

Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik

Integration von Geodaten in ein Planungssystem

Bachelorarbeit

Leipzig, 9. November 2012

vorgelegt von

Björn Buchwald

Studiengang B. Sc. Informatik

Betreuer Hochschullehrer: Prof. Dr. Klaus-Peter Fähnrich
Fakultät für Mathematik und Informatik/Institut für Informatik/
Abteilung Betriebliche Informationssysteme

Inhaltsverzeichnis

Abbildungsverzeichnis	IV
Abkürzungsverzeichnis.....	VI
1. Einleitung.....	1
1.1 Problemdarstellung	1
1.2 Methodisches Vorgehen.....	2
1.3 Aufbau der Arbeit	3
2. Planung und Konzeption.....	4
2.1 Anforderungen.....	4
2.2 Konkurrenzprodukte und Dienstleister	6
2.3 Einordnung in den wissenschaftlichen Kontext	7
2.3.1 Vergleich des Travