dyonisys essenziell, dass z.B. neue Aufträge hinzugefügt werden können. Daher kann keine Datenstruktur gewählt werden, die für alle Optimierer gleichermaßen geeignet ist, sie muss individuell durch Beerben von DataPool festgelegt werden. Aus diesem Grund müssen auch die Zugriffsfunktionen auf die Datenstrukturen abstrakt sein und jeweils abhängig von der gewählten Datenstruktur implementiert werden.

Ein ähnliches Bild zeigt sich bei der Konfiguration. Zwar kann hier eine große Obermenge für alle DOT-Optimierer gemeinsam implementiert werden, da aber für jedes der umgesetzten

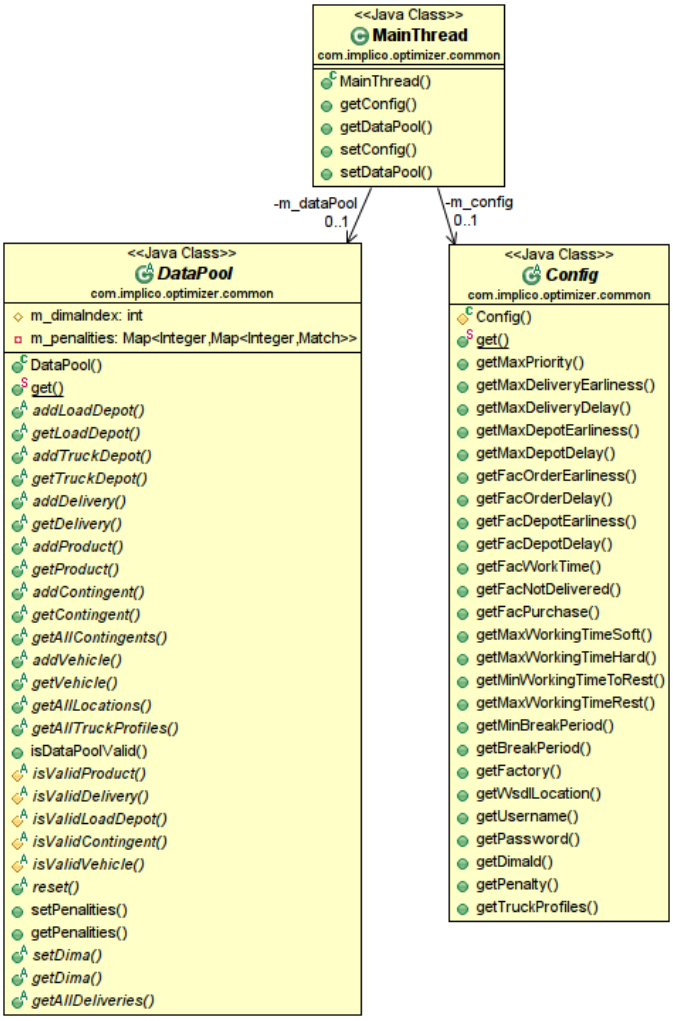


Abbildung 5.9.: UML-Diagramm zur Datenbasis und der Konfiguration

Verfahren[4] individuelle Konfigurationsparameter angegeben werden müssen, muss die Konfiguration von allen Optimierern beerbt und angepasst werden.

Die Datenbasis sowie die Konfiguration werden als statische Variablen in der Klasse MainThread verwaltet und stehen somit jeder Instanz von MainThread zur Verfügung, auch den Instanzen erbender Klassen. Die Verwaltung als statische Variable hat zur Folge, dass Datenbasis und Konfiguration selbst dann bestehen bleiben, wenn es keine Instanz von MainThread mehr gibt, dies ist der Arbeitsweise der JVM zu verdanken. Erst bei Terminierung der JVM oder bei Neueingabe von Datenbasis und Konfiguration gehen diese verloren.

5.2. Dyonisys

Dyonisys baut auf den in (Kapitel 5.1, S. 21) beschriebenen Strukturen auf und erweitert sie teilweise. So ist z.B. die Erweiterung des Parsers aufgrund weiterer Konfigurationsparameter für das Ameisen-System unbedingt notwendig. Ebenso müssen die Strukturen zur Datenverwaltung angepasst werden, da Dyonisys auf dynamische Veränderungen reagieren soll und somit eine feste Anzahl von Fahrzeugen oder Aufträgen in Widerspruch hierzu stünde.

Bereitgestellt wird Dyonisys als Webservice, der auf einem Tomcat[5] Webserver gehostet wird [Tomcat]. Ein Servlet[6] nimmt Anfragen entgegen, die über ATOS gemacht werden und erzeugt die zur Verarbeitung benötigten Objekte, veranlasst die Berechnung der Lösung und gibt das Ergebnis über ATOS wieder zurück.

Sobald eine Anfrage eintrifft, wird dabei ein OptimizerThread erzeugt, der die weitere Verarbeitung vornimmt. Da der OptimizerThread von MainThread erbt, ist hier Zugriff auf Datenbasis und Konfiguration gegeben. Es wird ein Parser erzeugt, der die eingegebenen Daten verarbeitet und die Konfiguration und Datenbasis initialisiert. Danach wird ein Optimizer instanziiert, der als $Proxy^{\star}$ implementiert ist und die Erzeugung des Ameisen-Systems übernimmt. Das Ameisen-System erzeugt die Ameisen und die Ausführungsthreads (AntRunner) und berechnet eine Lösung zum gegebenen Problem. Zum Schluss wird die Lösung über den Optimizer an den OptimizerThread geliefert und über ATOS zurückgegeben.

In den folgenden Unterkapiteln werden wir die eben kurz umrissenen Abläufe, die Dyonisys während der Lösungsberechnung durchführt, im Detail betrachten. Dabei

[4] siehe (Kapitel 2.2, S. 4), (Kapitel 2.3, S. 5) und (Kapitel 2.4, S. 5).

[5] Für Informationen über Tomcat sei auf [Tomcat] verwiesen.

[6] Bei einem Servlet handelt es sich um eine Java-Klasse, die durch einen Webserver zur Verfügung gestellt wird und seine Funktionalität erweitert. Servlets ermöglichen, wie auch z.B. PHP und Ruby, die Umsetzung dynamischer Webseiten. Das Wort *Servlet* ist zusammengesetzt aus den Wörtern Server und Applet. *Applet* wiederum setzt sich zusammen aus Application und Snippet. Ein *Servlet* ist also ein serverseitig ausgeführter Anwendungsschnipsel.

werden wir auch die verwendeten Algorithmen näher beleuchten und erklären. Unsere Erklärungen werden wir dabei auf (Abbildung 5.10, S. 35) beziehen.

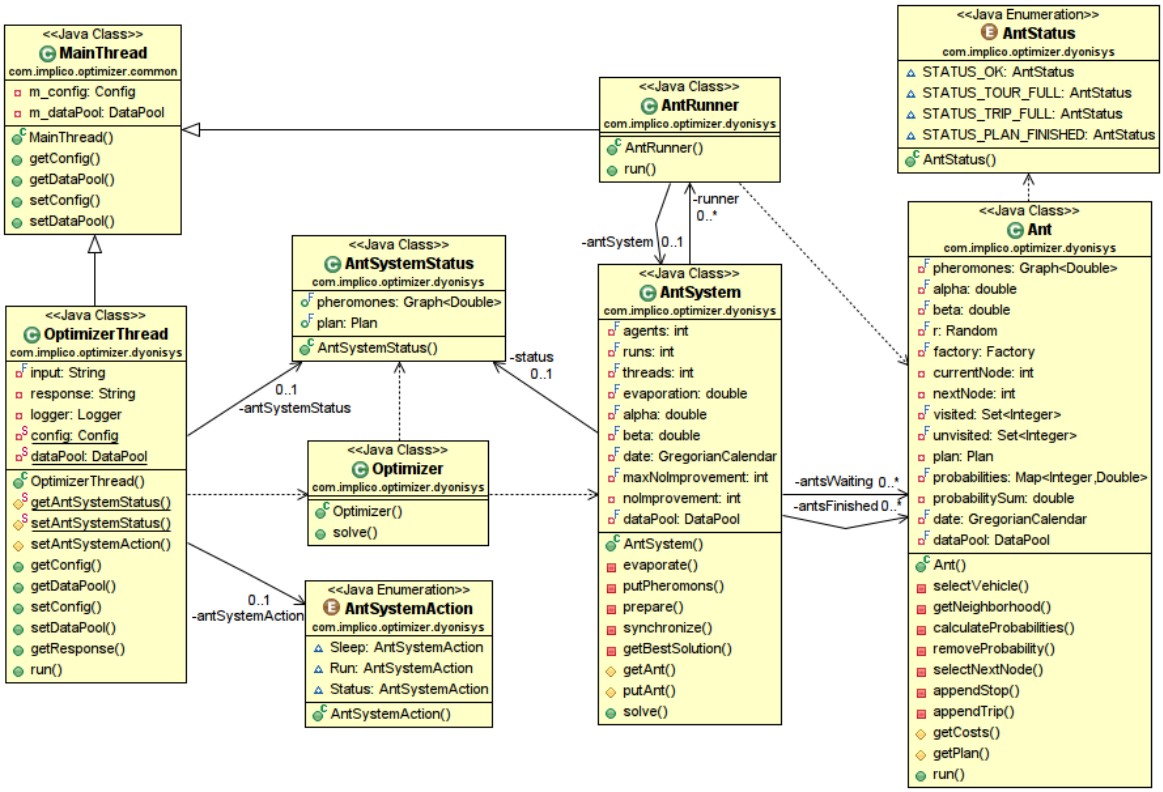


Abbildung 5.10.: UML-Diagramm zur Klassenübersicht von Dyonisys

5.2.1. Kommunikation

Um einen Überblick über die stattfindende Kommunikation zu geben, die im Rahmen der Optimierung abläuft, abstrahieren wir von den tatsächlich implementierten Klassen und konzentrieren uns auf den logischen Ablauf.

Im Wesentlichen besteht Dyonisys aus den in (Abbildung 5.11, S. 35) gezeigten Komponenten, zwischen denen Informationen übertragen werden. Auf jede stattgefundene Datenübertragung folgt eine kontextbezogene Reaktion durch den Empfänger.

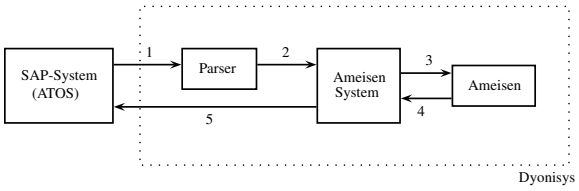


Abbildung 5.11.: Bestandteile von Dyonisys und Kommunikation über ATOS

1. Der Parser erhält seine Eingabe vom SAP-System über ATOS in JSON-Notation. Er erzeugt die notwendigen Datenstrukturen und füllt sie mit den erhaltenen Da-

ten. Hierzu zählen die Konfiguration, die Datenbasis, das Ameisen-System sowie der Eingabegraph für das Ameisen-System.

2. Das Ameisen-System erhält vom Parser die Eingabedaten und bereitet die Optimierung vor. Dabei werden die Ameisen erzeugt und der Status des Ameisen-Systems initialisiert. Zum Status zählen die Lösung, die initial leer ist, sowie der Problemgraph mit den zugehörigen Pheromonkonzentrationen.
3. Der Optimierungslauf wird gestartet und jede der Ameisen bestimmt einen Lösungskandidaten.
4. Das Ameisen-System fragt die Lösungskandidaten von den Ameisen ab und bestimmt daraus die beste gefundene Lösung. Sie wird in den Status des Ameisen-Systems übernommen. Der Ablauf von 3. und 4. wird solange wiederholt, bis die maximale Anzahl der Zyklen erreicht ist, Stagnation eintritt oder ein anderes Abbruchkriterium zutrifft.
5. Das Ergebnis wird in JSON codiert und über ATOS an das SAP-System geliefert.

5.2.2. Optimierungs-Thread

Ein Optimierungs-Thread (OptimizerThread) wird immer dann erzeugt, wenn das Servlet, das die Funktionalität von Dyonisys zur Verfügung stellt, über ATOS angesprochen wird. Dem Optimierungs-Thread werden als Eingabe die Daten übergeben, die über das Servlet getätigt wurden. Da OptimizerThread die Klasse MainThread beerbt, besteht von hier Zugriff auf die globale Konfiguration und Datenbasis.

Im Optimierungs-Thread wird ein Parser erzeugt, der die Eingabe interpretiert. Je nach übermittelter Information werden

- Datenbasis und Konfiguration initialisiert,
- Ein Optimierungslauf gestartet,
- Die gefundene Lösung abgefragt,
- Aufträge zur Datenbasis hinzugefügt,
- Fahrzeuge aus der Datenbasis gelöscht,
- Veränderungen an der Distanzmatrix vorgenommen.

Der Parser informiert den Optimierungs-Thread durch das Setzen von antSystemAction darüber, was das Ameisen-System tun soll, nämlich entweder Nichts, einen Optimierungslauf durchführen oder den Status ausgeben.

Neben der Datenbasis und der Konfiguration wird auch der Status des Ameisen-Systems, antSystemStatus, als statische Variable in der Klasse OptimizerThread verwaltet. Dadurch wird sichergestellt, dass diese drei Objekte selbst dann nicht verloren gehen, wenn es keine Instanz von OptimizerThread mehr gibt, sondern erst dann, wenn die JVM terminiert wird.

Der Optimierungs-Thread erzeugt zur Optimierung einen Optimizer, der wiederum die Erzeugung des Ameisen-Systems veranlasst, und es startet. Da der Optimizer und das Ameisen-System im Thread des OptimizerThread laufen, haben auch sie Zugriff auf die Konfiguration und die Datenbasis, also alle für die Optimierung notwendigen Informationen.

5.2.3. Ameisen-System, Ameisen und Ausführungs-Thread

Dem Ameisen-System werden bei seiner Erzeugung alle für die Optimierung notwendigen Parameter übergeben. Es wird die vorgesehene Anzahl an Ameisen erzeugt und diesen dabei die notwendigen Parameter weitergereicht. Außerdem werden Ausführungs-Threads (AntRunner) erzeugt, an die Teile der Optimierungsarbeit delegiert werden.

Die Ameisen werden in zwei Listen organisiert. Zum einen die Liste der Ameisen, die bereit sind, einen Lösungskandidaten zu konstruieren und zum anderen die Liste der Ameisen, die mit der Konstruktion eines Lösungskandidaten bereits fertig sind.

Jedem Ausführungs-Thread wird bei seiner Erzeugung eine Referenz auf das Ameisen-System mitgegeben. Wenn der Thread seine Arbeit aufnimmt, holt er sich vom Ameisen-System eine Ameise, die bereit ist, einen Lösungskandidaten zu bestimmen und startet den Berechnungslauf. Ist die Lösung bestimmt, so wird die Ameise an das Ameisen-System zurückgegeben und für die nächste wartende Ameise derselbe Ablauf durchgeführt. Dies wird solange wiederholt, bis der Ausführungs-Thread gestoppt wird, also ein Interrupt-Signal erhält. Der Zugriff auf die wartenden und fertigen Ameisen ist durch die Verwendung von *Monitoren*[7] synchronisiert, sodass die Arbeit mit mehreren Threads möglich ist. Auf diese Weise kann der Parallelisierungsgrad des Ameisen-Systems sehr einfach gesteigert und die verfügbare Rechenleistung optimal genutzt werden.

Die Lösungskonstruktion durch eine Ameise läuft so ab, dass zunächst eines der Fahrzeuge aus der Datenbasis ausgewählt wird und für dieses dann eine Tour bestimmt wird. Die Ameise beginnt damit, einen Trip zu erzeugen und ihm so lange Stops hinzuzufügen, bis er bezüglich der Ladekapazität des Fahrzeugs voll ist oder keine Zeit mehr

[7] Monitore sind ein Sprachkonzept, bei dem für Objekte eine Warteschlange erzeugt wird. Findet ein synchronisierter Zugriff auf ein Objekt statt, müssen sich die Threads in diese Warteschlange einreihen und warten, bis ihnen der Zugriff auf das synchronisierte Objekt zugeteilt wird. Die Funktionalität der Monitore wird durch den Compiler sichergestellt.

übrig ist. Ist noch Zeit, weitere Trips durchzuführen, so werden diese auch mit so vielen Stops gefüllt wie möglich. Zum Schluss kehrt das Fahrzeug zu seinem Heimatdepot zurück. Sofern es noch unverplante Aufträge *und* Fahrzeuge gibt, wählt die Ameise ein weiteres Fahrzeug aus und erstellt für dieses eine Tour. Da Fahrzeuge immer nur dann benutzt werden, wenn es für sie tatsächlich etwas zu tun gibt, wird die Anzahl der für den Plan benötigten Fahrzeuge minimiert.

Die Ameisen können dabei durch ihren Status, der vom Typ AntStatus ist, verschiedene Signale geben:

- STATUS_OK
  Alles ist in Ordnung, es können weitere Stops hinzugefügt werden.
- STATUS_TOUR_FULL
  Die aktuelle Tour ist fertig berechnet worden. Sofern es noch sowohl unverplante Fahrzeuge als auch Aufträge gibt, ist ein neues Fahrzeug auszuwählen und zu verplanen.
- STATUS_TRIP_FULL
  Der aktuelle Trip ist voll, da das Fahrzeug voll beladen ist oder keine Zeit mehr bleibt, um weitere Aufträge zu erledigen. Bleibt noch Zeit nach, so muss ein neuer Trip eingefügt werden, damit weitere Aufträge verplant werden können.
- STATUS_PLAN_FINISHED
  Der gesamte Plan ist fertig berechnet worden und steht als Lösungskandidat zur Verfügung.

Die Auswahl des jeweils nächsten Stops verläuft in abgewandelter Form nach dem in (Kapitel 4.3, S. 18) aufgezeigten Verfahren und basiert auf der Bewertung aller möglichen Nachfolge-Stops mit einer Besuchswahrscheinlichkeit. Es findet jedoch keine Normierung der Wahrscheinlichkeiten statt, sodass sie in der Summe verschieden von 1 sein können. Befindet sich die Ameise bzw. das Fahrzeug am Stop i, so wird jedem möglichen Folgestop j nach der Formel

$$p_j = \tau_{(i,j)}^{\alpha} \cdot \eta_{(i,j)}^{\beta} \tag{5.1}$$

eine Wahrscheinlichkeit p_j zugewiesen. Außerdem wird die Summe aller Wahrscheinlichkeiten gespeichert. Es wird dann ein *gleichverteilter* Zufallswert x zwischen 0 und der Summe der Wahrscheinlichkeiten bestimmt, über den der nächste Stop ausgewählt wird. Die Auswahl des Folgestops läuft so ab, dass die möglichen Folgestops zunächst nach ihrer Identifikationsnummer sortiert werden. Dann werden die ihnen zugewiesenen Besuchswahrscheinlichkeiten solange kumuliert, bis der zuvor generierte Zufallswert x erreicht oder überschritten wird. Der Stop, dessen Besuchswahrscheinlichkeit zuletzt addiert wurde, wird dann als Folgestop ausgewählt. Dieses Verfahren wird auch von Dorigo und Stützle vorgeschlagen [DS04].

Sind alle Ameisen mit der Lösungskonstruktion fertig, wird die beste Lösung bestimmt und in den Status des OptimizerThread übernommen. Es schließen sich die Evaporation und das Neulegen der Pheromone an. Für jede Ameise werden entlang ihres Weges auf jeder benutzten Kante neue Pheromone ausgeschüttet. Die Berechnung der Konzentration findet ebenfalls nach einer leicht modifizierten Formel statt. Die in (Kapitel 4.3, S. 18) beschriebene Formel wird dahingehend verändert, dass die Kosten der bisher besten bekannten Lösung mit einbezogen werden. Hat eine Ameise die Kante (i, j) verwendet, so ergibt sich die neue Pheromonkonzentration aus

$$\tau_{(i,j)} \leftarrow \tau_{(i,j)} + \frac{c_{best}}{c_{ant}}. \tag{5.2}$$

Dabei bezeichnet c_{best} die Kosten der bisher besten bekannten Lösung und c_{ant} die Kosten der Lösung, die durch die aktuelle Ameise bestimmt wurde. Diese Modifikation kommt zur Anwendung, da die Konzentration der neu gelegten Pheromone umgekehrt proportional zu den Kosten der gefundenen Lösung ist und sich bei hohen Kostenwerten ansonsten sehr geringe Pheromonmengen ergäben. Sind für alle Ameisen neue Pheromone gelegt worden, so werden alle Ameisen zurück in die Liste der wartenden Ameisen verschoben und die Ausführungs-Threads darüber informiert, dass wieder Arbeit zu erledigen ist.

Der Zyklus der Lösungsberechnung wird so lange wiederholt bis die in der Konfiguration festgelegte Maximalzahl an Durchläufen erreicht ist. Falls Stagnation auftritt oder ein anderes Abbruchkriterium erfüllt ist, kommt es evtl. schon vorher zum Ende der Berechnung.

Die meisten der in (Kapitel 4.2, S. 16) vorgestellten Erweiterungen kommen hier nicht zum Einsatz, sollten sie sich als notwendig erweisen, können sie später ergänzt werden. Als einzige Ergänzung ist eine untere Grenze für die Pheromone auf den Kanten implementiert, die standardmäßig auf 1 gesetzt wird.

5.2.4. Pheromone

Die Pheromone auf den Kanten werden in Form eines Graphen im Status des Ameisen-Systems abgebildet.

Die Knoten des Graphen entsprechen den Orten, also den Fahrzeug- und Ladedepots und den Aufträgen, die Kanten den Verbindungen zwischen den Orten. Die Bewertung der Kanten gibt die Pheromonkonzentration auf ihr an. Da die Kanten im vorliegenden Problem gerichtet sind, also die Länge sich für beide Richtungen unterscheiden kann, werden auch die Pheromone pro Richtung gespeichert.

Über die Methoden getDistance und setDistance lässt sich die Länge der Kante bzw. die Pheromonkonzentration auslesen und verändern. Die restlichen Methoden sind für die Verwaltung von Pheromonen eher uninteressant, aber bieten ein gutes Werkzeug zur Arbeit mit allgemeinen Graphen und können für die Umsetzung von Nachbarschaftslisten herangezogen werden.

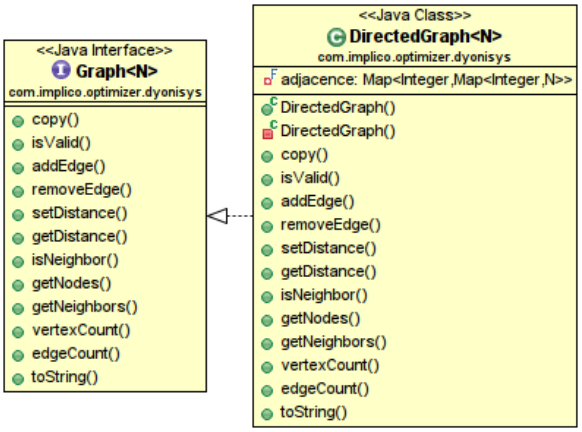


Abbildung 5.12.: UML-Diagramm zu den Pheromonen.

Über den Typparameter N wird angegeben, von welchem Typ die Pheromonwerte sein sollen, wir verwenden hier Fließkommawerte (Double), da allein schon aufgrund der Evaporation Ganzzahlen (Integer) ausgeschlossen sind.

5.2.5. Abbildung der Dynamik

Die Abbildung der dynamischen Ereignisse erfolgt indirekt, sodass das Ameisen-System davon im Prinzip nichts mitbekommt. Tritt ein Ereignis ein, so erhält DYONISYS diese Information stets vom SAP-System über ATOS, wie in (Abbildung 5.11, S. 35) dargestellt. Der Parser[8] interpretiert die Eingabe und nimmt derart Veränderungen am Szenario vor, dass die neue Situation abgebildet wird. Dazu muss solange gewartet werden, bis das Ameisen-System die Verwaltungsphase durchlaufen hat. Vor dem erneuten Eintritt in die Konstruktionsphase wird es dann angehalten, um die Veränderungen vornehmen zu können. Das Ameisen-System selbst muss dabei nichts tun, es muss nur hinterher auf die neue Situation reagieren.

Abhängig vom Typ der aufgetretenen Ereignisse müssen dabei unterschiedliche Aktionen durchgeführt werden. Da es dabei in der Regel der Fall sein wird, dass es nur lokale Veränderungen gibt, die Gesamtsituation also im Wesentlichen bestehen bleibt, können die bereits vorhandenen Pheromonwerte in die neue Situation übernommen werden. Somit kann die gemeinsame, verteilte Information, das Gedächtnis der Ameisen, weiter genutzt werden.

In den folgenden Beschreibungen der Behandlung der Ereignisse gehen wir davon aus, dass sich das Ameisen-System in angehaltenem Zustand befindet. Nach der Durchführung der Änderung wird die Berechnung fortgesetzt und das Ameisen-System tritt wieder in die Konstruktionsphase ein.

[8] siehe (Kapitel 5.1.8, S. 31).

Ausfall eines Fahrzeugs

Fällt ein Fahrzeug aus, so wird es aus der Datenbasis gelöscht. Alle Aufträge, die durch dieses Fahrzeug beliefert werden sollen, werden im aktuellen Plan als unverplant markiert. In der nächsten Konstruktionsphase versuchen die Ameisen, die nun unverplanten Aufträge auf die verbleibenden Fahrzeuge aufzuteilen. Bereits nach einem Zyklus können dabei alle betroffenen Aufträge wieder verplant sein, es ist jedoch ebenso denkbar, dass die Belieferung aller ursprünglich geplanten Aufträge in der neuen Situation nicht möglich ist.

Hinzukommen von Aufträgen

Kommen neue Aufträge hinzu, so werden diese zunächst in der Datenbasis in die Menge aller Aufträge eingefügt. Im aktuellen Plan werden sie als unverplant vermerkt.

Durch die neuen Aufträge gibt es im Problemgraphen nun auch neue Knoten und somit neue Kanten, auf denen noch keine Pheromone vorhanden sind. Für die Konstruktion der Lösung sind diese aber erforderlich, sonst würden die Ameisen den betroffenen Knoten aufgrund der Berechnungsvorschrift (Gleichung 5.1, S. 38) immer eine Besuchswahrscheinlichkeit von 0 zuweisen, obwohl sie noch nicht besucht wurden.

Für die Pheromonkonzentration auf den neuen Kanten kommen zwei sinnvolle Möglichkeiten in Betracht. Zum einen der Durchschnitt der Konzentrationen auf den bisherigen Kanten und zum anderen eine Konzentration in Höhe der Initialkonzentration τ_0. Aus Gründen der Performance wurde die zweite Option, also das Legen von Pheromonen der Konzentration τ_0 gewählt.

Veränderung von Wegkosten

Die Veränderung von Wegkosten tritt z.B. ein, wenn sich ein Stau bildet oder eine Straßensperrung besteht. Es gibt zum einen die Möglichkeit, diese Veränderung direkt in der Distanzmatrix zu speichern und zum anderen, eine neue, aktualisierte Distanzmatrix vom xServer abzurufen. Die Ameisen verwenden dann beim nächsten Durchlauf der Konstruktionsphase die aktualisierten Distanzinformationen und beziehen sie direkt in die Berechnung der Besuchswahrscheinlichkeiten ein. Eine denkbare Erweiterung des Verhaltens ist die Veränderung der Konzentration der Pheromone auf den betroffenen Kanten proportional zur Veränderung der Wegkosten.

Durch die Veränderung der Wegkosten kann die bisherige beste Lösung stark verschlechtert werden, da Kunden ggf. nicht mehr fristgerecht erreicht werden können. Alle Aufträge, die durch dieses Problem betroffen sind, werden als unverplant markiert und der dadurch entstehende Plan als aktuell bester gespeichert.

6. Test

In diesem Kapitel werden wir einige Tests durchführen, mit denen wir die Funktionalität von Dyonisys prüfen wollen. Dabei werden wir zum einen die Reaktion auf dynamische Ereignisse untersuchen und zum anderen einen Vergleich mit IcedG durchführen. Dyonisys soll zwar in der Praxis zur Anpassung bereits vorhandener Pläne eingesetzt werden, ist aber auch in der Lage, selbst Pläne zu berechnen.

Dabei sei darauf hingewiesen, dass Pläne, die bei erster Betrachtung schlecht aussehen mögen, dies nicht unbedingt sind. Aufgrund ungünstiger Zeitfenster kann es notwendig sein, dass ein Fahrzeug ähnliche Strecken mehrfach fahren muss, um so beispielsweise einen Kunden am Vormittag zu beliefern und seinen Nachbarn am Nachmittag. Auch durch die Tatsache, dass zwei dicht beieinander liegende Aufträge unter Umständen keine Direktverbindung aufweisen, kann der Eindruck erweckt werden, dass ein Fahrzeug "kreuz und quer" fährt. Da die Routing-Informationen, die wir für die Optimierung vom xServer erhalten, lediglich Aussagen bezüglich Distanz, Reisedauer und Mautkosten zwischen zwei Orten treffen, ist uns der exakte Weg, der dabei gefahren werden muss, nicht bekannt. Daher werden wir die Reise zwischen zwei Knoten als direkte Verbindung zwischen diesen darstellen.

Die genaue Bewertung der Lösungen für die verwendeten Szenarien wird uns nicht möglich sein, da wir die jeweiligen optimalen Lösungen nicht kennen. Wir werden nur dazu in der Lage sein, quantitative Aussagen zu treffen oder die erzielten Ergebnisse mit denen anderer Optimierer, wie z.B. IcedG zu vergleichen.

6.1. Testtool

Die Tests werden mit dem von Nicolas Woldt extra zu diesem Zweck implementierten Testtool durchgeführt. Es bietet eine graphische Oberfläche und die Möglichkeit, sämtliche Optimierungsparameter einzustellen und die Ladedepots, Aufträge, Truckdepots etc. zu bearbeiten. Außerdem ist es in der Lage, die berechneten Touren anzuzeigen, dabei werden die Aufträge als rote, die Truckdepots als grüne und die Ladedepots als blaue Punkte dargestellt. Zur besseren Unterscheidbarkeit werden die Touren der Fahrzeuge in unterschiedlichen Farben dargestellt. (Abbildung 6.1, S. 43) zeigt die Startansicht des Testtools.

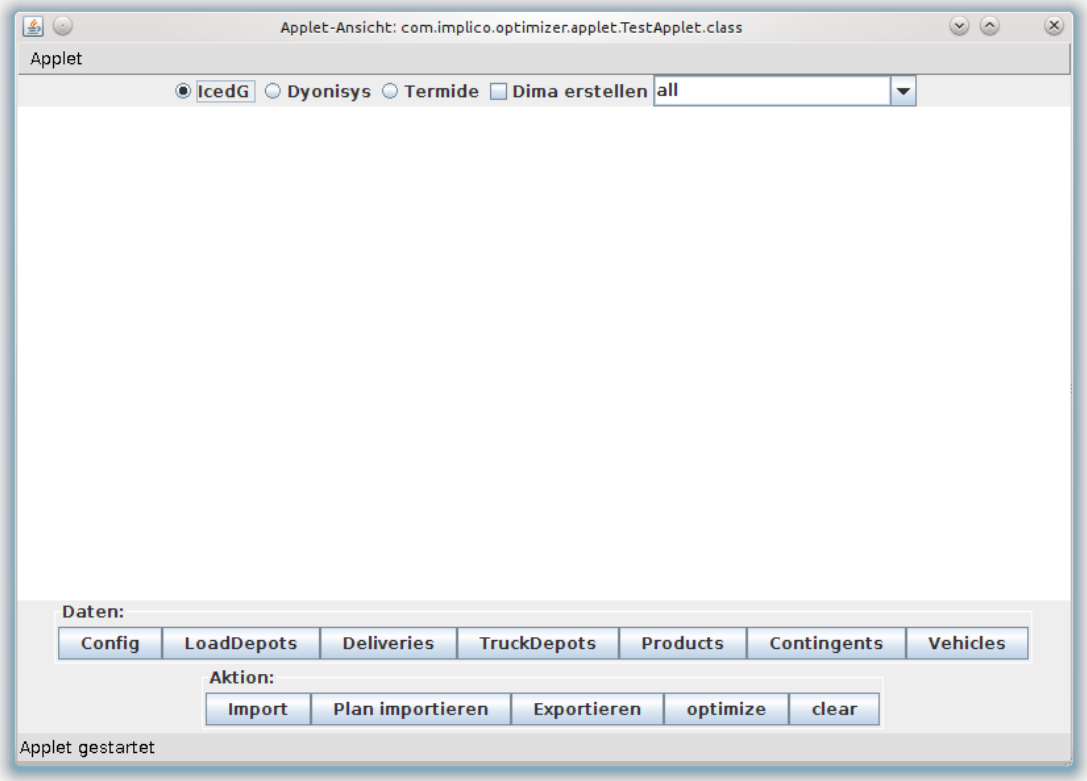


Abbildung 6.1.: Startansicht des Testtools

Zur Lösungsberechnung stellt das Testtool eine Verbindung zu dem auf einem Tomcat-Webserver gehosteten Optimierer-Servlet her und delegiert die Arbeit an dieses. Die berechnete Lösung wird an das Testtool übermittelt und dargestellt.

6.2. Dynamische Tests

Um DYONISYS' Reaktionsfähigkeit auf dynamische Ereignisse zu testen, verwenden wir ein Szenario, in dem 4 Fahrzeuge, 23 Aufträge, 2 Truckdepots und 2 Ladedepots vorkommen. Entsprechend der geographischen Verteilung der Aufträge befinden sich 1 Fahrzeug im nördlichen und 3 im südlichen Truckdepot. Die Kontingente enthalten eine so hohe Warenmenge, dass kein Engpass entsteht und alle Aufträge beliefert werden können. Die Transportkammern der Fahrzeuge haben ein Volumen von 15.000 Litern, jeder der Aufträge hat eine Nachfrage von 5.000 Litern. Die Fahrzeuge können also 3 Aufträge beliefern, ehe sie erneut betankt werden müssen.

Das beschriebene Szenario ist in (Abbildung 6.2, S. 44) dargestellt. Zum Testen von DYONISYS' Funktionalitäten (Reaktion auf Fahrzeugausfall, neue Aufträge, Wegkostenveränderung) werden wir das Szenario jeweils in geeigneter Weise anpassen.

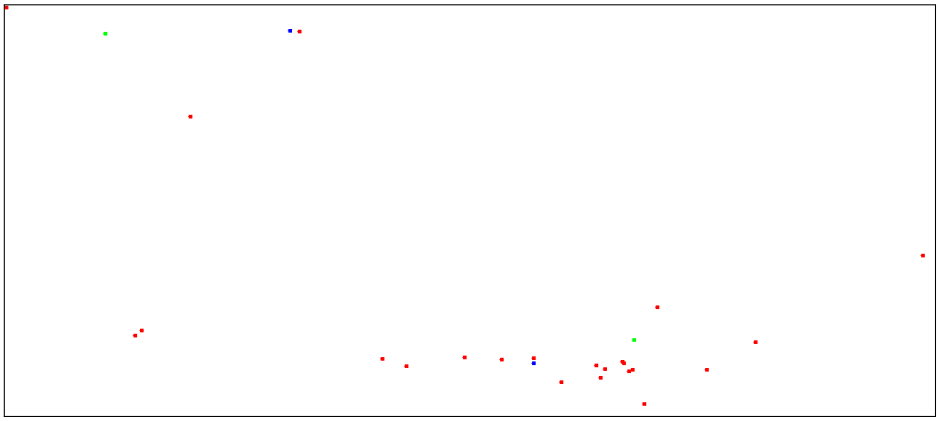


Abbildung 6.2.: Das Testszenario mit 4 Fahrzeugen, 23 Aufträgen, 2 Truckdepots und 2 Ladedepots

6.2.1. Fahrzeugausfall

Zum Planungszeitpunkt wird davon ausgegangen, dass vier Fahrzeuge zur Belieferung der Aufträge eingesetzt werden können. DYONISYS berechnet für diesen Fall den in (Abbildung 6.3, S. 45) dargestellten Plan, in dem alle 23 Aufträge beliefert werden können.

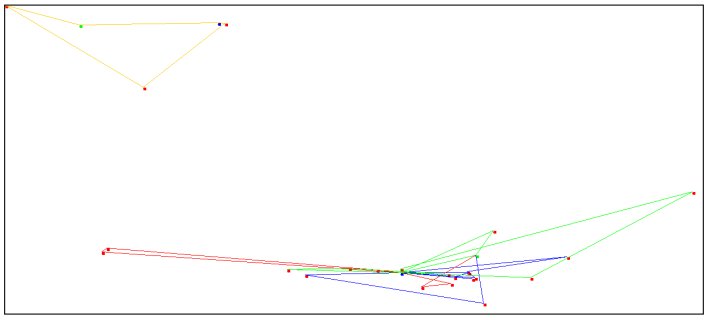


Abbildung 6.3.: Plansituation, in der alle Aufträge, aufgeteilt auf die 4 vorhandenen Fahrzeuge, beliefert werden können.

Durch den unerwarteten Ausfall eines der Fahrzeuge tritt eine Situation ein, in der 8 der 23 Aufträge nicht mehr beliefert werden. Die übrigen Fahrzeuge sind dabei nicht betroffen und verbleiben zunächst wie geplant (Abbildung 6.4, S. 45).

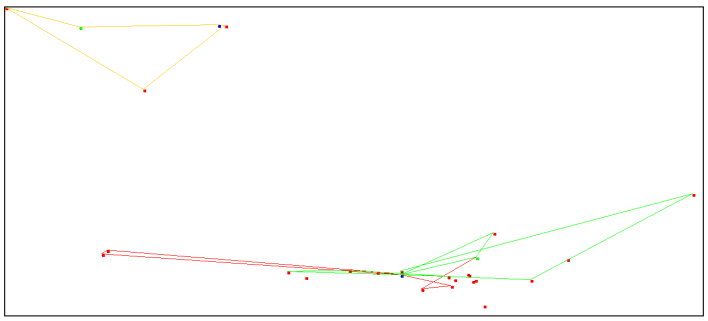


Abbildung 6.4.: Durch den Ausfall eines der Fahrzeuge fallen 8 Aufträge aus dem Plan. Die übrigen Fahrzeuge sind nicht betroffen.

Durch einen erneuten Optimierungslauf können die Aufträge im neuen Plan derart auf die 3 verbliebenen Fahrzeuge aufgeteilt werden, dass nur einer der Aufträge unbeliefert bleibt (Abbildung 6.5, S. 45).

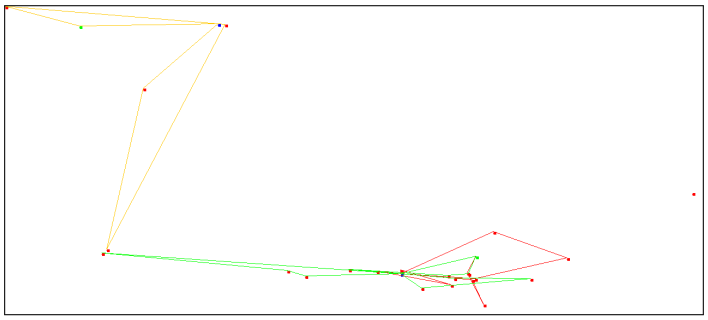


Abbildung 6.5.: Durch einen erneuten Optimierungslauf werden die Aufträge auf die verbleibenden Fahrzeuge verteilt, nur einer bleibt unverplant.

6.2.2. Neue Aufträge

Zum Zeitpunkt der Planung sind 18 Aufträge für den zu planenden Tag vorgemerkt. DYONISYS kann sie unter Verwendung von nur 3 der 4 verfügbaren Fahrzeuge einplanen (Abbildung 6.6, S. 46).

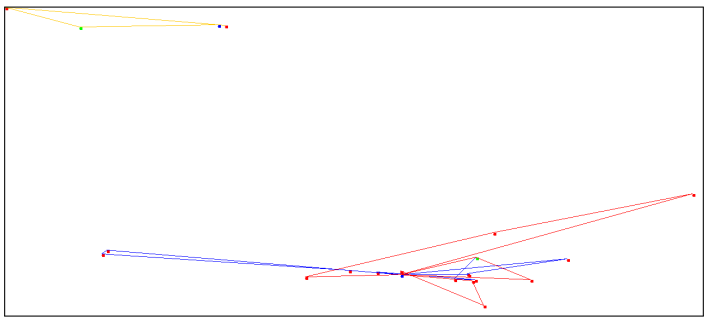


Abbildung 6.6.: Alle 18 Aufträge können mit nur 3 Fahrzeugen beliefert werden.

Durch Kundenanfragen kommen 5 neue Aufträge hinzu, die zunächst unverplant sind (Abbildung 6.7, S. 46).

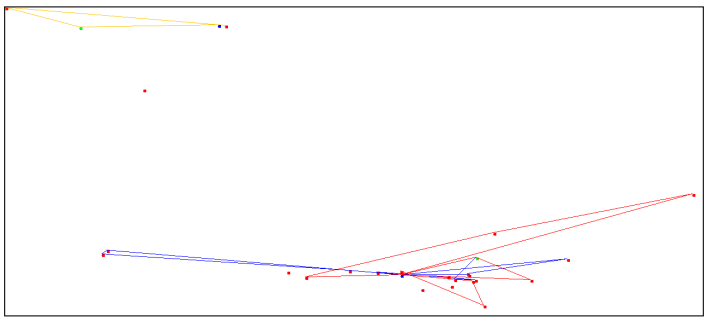


Abbildung 6.7.: 5 neue Aufträge sind durch Kundenanfragen hinzugekommen, sie sind zunächst unverplant.

In einem erneuten Optimierungslauf ist DYONISYS in der Lage, alle Aufträge einzuplanen, dafür werden jedoch alle 4 verfügbaren Fahrzeuge benötigt.

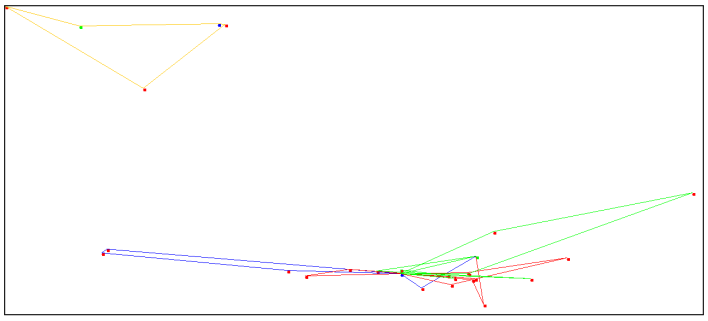


Abbildung 6.8.: Durch einen erneuten Optimierungslauf konnten alle Aufträge eingeplant werden.

6.2.3. Wegkostenveränderung

Die Veränderung von Wegkosten werden wir dadurch testen, dass wir im ersten Schritt einen Plan für das gegebene Szenario berechnen. Als zweites werden wir die Kosten der Wege, die im Plan verwendet werden, zufallsbasiert bis zu 10 mal höher setzen und somit eine Stausimulation durchführen. In der Ausgangssituation berechnet DYONISYS die in (Abbildung 6.9, S. 47) dargestellte Lösung.

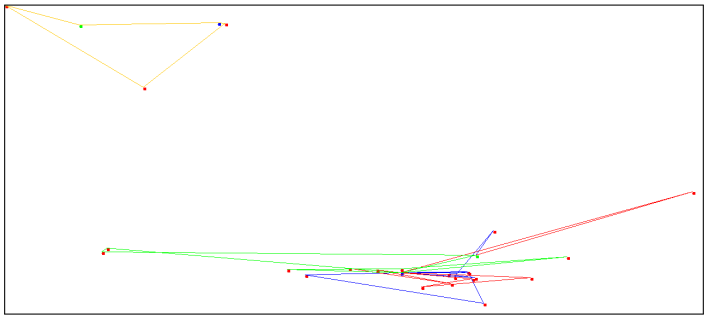


Abbildung 6.9.: Die Plansituation vor dem Auftreten des Staus, alle Aufträge sind eingeplant.

Durch den Stau verspätet sich die Ankunft der Fahrzeuge bei den Kunden und 7 von ihnen können nicht mehr rechtzeitig beliefert werden.

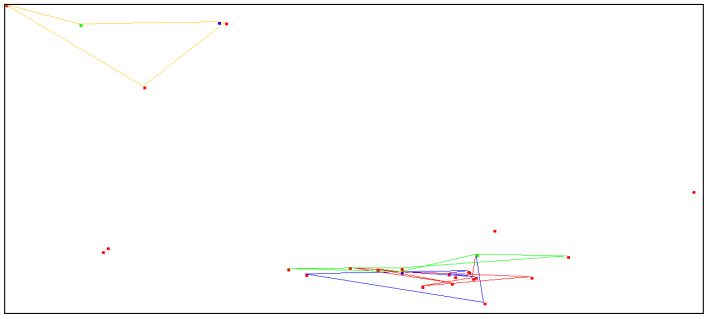


Abbildung 6.10.: Situation, die sich durch den Stau ergibt. 7 Aufträge können aufgrund der staubedingten Verspätung nicht beliefert werden.

Durch einen erneuten Optimierungslauf kann DYONISYS einen neuen Plan erstellen, in dem die Staus umfahren und somit alle Aufträge fristgerecht erledigt werden können.

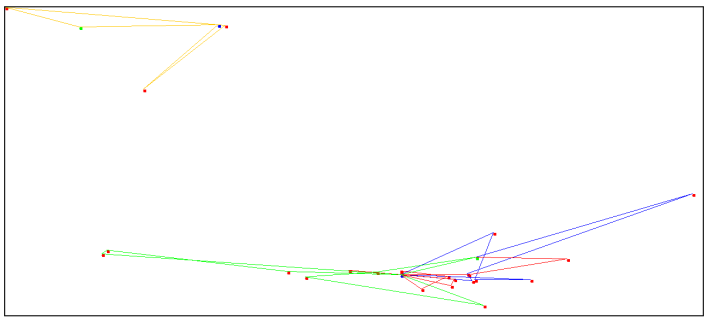


Abbildung 6.11.: Im neuen Plan werden die Staubereiche umgangen und alle Aufträge beliefert.

6.3. Variation der Optimierungsparameter

In diesem Abschnitt wollen wir untersuchen, welchen Einfluss die Optimierungsparameter auf die berechneten Lösungen haben. Als Referenzkonfiguration werden wir die in (Tabelle 6.1, S. 48) aufgelisteten Parametereinstellungen verwenden. Sie hat sich in Tests als brauchbar herausgestellt und bewegt sich in dem von Dorigo und Stützle in [DS04] vorgeschlagenen Bereich. Für jeden Parameter werden wir separate Tests durchführen, in denen wir nur den jeweiligen Parameter verändern. Die gleichzeitige Variation mehrerer Parameter werden wir nicht durchführen, da dann kein direkter Rückschluss mehr auf die Auswirkungen eines bestimmten Parameters mehr möglich ist.

Parameter	Wert
Anzahl Ameisen n_{ant}	10 / Anzahl Knoten
Anzahl Zyklen n_{run}	100
Evaporationsrate ρ	0.1
Initialpheromone τ_0	25
Gewichtung der Pheromone α	1
Gewichtung der Heuristik β	5

Tabelle 6.1.: Verwendete Referenzkonfiguration

Da bezüglich der Anzahl zu verwendender Ameisen unterschiedliche Empfehlungen bestehen, werden wir diese in jedem der Testfälle berücksichtigen. Die in der Literatur zu findenden Empfehlungen bewegen sich zwischen 10 Ameisen und so vielen, wie der Problemgraph Knoten aufweist [DDG99], [DS04], dies sind auch die beiden Werte, die wir verwenden werden.

Im Rahmen dieser Tests wollen wir nicht nur untersuchen, welchen Einfluss die Veränderung der jeweiligen Parameter nimmt. Wir wollen auch eine Empfehlung dazu geben, welche Parameter-Konfiguration zum einen zur Erzielung guter Lösungen verwendet werden kann und zum anderen die Laufzeit nicht allzu sehr erhöht. Daher ist es notwendig, dass wir die zur Lösungsberechnung benötigte Zeit ebenfalls betrachten. Es sei jedoch explizit darauf hingewiesen, dass die gemessenen Zeiten nur zu dem Zweck dienen, sie in Relation zur erzielten Lösungsverbesserung zu setzen, die durch die Variation von Parametern entsteht. Auf dieser Grundlage kann dann für die Praxis entschieden werden, ob die erzielte Qualitätsverbesserung der Lösung die damit verbundene Rechenzeit rechtfertigt oder ob zugunsten der Performance eine weniger gute Lösung in Kauf zu nehmen ist.

An dieser Stelle sei noch der Hinweis gegeben, dass die gemessenen Laufzeiten keine allgemeine Gültigkeit aufweisen. Sie hängen stark von der Leistungsfähigkeit des verwendeten Rechners, in diesem Fall ein Intel Core-i5 M560 mit 6MB Cache und 8GB Arbeitsspeicher, ab. Auf anderen Rechnern ist davon auszugehen, dass die Messungen vollkommen andere Laufzeiten ergeben. Außerdem kann nicht garantiert werden,

dass die Messungen störungsfrei ablaufen. So *können* z.B. Verwaltungsaktivitäten des Betriebssystems oder der JVM die Ausführung des Ameisen-Systems beeinflussen und verlangsamen. Wir werden das Ameisen-System in den Tests mit den auf dem Core-i5 M560 maximal möglichen 4 voll-parallelen Threads laufen lassen.

Über das bereits in (Kapitel 6.2, S. 44) vorgestellte Szenario hinaus werden wir eines mit 4 Fahrzeugen, 31 Aufträgen, 1 Truckdepot und 1 Ladedepot verwenden. Auch hier hat jeder Kunde eine Nachfrage von 5.000 Litern und jedes Fahrzeug eine Kapazität von 15.000 Litern. Es ist in (Abbildung 6.12, S. 49) zu finden.

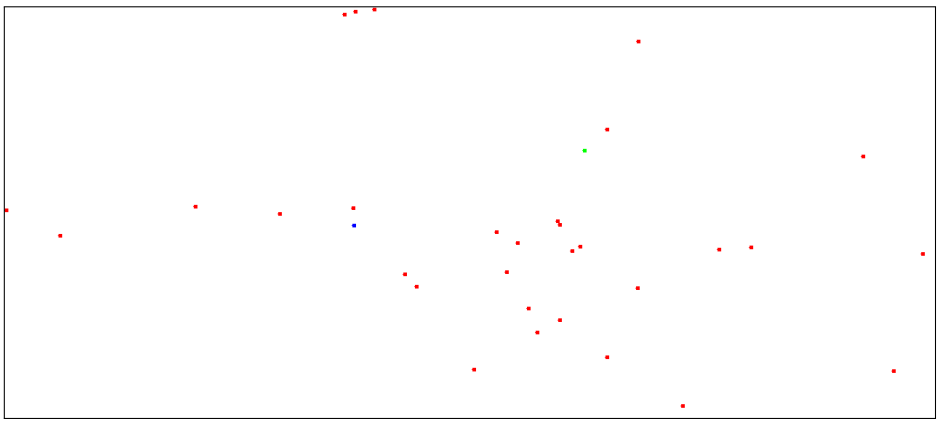


Abbildung 6.12.: Das große Testszenario

Wir bezeichnen das Szenario aus (Abbildung 6.2, S. 44) als s_{klein} und das Szenario aus (Abbildung 6.12, S. 49) als $s_{groß}$. Jeden Test werden wir 100 mal durchführen und dann den Mittelwert der gemessenen Ergebnisse bilden. Die entstehenden Kosten geben wir allgemein in Geldeinheiten an und bezeichnen sie mit c, die Distanz messen wir in Längeneinheiten und bezeichnen sie mit d. Die für die Berechnungen verstrichene Zeit geben wir in Millisekunden an und nennen sie t.

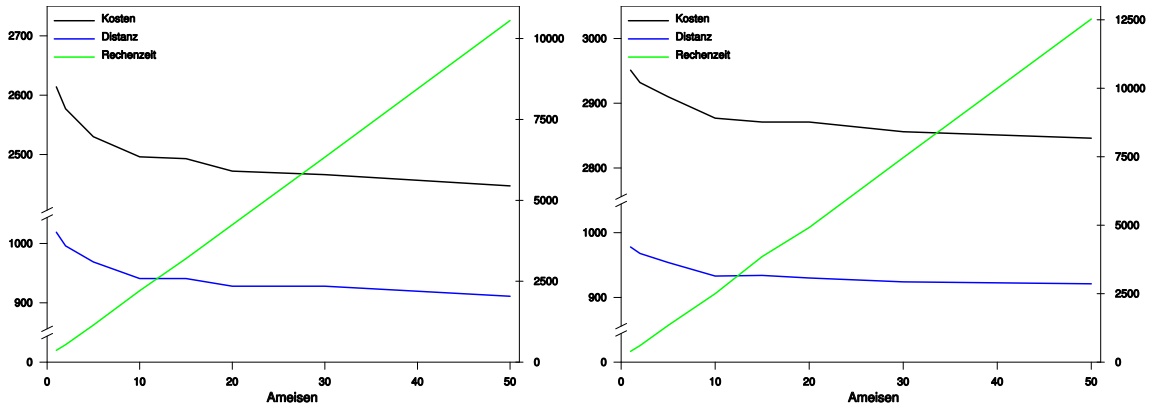
6.3.1. Anzahl der Ameisen

Es ist klar zu erkennen, dass durch die Verwendung von mehr Ameisen bessere Lösungen erzielt werden können. (Abbildung 6.13, S. 50) kann entnommen werden, dass der pro hinzugefügter Ameise erzielte Qualitätszuwachs bereits ab einer Zahl von 10 Ameisen deutlich abnimmt. Die Laufzeit hingegen nimmt pro hinzugefügter Ameise um einen konstanten Betrag zu, die Vermutung liegt nahe, dass sie linear von der Anzahl der verwendeten Ameisen abhängt.

Für die Praxis ergibt sich daraus: Die Anzahl der Ameisen sollte nicht zu hoch gewählt werden. Ab einer bestimmten Zahl von Ameisen ist zu erwarten, dass sich durch weitere Ameisen keine Lösungsverbesserung mehr erzielen lässt, während aber die Laufzeit immer weiter zunimmt.

	s_{klein}			$s_{groß}$		
n_{ant}	c	d	t	c	d	t
1	2.614	1.019	367	2.951	978	395
2	2.577	996	539	2.932	968	604
5	2.530	969	1.144	2.910	954	1.343
10	2.496	941	2.207	2.877	933	2.501
15	2.493	941	3.199	2.871	934	3.856
20	2.472	928	4.246	2.871	930	4.918
30	2.466	928	6.334	2.856	924	7.468
50	2.447	911	10.556	2.846	921	12.529

Tabelle 6.2.: Messungen bei Variation der Ameisenanzahl



(a) Szenario s_{klein} (b) Szenario $s_{groß}$

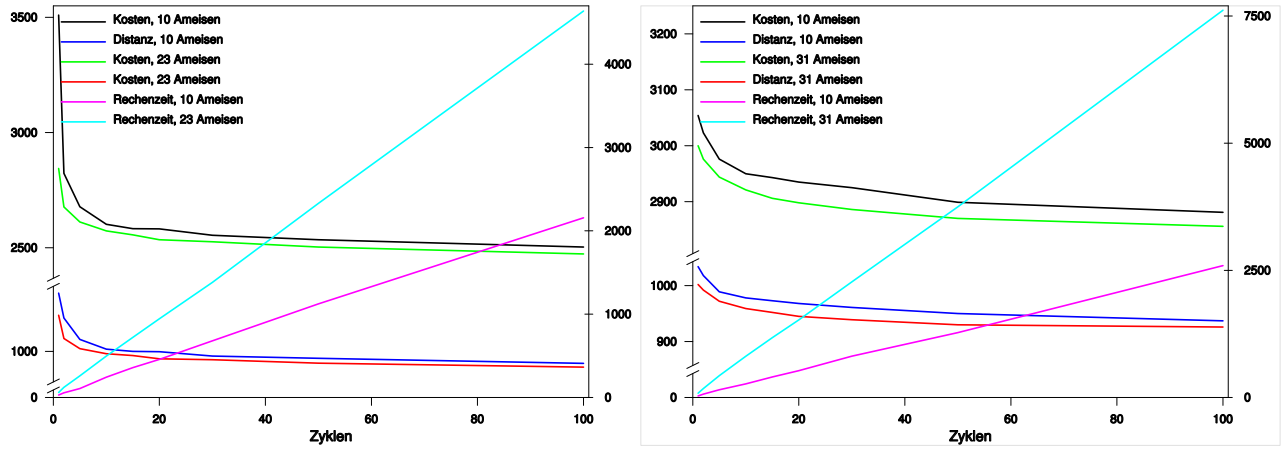
Abbildung 6.13.: Entwicklung von Kosten, Distanz und Rechenzeit bei Variation der Ameisenanzahl

6.3.2. Anzahl der Zyklen

Die Beobachtungen zeigen deutlich, dass eine höhere Anzahl an Zyklen zu besseren Lösungen führt. Ab einer Grenze von ca. 20 Zyklen ist diese Verbesserung jedoch nicht mehr signifikant, wie (Tabelle 6.3, S. 51) und (Abbildung 6.14, S. 51) entnommen werden kann. Eine Erhöhung um diese Grenze hinaus kann zwar das erzielte Ergebnis verbessern, es muss jedoch geprüft werden, ob die damit verbundene Erhöhung der Rechenzeit für praktische Anwendungen zu rechtfertigen ist. Auch bezüglich der Anzahl der Zyklen liegt die Vermutung nahe, dass die Laufzeit von DYONISYS linear von ihnen abhängt.

	$n_{ant} = 10$						$n_{ant} = 23$			$n_{ant} = 31$		
	s_{klein}			$s_{groß}$			s_{klein}			$s_{groß}$		
n_{run}	c	d	t	c	d	t	c	d	t	c	d	t
1	3.510	1.253	30	3.054	1.034	31	2.844	1.157	60	3.000	1.002	85
2	2.823	1.145	56	3.023	1.018	67	2.677	1.056	122	2.976	992	177
5	2.678	1.052	107	2.976	989	152	2.612	1.012	260	2.944	972	429
10	2.602	1.010	244	2.950	978	268	2.573	990	497	2.921	959	810
15	2.583	1.000	359	2.943	973	404	2.556	982	722	2.906	952	1.176
20	2.582	999	457	2.935	968	527	2.535	968	944	2.898	945	1.526
30	2.554	980	680	2.925	961	812	2.526	964	1.382	2.886	939	2.274
50	2.535	970	1.124	2.899	950	1.276	2.503	949	2.329	2.870	930	3.749
100	2.503	948	2.156	2.881	937	2.592	2.473	932	4.638	2.856	926	7.614

Tabelle 6.3.: Messungen bei Variation der Zyklenzahl



(a) Szenario s_{klein} (b) Szenario $s_{groß}$

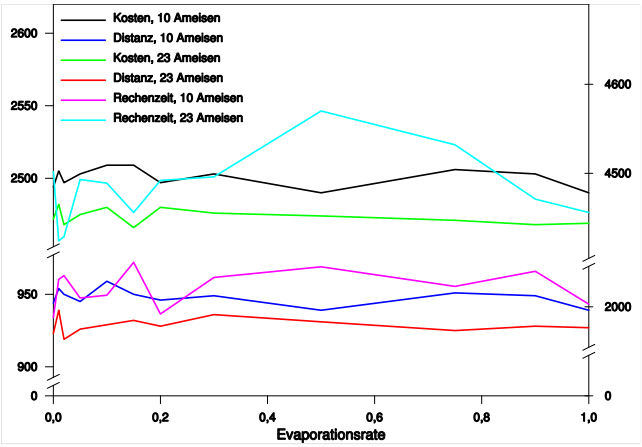
Abbildung 6.14.: Entwicklung von Kosten, Distanz und Rechenzeit bei Variation der Zyklenzahl

6.3.3. Evaporationsrate

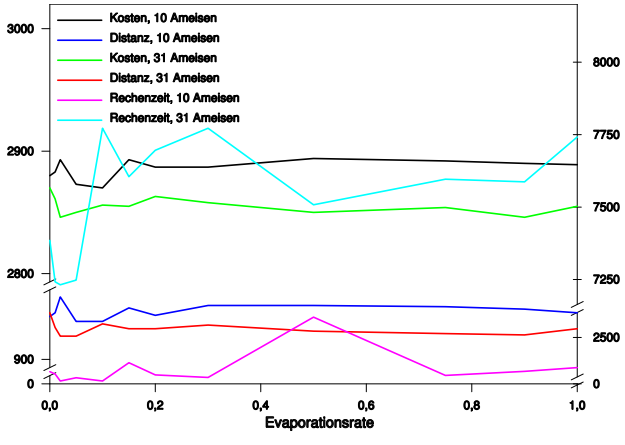
Entgegen den Erwartungen zeigen die in (Tabelle 6.4, S. 52) und (Abbildung 6.15, S. 52) dokumentierten Messungen, dass die Evaporationsrate keinen nennenswerten Einfluss auf die Qualität der berechneten Lösungen hat. Auch die Rechenzeit bleibt von der Evaporationsrate unberührt, was sich dadurch erklären lässt, dass sie keinen Einfluss auf die Anzahl der durchzuführenden Rechenoperationen nimmt.

	$n_{ant} = 10$						$n_{ant} = 23$			$n_{ant} = 31$		
	s_{klein}			$s_{groß}$			s_{klein}			$s_{groß}$		
ρ	c	d	t	c	d	t	c	d	t	c	d	t
0.0	2.494	944	1.988	2.880	935	2.382	2.472	923	4.502	2.870	938	7.385
0.01	2.505	954	2.031	2.883	938	2.374	2.482	939	4.424	2.861	926	7.242
0.02	2.497	950	2.035	2.893	951	2.350	2.468	919	4.429	2.846	919	7.232
0.05	2.503	945	2.010	2.873	931	2.361	2.475	926	4.493	2.850	919	7.248
0.1	2.509	959	2.013	2.870	931	2.344	2.480	929	4.489	2.856	929	7.772
0.15	2.509	950	2.209	2.893	942	2.413	2.466	932	4.456	2.855	925	7.605
0.2	2.497	946	1.992	2.887	936	2.371	2.480	928	4.492	2.863	925	7.696
0.3	2.503	949	2.033	2.887	944	2.362	2.476	936	4.496	2.858	928	7.772
0.5	2.490	939	2.045	2.894	944	2.570	2.474	931	4.570	2.850	923	7.508
0.75	2.506	951	2.023	2.892	943	2.369	2.471	925	4.532	2.854	921	7.596
0.9	2.503	949	2.040	2.890	941	2.383	2.468	928	4.471	2.846	920	7.587
1.0	2.490	939	2.003	2.889	938	2.396	2.469	927	4.456	2.855	925	7.742

Tabelle 6.4.: Messungen bei Variation der Evaporationsrate



(a) Szenario s_{klein}



(b) Szenario $s_{groß}$

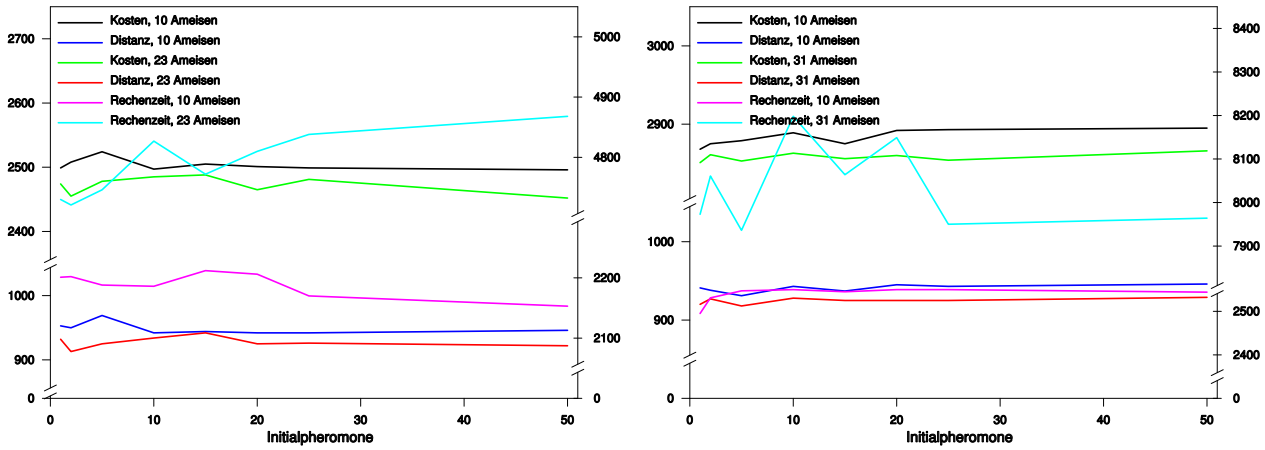
Abbildung 6.15.: Entwicklung von Kosten, Distanz und Rechenzeit bei Variation der Evaporationsrate

6.3.4. Initialpheromone

Auch die Konzentration der Initialpheromone scheint weder Einfluss auf die Lösungsqualität noch auf die Laufzeit zu nehmen, wie durch (Tabelle 6.5, S. 53) und (Abbildung 6.16, S. 53) nahegelegt wird. Die Rechenzeit zeigt zwar einige Schwankungen auf, diese sind jedoch so gering, dass sie mit den bereits erwähnten Hintergrundaktivitäten von Betriebssystem und JVM zu erklären sind. Eine Beeinflussung der Rechenzeit war auch nicht zu erwarten, da die Konzentration der Initialpheromone keinen Einfluss auf die Anzahl der durchzuführenden Rechenoperationen nimmt.

	$n_{ant} = 10$						$n_{ant} = 23$			$n_{ant} = 31$		
	s_{klein}			$s_{groß}$			s_{klein}			$s_{groß}$		
τ_0	c	d	t	c	d	t	c	d	t	c	d	t
1	2.499	953	2.201	2.868	941	2.495	2.474	932	4.730	2.851	920	7.973
2	2.508	950	2.202	2.875	938	2.531	2.455	913	4.721	2.861	927	8.061
5	2.524	969	2.188	2.879	931	2.547	2.478	925	4.746	2.853	918	7.936
10	2.497	942	2.186	2.889	943	2.585	2.485	934	4.827	2.863	928	8.198
15	2.505	944	2.212	2.875	937	2.545	2.488	942	4.772	2.856	925	8.064
20	2.501	942	2.206	2.892	945	2.593	2.465	925	4.810	2.860	925	8.149
25	2.499	942	2.170	2.893	943	2.585	2.481	926	4.838	2.854	925	7.950
50	2.496	946	2.153	2.895	946	2.544	2.452	922	4.868	2.866	929	7.964

Tabelle 6.5.: Messungen bei Variation der Initialpheromone



(a) Szenario s_{klein} (b) Szenario $s_{groß}$

Abbildung 6.16.: Entwicklung von Kosten, Distanz und Rechenzeit bei Variation der Initialpheromone

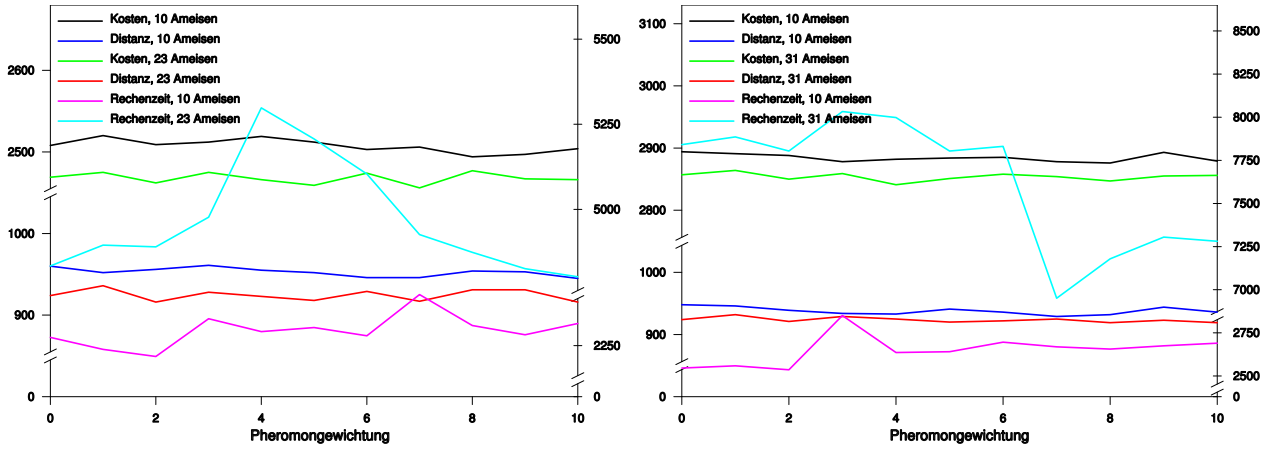
6.3.5. Gewichtung der Pheromone

Unerwarteterweise hat die Variation der Gewichtung der Pheromone keine Wirkung auf die Qualität der berechneten Lösungen. Eine Mögliche Erklärung dafür mag sein, dass die sonstige Konfiguration des Ameisen-Systems derart robust ist, dass eine Reaktion auf die Veränderung der Pheromongewichtung dadurch ausgeschlossen wird.

Auch die Rechenzeit wird nicht beeinflusst, es sind zwar leichte Schwankungen, jedoch keine Tendenz zu erkennen. Die Schwankungen sind daher auf Hintergrundaktivitäten von Betriebssystem und JVM zurückzuführen.

	$n_{ant} = 10$						$n_{ant} = 23$			$n_{ant} = 31$		
	s_{klein}			$s_{groß}$			s_{klein}			$s_{groß}$		
α	c	d	t	c	d	t	c	d	t	c	d	t
0	2.508	960	2.274	2.894	948	2.546	2.469	924	4.837	2.857	924	7.841
1	2.520	952	2.239	2.891	946	2.559	2.475	936	4.895	2.864	932	7.886
2	2.509	956	2.218	2.888	939	2.536	2.462	916	4.890	2.850	921	7.804
3	2.512	961	2.329	2.878	934	2.915	2.475	928	4.977	2.859	929	8.033
4	2.519	955	2.291	2.882	933	2.636	2.466	923	5.298	2.841	925	7.998
5	2.512	952	2.303	2.884	941	2.641	2.459	918	5.207	2.851	920	7.804
6	2.503	946	2.279	2.885	936	2.696	2.474	929	5.104	2.858	922	7.831
7	2.506	946	2.402	2.878	929	2.669	2.456	917	4.926	2.854	925	6.951
8	2.494	954	2.309	2.876	932	2.659	2.477	931	4.874	2.847	919	7.179
9	2.497	953	2.282	2.893	944	2.675	2.467	931	4.826	2.855	923	7.305
10	2.504	945	2.315	2.879	936	2.690	2.466	916	4.802	2.856	919	7.281

Tabelle 6.6.: Messungen Variation der Gewichtung der Pheromone



(a) Szenario s_{klein} (b) Szenario $s_{groß}$

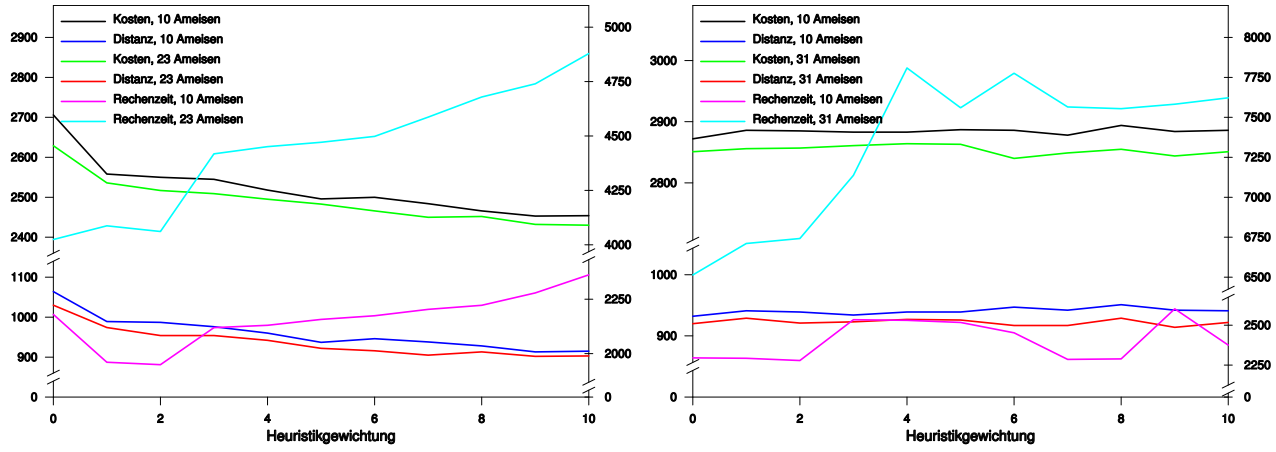
Abbildung 6.17.: Entwicklung von Kosten, Distanz und Rechenzeit bei Variation der Gewichtung der Pheromone

6.3.6. Gewichtung der Heuristik

Bei der Veränderung der Gewichtung der Heuristik kann beobachtet werden, dass sich bessere Lösungen ergeben, je höher β eingestellt wird. Außerdem zeigt sich ein leichter Anstieg der Rechenzeit bei höheren Werten von β. Dies ist möglicherweise darauf zurückzuführen, dass, abhängig von dem zum Potenzieren verwendeten Verfahren, die benötigte Rechenzeit bei höheren Exponenten länger sein kann.

	$n_{ant} = 10$						$n_{ant} = 23$			$n_{ant} = 31$		
	s_{klein}			$s_{groß}$			s_{klein}			$s_{groß}$		
β	c	d	t	c	d	t	c	d	t	c	d	t
0	2.706	1.064	2.180	2.872	932	2.295	2.629	1.030	4.024	2.851	920	6.514
1	2.558	989	1.960	2.886	941	2.293	2.536	974	4.087	2.856	929	6.711
2	2.550	987	1.949	2.885	939	2.279	2.517	954	4.061	2.857	921	6.742
3	2.545	976	2.119	2.883	934	2.534	2.509	954	4.418	2.861	923	7.137
4	2.518	960	2.130	2.883	939	2.530	2.495	942	4.451	2.864	927	7.809
5	2.496	937	2.157	2.887	939	2.517	2.483	922	4.471	2.863	926	7.560
6	2.500	946	2.174	2.886	947	2.452	2.466	916	4.498	2.840	917	7.776
7	2.484	938	2.203	2.878	942	2.286	2.450	905	4.587	2.849	917	7.565
8	2.466	928	2.222	2.894	951	2.289	2.452	913	4.679	2.855	929	7.554
9	2.453	913	2.279	2.884	942	2.602	2.432	902	4.740	2.844	914	7.582
10	2.454	915	2.362	2.886	941	2.375	2.430	903	4.879	2.851	922	7.622

Tabelle 6.7.: Messungen bei Variation der Gewichtung der Heuristik

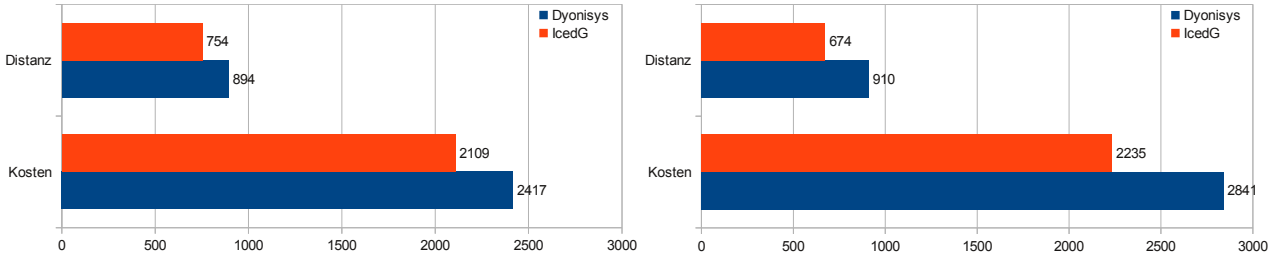


(a) Szenario s_{klein} (b) Szenario $s_{groß}$

Abbildung 6.18.: Entwicklung von Kosten, Distanz und Rechenzeit bei Variation der Gewichtung der Heuristik

6.4. Dyonisys vs. IcedG

In einem letzten Test wollen wir DYONISYS und ICEDG gegeneinander antreten lassen und die Qualität ihrer Lösungen vergleichen. Dafür werden wir wieder die beiden bereits bekannten Szenarien s_{klein} und $s_{groß}$ verwenden. Das Ameisen-System konfigurieren wir dabei so, dass die Ameisen viel Zeit für die Konstruktion der Lösungen bekommen. Im Fall von s_{klein} verwenden wir 23 Ameisen, im Fall von $s_{groß}$ 31. Das Ameisen-System soll 1.000 Zyklen durchlaufen, aber vorzeitig abbrechen, wenn nach 250 Durchläufen keine weitere Lösungsverbesserung gefunden werden kann. Die weiteren Parameter sind wie folgt konfiguriert: $\rho = 0.1, \tau_0 = 10, \alpha = 1, \beta = 5$. Wir führen die Tests wieder 100 mal durch und bilden dann Durchschnittswerte und betrachten Kosten und Distanz als dimensionslos, um sie in einer Graphik darstellen zu können.



(a) Szenario s_{klein} (b) Szenario $s_{groß}$

Abbildung 6.19.: Vergleich der Ergebnisse von DYONISYS und ICEDG

Den Vergleich mit DYONISYS kann ICEDG deutlich für sich entscheiden. Wie (Abbildung 6.19, S. 56) entnommen werden kann, findet ICEDG sowohl für s_{klein} als auch für $s_{groß}$ bessere Lösungen als DYONISYS. Die zur Lösungsfindung benötigte Rechenzeit wird nicht betrachtet, da DYONISYS und ICEDG auf unterschiedlichen Verfahren beruhen und somit nicht direkt vergleichbar sind, lediglich verfahrensunabhängige Größen, wie die Qualität der Lösungen, lassen sich für Vergleiche heranziehen.

Um möglicherweise Gründe für die unterschiedliche Lösungsqualität ausmachen zu können, betrachten wir exemplarisch pro Optimierer und Szenario einen Lösungsvorschlag in seiner graphischen Repräsentationsform.

(Abbildung 6.20, S. 56) und (Abbildung 6.21, S. 57) zeigen die Lösungen von DYONISYS und ICEDG zu s_{klein}. Auf den ersten Blick sehen sie recht ähnlich aus, bei genauerer Betrachtung lässt sich aber erkennen, dass in DYONISYS' Lösung einige

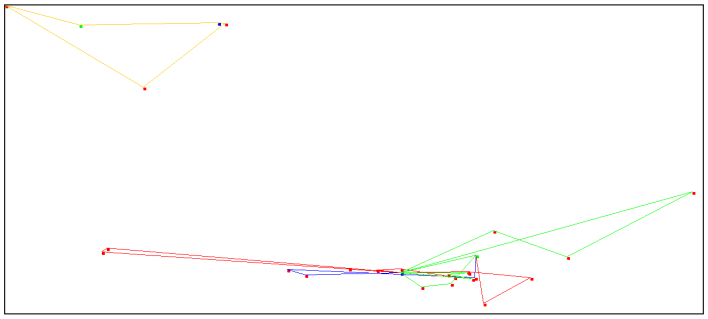


Abbildung 6.20.: Eine Lösung von DYONISYS zu s_{klein} mit c = 2.373 und d = 803

Überkreuzungen vorkommen,
während die Touren in IcedG's Lösung eher den rundreiseartigen Charakter aufweisen, den wir erwarten.

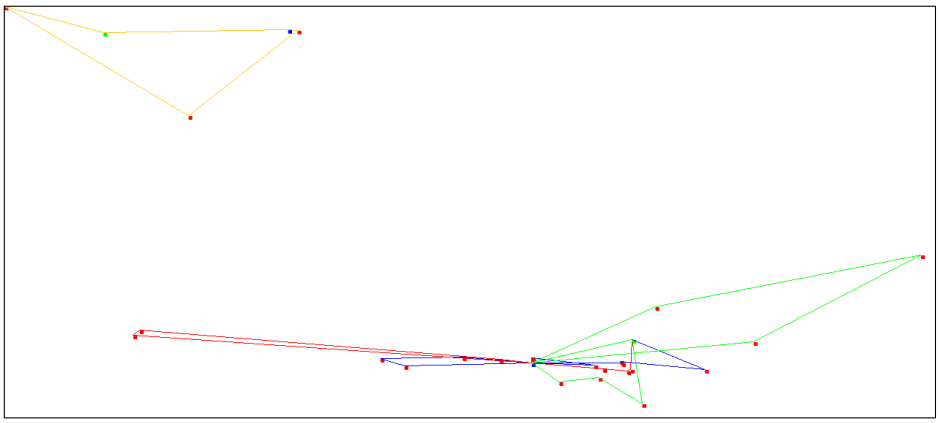


Abbildung 6.21.: Eine Lösung von IcedG zu s_{klein} mit c = 2.016 und d = 720

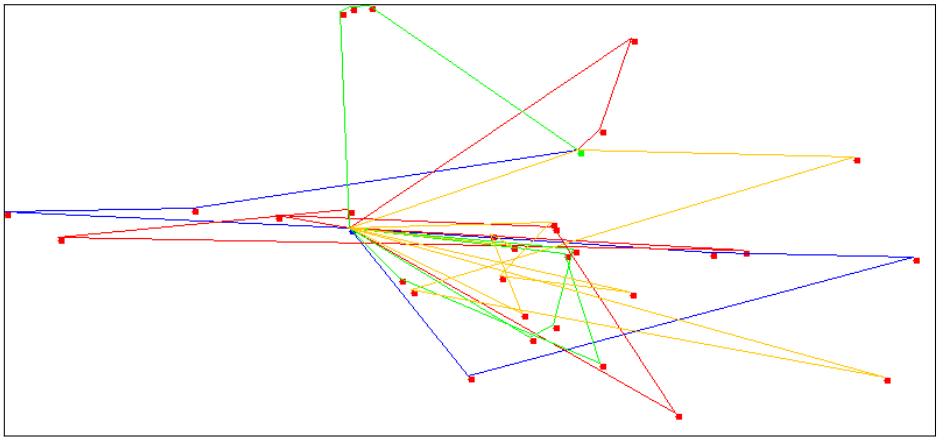


Abbildung 6.22.: Eine Lösung von Dyonisys zu s_{gross} mit c = 2.763 und d = 875

Die in (Abbildung 6.22, S. 57) dargestellte Lösung von Dyonisys zu s_{gross} lässt Probleme erkenne, die für die weniger guten Lösungen gegenüber IcedG verantwortlich sein können. In Dyonisys' Lösung werden Kunden, die sich weit voneinander entfernt befinden, direkt nacheinander besucht. Diesem Problem kann durch die Verwendung einer Nachbarschaftsliste begegnet werden.

Auch die Vorgehensweise, zunächst Fahrzeuge voll zu verplanen, bevor ein neues verwendet wird, kann die vorliegende Problematik begünstigen. IcedG hingegen ist in der Lage, mit der Situation umzugehen, wie (Abbildung 6.23, S. 57) zeigt.

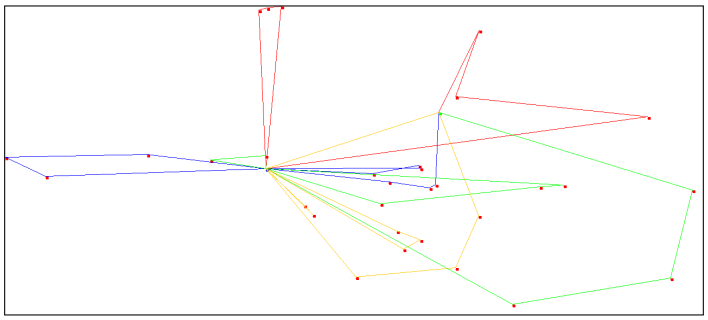


Abbildung 6.23.: Eine Lösung von IcedG zu s_{gross} mit c = 2.202 und d = 648

7. Abschließende Betrachtungen

7.1. Zusammenfassung

Die Zusammenfassung gibt noch einmal einen kurzen Überblick darüber, welche Thematiken in welchen Kapiteln behandeln worden sind und gegebenenfalls, welche Erkenntnisse sie nach sich gezogen haben.

Das erste Kapitel gibt einen informellen Überblick über das Problem der Tourenplanung und formuliert die Zielsetzung für diese Arbeit.

Im zweiten Kapitel wird aufgezeigt, in welchem Umfeld eine Lösung entwickelt werden soll und es wird die Entscheidung für die Verwendung der Programmiersprache Java erläutert.

Eine kurze Einführung in die theoretischen Grundlagen, die zum Verständnis der nachfolgenden Kapitel notwendig sind, wird im dritten Kapitel gegeben. Es wird dargelegt, dass das Problem der Tourenplanung zu den schwersten Problemen überhaupt zählt und eine exakte Lösung vermutlich nicht in praxistauglicher Zeit gefunden werden kann.

Das vierte Kapitel befasst sich mit der Beschreibung der Funktionsweise von Ameisen-Systemen und gibt eine informelle Einführung und eine Übersicht über Varianten und Erweiterungen. Nach der Betrachtung der formellen Konzepte von Ameisen-Systemen wird ihre Eignung für die Problematik der Tourenplanung diskutiert und als gut befunden.

Das fünfte und umfangreichste Kapitel dieser Arbeit befasst sich mit dem Entwurf und der Konzeption von DOT und Dyonisys. Im ersten Teil wird das Entstehen der gemeinsamen Basis für TermiDe, IcedG und Dyonisys erläutert. Der zweite Teil widmet sich der Umsetzung von Dyonisys. In diesem Rahmen wird festgestellt, dass sich die Behandlung dynamischer Ereignisse, die im Zusammenhang mit der Tourenplanung auftreten, als wesentlich einfacher herausstellt als vermutet.

Im sechsten Kapitel werden Tests zur Performance und Qualität von Dyonisys durchgeführt. Dabei wird untersucht, welchen Einfluss die einzelnen Parameter auf die Qualität der Lösungen nehmen. Zum Schluss des Kapitels wird ein Vergleich von Dyonisys und IcedG durchgeführt.

Der Anhang umfasst die Definition der Anfragesprache für die Optimierung mit DYONISYS und eine Übersicht über die dafür notwendigen Parameter.

7.2. Fazit

Wir haben gesehen, dass Ameisen-Systeme für die Problemstellung der dynamischen Tourenoptimierung gut geeignet sind und die richtige Wahl darstellen. Treten Ereignisse - und damit Veränderungen - auf, so lässt sich die vorliegende Probleminstanz dahingehend modifizieren, dass ebendies abgebildet wird. Die Ameisen orientieren sich neu und beachten diese Veränderung sofort bei der Lösungsberechnung. Ein großer Vorteil dabei ist, dass Ameisen-Systeme bereits so konzipiert sind, dass ebendieses Verhalten für sie "natürlich" ist, es muss ihnen nur die aktuelle Situation dargeboten werden und sie passen sich ihr an. Die Behandlung der dynamischen Ereignisse hat sich wesentlich einfacher gestaltet als zunächst vermutet.

Die durchgeführten Tests haben sich als wertvoll erwiesen. Durch sie konnte aufgedeckt werden, dass bei der Lösungsberechnung durch DYONISYS noch Probleme bestehen und das Ameisen-System nicht voll ausgereift ist.

Die Entscheidung für die Programmiersprache Java hat sich bewährt. Die Implementierung konnte termingerecht und problemlos fertiggestellt werden. Mit Hilfe guter Debugging-Werkzeuge war es möglich, Fehler effektiv aufzuspüren und zu beheben, sowie das Verhalten des Ameisen-Systems zu beobachten. Aufgrund der modularen Konzeption des Systems besteht die Möglichkeit zukünftige Erweiterungen oder Modifikationen einfach einzufügen.

7.3. Ausblick

Das entwickelte Ameisen-System hat gezeigt, dass es in der Lage ist, Tourenpläne an dynamische Veränderungen anzupassen. Einer der nächsten Schritte soll es sein, die Fahrzeuge über mobile Clients an das SAP-System anzubinden und DYONISYS Informationen über ihren aktuellen Status zur Verfügung zu stellen und somit Live-Daten einzubringen. Über die mobilen Clients könnte dann auch jedem Fahrer seine aktuelle Tour mitgeteilt werden, die sich abhängig von Ereignissen ändern kann.

Zuvor muss jedoch versucht werden, der in (Kapitel 6.4, S. 56) aufgedeckten Problematik entgegenzutreten und die Qualität der von DYONISYS berechneten Lösungen zu verbessern. Als erster, vielversprechender Versuch soll die Implementierung einer Nachbarschaftsliste durchgeführt werden. Im Anschluss daran soll untersucht werden, ob ein modifiziertes Ameisenverhalten zu besseren Lösungen führen kann. Dazu sollen

die Ameisen dann in jedem Schritt prüfen, welches Fahrzeug als nächstes einen weiteren Stop zugewiesen bekommt, statt immer nur ein Fahrzeug zur Zeit zu betrachten.

Mit weiteren Szenarien aus der Praxis und mit der Hilfe von Massentests soll dann die Robustheit und Performance von DYONISYS getestet werden. Zum Zeitpunkt der Erstellung dieser Arbeit lagen leider keine hinreichend umfangreichen Testdaten für diesen Zweck vor. Selbst erdachte, womöglich praxisferne Szenarien hätten das Risiko geborgen, Ergebnisse ohne Praxisbezug zu liefern und falsche Schlussfolgerungen hervorzurufen.

DORIGO und STÜTZLE beschreiben in [DS04] einige Techniken, die Ameisen-Systemen zu einer besseren Performance verhelfen, ihre Skalierbarkeit steigern und zu qualitativ besseren Lösungen führen. In der Kürze der Zeit ist es nicht möglich gewesen, diese alle umzusetzen, es soll aber in der Zukunft geschehen. Zu diesen Techniken zählen

- Nachbarschaftslisten zur effizienteren Berechnung von Besuchswahrscheinlichkeiten, es wird jeweils nur eine Auswahl von Folgeknoten betrachtet
- Berechnen aller Besuchswahrscheinlichkeiten in der Verwaltungsphase und Speichern in einer gemeinsam verfügbaren Matrix
- Berechnung von Besuchswahrscheinlichkeiten und Evaporation nur auf Kanten, die durch die Nachbarschaftslisten charakterisiert werden

Da durch die Verwendung weniger Ameisen in kurzer Zeit Startlösungen bestimmt werden können, soll getestet werden, ob diese sinnvoll als Eingabe für die Optimierung durch ICEDG verwendet werden können. Optimalerweise könnte damit die Berechnung von ICEDG beschleunigt und verbessert werden.

A. JSON

A.1. Gemeinsame Syntax

INPUT → {"**config**":CONFIG, "**data**":DATA, "**plan**":PLAN}

CONFIG → {"**maxDeliveryEarlyness**":NUM, "**maxDeliveryDelay**":NUM
, "**maxDepotEarliness**":NUM, "**maxDepotDelay**":NUM
, "**maxWorkingTimeSoft**":NUM, "**maxWorkingTimeHard**":NUM
, "**maxWorkingTimeRest**":NUM, "**minWorkingTimeToRest**":NUM
, "**minBreakPeriod**":NUM, "**breakPeriod**":NUM
, "**facOrderEarliness**":DNUM, "**facOrderDelay**":DNUM
, "**facWorkTime**":DNUM, "**penalties**":[PENALITY]
, "**truckProfiles**":NUM, "**productCount**":NUM
, "**contingentCount**":NUM, "**loadDepotCount**":NUM
, "**truckDepotCount**":NUM, "**vehicleCount**":NUM
, "**deliveryCount**":NUM, "**wsdlLocation**":STRING
, "**username**":STRING, "**password**":STRING, "**dimaId**":NUM}

DATA → {"**products**":[PRODUCT]
, "**contingents**":[CONTINGENT]
, "**loadDepots**":[LOADDEPOT]
, "**truckDepots**":[TRUCKDEPOT]
, "**vehicles**":[VEHICLE]
, "**deliveries**":[DELIVERY]}

PENALTY → "**penalty**":{"NUM":DNUM} · (, PENALTY) | ϵ

PRODUCT → "**product**":{"**id**":NUM, "**baseProducts**":[BASEPRODUCTS]
, "**allowedProducts**":[NUMS], "**flushingCosts**":[DNUMS]}
· (,PRODUCT| ϵ)

CONTINGENT → "**contingent**":{"**id**":NUM, "**productId**":NUM, "**amount**":NUM
, "**price**":DNUM, "**start**":DATE, "**end**":DATE}
· (, CONTINGENT | ϵ)

LOADDEPOT → "**loadDepot**":{"**id**":NUM, "**geoPoint**":GEOPOINT
, "**contingents**":[NUMS,] "**waitingTime**":NUM
, "**maxLoadRate**":NUM, "**schedule**":[SCHEDULE]
, "**requirements**":[REQUIREMENT]} · (, LOADDEPOT | ϵ)

TRUCKDEPOT → "**truckDepot**":{"**id**":NUM, "**geoPoint**":GEOPOINT
, "**prepareTime**":NUM, "**parkingTime**":NUM
, "**schedule**":[SCHEDULE]} · (, TRUCKDEPOT | ϵ)

VEHICLE → "**vehicle**":{"**id**":NUM, "**transportUnits**":[TRANSPORTUNIT]
, "**truckProfile**":NUM, "**truckDepot**":NUM
, "**costsPerTime**":DNUM, "**costsPerDistance**":DNUM
, "**costsPerUsage**":DNUM, "**costsPerTrip**":DNUM
, "**maxLoadRate**":NUM, "**maxUnloadRate**":NUM
, "**allowedProducts**":[NUMS], "**equipment**":[EQUIPMENT]
, "**schedule**":[SCHEDULE], "**loadType**":LOADTYPE}
· (, VEHICLE | ϵ)

DELIVERY → "**delivery**":{"**id**":NUM, "**geoPoint**":GEOPOINT
, "**position**":[POSITION], "**requirements**":[REQUIREMENT]
, "**prio**":NUM, "**schedule**":[SCHEDULE], "**prepareTime**":NUM
, "**maxUnloadRate**":NUM} · (, DELIVERY | ϵ)

NUM → (**0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**) · (NUM | ϵ)

DNUM → NUM(.NUM | ϵ)

BASEPRODUCTS → {"NUM":DNUM} · (, BASEPRODUCTS) | ϵ

DATE → NUM.NUM.NUM

GEOPOINT → {"**x**":DNUM, "**y**":DNUM}

SCHEDULE → {"DATE":[TIMEWINDOW]} · (, SCHEDULE | ϵ)

REQUIREMENT → {"**id**":NUM, "**values**":[NUMS], "**operator**":EQUIPMENTOP}
· (, REQUIREMENT) | ϵ

EQUIPMENT → {"**id**":NUM, "**values**":[NUMS]} · (, EQUIPMENT) | ϵ

LOADTYPE → "**preload**" | "**default**"

POSITION → {"**id**":NUM, "**amount**":NUM} · (, POSITION | ϵ)

TIMEWINDOW → {"**start**":NUM, "**end**":NUM} · (, TIMEWINDOW | ϵ)

NUMS → NUM · (, NUMS | ϵ)

EQUIPMENTOP → "**eq**" | "**neq**" | "**leq**" | "**geq**"

PLAN → [DATETOUR]

DATETOUR → {"DATE":[TOURS]} · (, DATETOUR) | ϵ

TOURS → {"NUM":[TRIP]} · (, TOURS | ϵ)

TRIP → [NUMS] · (, TRIP | ϵ)

TRANSPORTUNIT → {"**id**":NUM, "**compartments**":[COMPARTMENT]}
· (, TRANSPORTUNIT | ϵ)

DNUMS → DNUM · (, DNUMS | ϵ)

COMPARTMENT → {"**id**":NUM, "**productId**":NUM, "**amount**":NUM
, "**volume**":NUM} · (, COMPARTMENT | ϵ)

A.2. Für Dyonisys angepasster Teil der Syntax

INPUT → {"**action**":ACTION}
ACTION → "**INIT**", · INITACTION
| "**MODIFY**", · MODIFYACTION
| "**SOLVE**"
| "**STATUS**"
INITACTION → "**config**":CONFIG, "**data**":DATA, "**plan**":PLAN
MODIFYACTION → ("**addDelivery**":[DELIVERY]
| "**removeVehicle**":[NUMS]
| "**trafficJam**":DNUM
| "**resetDima**":NUM
| "**ants**":NUM
| "**runs**":NUM
| "**threads**":NUM
| "**maxNoImprovement**":NUM
| "**evaporationRate**":DNUM
| "**alpha**":NUM
| "**beta**":NUM)
· (, MODIFYACTION | ϵ)
CONFIG → {"**maxDeliveryEarlyness**":NUM, "**maxDeliveryDelay**":NUM
, "**maxDepotEarliness**":NUM, "**maxDepotDelay**":NUM
, "**maxWorkingTimeSoft**":NUM, "**maxWorkingTimeHard**":NUM
, "**maxWorkingTimeRest**":NUM, "**minWorkingTimeToRest**":NUM
, "**minBreakPeriod**":NUM, "**breakPeriod**":NUM
, "**facOrderEarliness**":DNUM, "**facOrderDelay**":DNUM
, "**facWorkTime**":DNUM, "**penalties**":[PENALITY]
, "**truckProfiles**":NUM, "**productCount**":NUM
, "**contingentCount**":NUM, "**loadDepotCount**":NUM
, "**truckDepotCount**":NUM, "**vehicleCount**":NUM
, "**deliveryCount**":NUM, "**wsdlLocation**":STRING
, "**username**":STRING, "**password**":STRING, "**dimaId**":NUM
, "**optimizationDate**":DATE, "**ants**":NUM, "**runs**":NUM
, "**threads**":NUM, "**maxNoImprovement**":NUM
, "**evaporationRate**":DNUM, "**initialPheromone**":DNUM
, "**alpha**":NUM, "**beta**":NUM}

B. Eingabeparameter

B.1. Basis

Parameter	Bedeutung
maxDeliveryEarliness	Maximal erlaubte zu frühe Ankunft bei Lieferungen
maxDeliveryDelay	Maximal erlaubte Verspätung bei Lieferungen
maxDepotEarliness	Maximal erlaubte zu frühe Ankunft bei Depots
maxDepotDelay	Maximal erlaubte Verspätung bei Depots
maxWorkingTimeSoft	Weiche grenze für die Arbeitszeit der Fahrer (Arbeitszeit ohne Überstunden)
maxWorkingTimeHard	Harte Grenze für die Arbeitszeit der Fahrer (Arbeitszeit mit Überstunden)
minWorkingTimeToRest	Minimale Arbeitszeit, nach der eine Pause eingelegt werden darf
maxWorkingTimeRest	Maximale Arbeitszeit, nach der eine Pause eingelegt werden muss
minBreakPeriod	Minimaldauer einer Pause
breakPeriod	Gesamt zu nehmende Pausezeit
facOrderEarliness	Strafkosten für zu frühe Ankunft bei Lieferungen
facOrderDelay	Strafkosten für verspätete Ankunft bei Lieferungen
facDepotEarliness	Strafkosten für zu frühe Ankunft bei Depots
facDepotDelay	Strafkosten für verspätete Ankunft bei Depots
facWorkTime	Strafkosten für Überstunden
facNotDelivered	Strafkosten für nicht belieferte Aufträge
facPurchase	Strafkosten für das Einladen von Produkten
penalties	Strafkosten für die Verletzung von Anforderungen
truckProfiles	Anzahl der im Szenario vorkommenden Truckprofile
productCount	Anzahl der im Szenario vorkommenden Produkte
contingentCount	Anzahl der im Szenario vorkommenden Kontingente
loadDepotCount	Anzahl der im Szenario vorkommenden Ladedepots
truckDepotCount	Anzahl der im Szenario vorkommenden Truckdepots
vehicleCount	Anzahl der im Szenario vorkommenden Fahrzeuge
deliveryCount	Anzahl der im Szenario vorkommenden Lieferungen
wsdlLocation	URI, under dem die WSDL zum Abrufen der Distanzmartix zu finden ist
username	Benutzername zum Abrufen der Distanzmatrix
password	Passwort zum Abrufen der Distanzmatrix
dimaId	Identifikationsnummer der zu verwendenden Distanzmatrix

Tabelle B.1.: Eingabeparameter für die Basis

B.2. Dyonisys

Parameter	Bedeutung
optimizationDate	Datum, für das die Optimierung durchgeführt werden soll
ants	Anzahl zu verwendender Ameisen
runs	Anzahl zu durchlaufender Zyklen
threads	Anzahl an Threads, die für die Optimierung verwendet werden sollen
maxNoImprovement	Anzahl an Zyklen, die ohne Verbesserung ablaufen dürfen
evaporationRate	Evaporationsrate für die Pheromone
initialPheromone	Konzentration der Initialpheromone
minPheromoneConcentration	Auf den Kanten minimal erlaubte Pheromonkonzentration
alpha	Exponent zur Gewichtung der Pheromonwerte
beta	Exponent zur Gewichtung der Heurisitk

Tabelle B.2.: Eingabeparameter für Dyonisys

Abkürzungs- und Symbolverzeichnis

α	Exponent für die Gewichtung der Pheromone
β	Exponent für die Gewichtung der Heuristik
$\cdot$	Konkatenation
ϵ	Das leere Wort
$\eta_{(i,j)}$	Heuristischer Wert auf der Kante (i, j)
(i, j)	Die gerichtete Kante von Knoten i nach Knoten j
$\mid$	Option
ρ	Faktor für die Evaporation der Pheromone
τ_0	Konzentration der Initialpheromone
$\tau_{(i,j)}$	Pheromonkonzentration auf der Kante (i, j)
c	Kosten einer Lösung
c_{ant}	Kosten der Lösung einer bestimmten Ameise
c_{best}	Kosten der besten bekannten Lösung
DOT	Disposition Optimization Toolset
E	Menge von Kanten
G	Ein Graph
GB	Gigabyte
JIT	Just In Time
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MB	Megabyte
TSP	Traveling Salesman Problem
UML	Unified Modelling Language
V	Menge von Knoten
VRP	Vehicle Routing Problem

Abbildungsverzeichnis

2.1. Interaktion von TermiDe, IcedG und Dyonisys 6
2.2. Ausführung von Java-Programmen 7

3.1. Beispiel: Graph 9
3.2. Beispiel: VRP 11
3.3. Beispiel: TSP 12

5.1. UML-Diagramm zum Fahrzeug 22
5.2. UML-Diagramm zu den Orten 24
5.3. UML-Diagramm zu den Anforderungen und der Fahrzeugausrüstung . . 26
5.4. UML-Diagramm zur Struktur der Produkte 26
5.5. UML-Diagramm zum Kontrakt und den Kontingenten 27
5.6. UML-Diagramm zur Beladung der Fahrzeuge 29
5.7. UML-Diagramm zum Plan 31
5.8. UML-Diagramm zum Parser 32
5.9. UML-Diagramm zur Datenbasis und der Konfiguration 33
5.10. UML-Diagramm zur Klassenübersicht von Dyonisys 35
5.11. Bestandteile von Dyonisys und Kommunikation über ATOS 35
5.12. UML-Diagramm zu den Pheromonen. 40

6.1. Startansicht des Testtools 43
6.2. Das Testszenario 44
6.3. Plansituation vor Fahrzeugausfall 45
6.4. Situation nach Fahrzeugausfall 45
6.5. Angepasster Plan nach Fahrzeugausfall 45
6.6. Plansituation vor Hinzukommen neuer Aufträge 46
6.7. Situation mit 5 neuen Aufträgen 46
6.8. Situation, in der alle neuen Aufträge verplant wurden 46
6.9. Plansituation vor Auftreten des Staus 47
6.10. Situation, die sich durch den Stau ergibt 47
6.11. Neuplanung nach Stau 47
6.12. Großes Szenario 49
6.13. Entwicklungen bei Variation der Ameisenanzahl 50
6.14. Entwicklungen bei Variation der Zyklenzahl 51
6.15. Entwicklungen bei Variation der Evaporationsrate 52
6.16. Entwicklungen bei Variation der Initialpheromone 53
6.17. Entwicklungen bei Variation der Pheromongewichtung 54
6.18. Entwicklungen bei Variation der Heuristikgewichtung 55

6.19. Vergleich der Ergebnisse von DYONISYS und ICEDG 56
6.20. Eine Lösung von DYONISYS zu s_{klein} 56
6.21. Eine Lösung von ICEDG zu s_{klein} 57
6.22. Eine Lösung von DYONISYS zu s_{gross} 57
6.23. Eine Lösung von ICEDG zu s_{gross} 57

Tabellenverzeichnis

4.1. Parameter für die Optimierung mit einem Ameisen-System 18

5.1. Beispielausrüstung eines Fahrzeugs mit den Schlauchlängen 5m und 10m 25
5.2. Anforderung an die Schlauchlänge: Sie soll mindestens 5m betragen. . . 25

6.1. Verwendete Referenzkonfiguration 48
6.2. Messungen bei Variation der Ameisenanzahl 50
6.3. Messungen bei Variation der Zyklenzahl 51
6.4. Messungen bei Variation der Evaporationsrate 52
6.5. Messungen bei Variation der Initialpheromone 53
6.6. Messungen Variation der Gewichtung der Pheromone 54
6.7. Messungen bei Variation der Gewichtung der Heuristik 55

B.1. Eingabeparameter für die Basis 64
B.2. Eingabeparameter für Dyonisys 65

Literaturverzeichnis

[Beu07] A. Beutelspacher. *Diskrete Mathematik für Einsteiger. Mit Anwendungen in Technik und Informatik.* Vieweg, 2007.

[Co00] S. A. Cook. *The P versus NP Problem.* Manuscript prepared for the Clay Mathematics Institute for the Millennium Prize Problems, April 2000.

[DDG99] M. Dorigo, G. Di Caro and L. M. Gambardella. *Ant Algorithms for Discrete Optimization.* Artificial Life, Spring 1999, Vol. 5, No. 2, Pages 137-172.

[DS04] M. Dorigo and T. Stützle. *Ant Colony Optimization.* The MIT Press, 2004.

[GHJV04] E. Gamma, R. Helm, R. E. Johnson, J. Vlissides. *Entwurfsmuster - Elemente wiederverwendbarer objektorientierter Software.* Addison Wesley, München 2004.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness.* Freeman, New York, 1979.

[Ka72] R. M. Karp. *Reducibility Among Combinatorial Problems.* In Raymond E. Miller and James W. Thatcher. Complexity of Computer Computations. New York: Plenum. pp. 85–103.

[LLRS85] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. *The Traveling Salesman Problem.* John Wiley & Sons, 1985.

[Tu09] V. Turau. *Algorithmische Graphentheorie.* Oldenbourg, 2009.

[Sip06] M. Sipser. *Introduction to the Theory of Computation.* Thomson, 2006.

[Tomcat] `http://tomcat.apache.org/`

[Vaz01] V. V. Vazirani. *Approximation Algorithms.* Springer, 2001.

[VV86] L. G. Valiant and V. V. Vazirani. *NP is as easy as detecting unique solutions.* Proceedings of the seventeenth annual ACM symposium on Theory of computing, 1985, S. 458–463.

[Wan06] R. Wanka. *Approximationsalgorithmen - Eine Einführung.* Teubner, 2006.

Index

A
ABAP 7
Abbruchkriterien 17
Abbruchkriterium 39
Abgrenzung 2
Abstrakte Fabrik 30
Ameise 34, 37
Ameisen-Rangfolge 16
Ameisen-System 14, 34, 37
Ameisen-Systemen 3
Anfangsbeladung 23
Anforderung 24, 25
Approximationsverfahren 12, 20
ATOS 4, 31, 40
Ausblick 59
Ausführungs-Thread 37
Ausrüstung 22, 24

B
Basis 21
Basisprodukt 26
Befehl 28
Beladerate 23, 24
Beladung 27, 28
Best Fit 29
Bestellung 23
Besuchswahrscheinlichkeit . . . 15, 18, 38
Bytecode 7

C
C++ 7

D
Datenbasis 33
DOT 6, 31
Dynamik 40
Dyonisys 5, 44, 56

E
Einführung 1
Eingabeparameter 64
Elite-Ameise 16
Emergenz 14
Entladerate 23
Entwurf 21
Entwurfsmuster 21
Erweiterungen 16
Evaporation 15, 19, 39
Evaporationsrate 52

F
Fahrzeug 22
Fahrzeugausfall 41
Fahrzeugdepot 23
Fazit 59
Fliegengewicht 25
Futterquelle 14

G
Gewinn 1
Graph 9, 39
Graphentheorie 9
Grundlagen 9

H
Heuristik 15

I
IcedG 5, 42, 56
Implementierung 21
Initialisierungsphase 15, 18
Initialpheromone 15, 18, 53
Interpretierer 32

J
Java 7, 58, 59
Java Virtual Machine 7

JSON 31
JVM 34, 49

K
Kante 9
Knoten 9
Kommunikation 35
Komplexitätstheorie 10
Konfiguration 33
Konstruktionsphase 15, 18, 40, 41
Kontingent 23, 27
Kontrakt 27

L
Lösungskandidat 15, 18, 37
Lösungsraum 13
Ladedepot 23
Lieferplan 30
Liefertour 1
Lieferung 23
Lokale Suche 17

M
Matching 25
Motivation 1

N
Nachbarschaftsliste 40
Nest 15, 18

O
Optimierungs-Thread 36

P
Parallelisierungsgrad 37
Parser 31, 40
Performance 60
Pheromon 14, 39
Pheromonaktualisierung 17
Pheromonbedingungen 17
Pheromonkonzentration 14, 15, 41
Problembeschreibung 2
Produkt 26
Proxy 34

R
Reinigung 27

S
SAP 4, 31, 40, 59
Scanner 32
Servlet 36
Singleton 25
Skalierbarkeit 60
Stagnation 15, 36, 39
Stop 30
Strategie 25

T
Tabu-Suche 5
TermiDe 4
Test 42
Testtool 43
Tomcat 34, 43
Tour 30
Transporteinheit 22, 23
Transportkammer 22
Traveling Salesman Problem 12
Trip 30
Truckprofil 23

U
Umfeld 4
UML-Diagramm 21

V
Varianten 16
Vehicle Routing Problem 11
Verwaltungsphase 15, 18, 19, 40
Vordisposition 4

W
Webserver 34
Webservice 34
Whitelist 26, 28

X
XML 31
xServer 6, 41, 42

Z
Zeitfenster 2, 42
Zeitplan 2, 22, 23
Zielsetzung 3
Zusammenfassung 58
Zyklus 15, 19, 41, 51

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Hamburg, den 01. September 2011 Christopher Blöcker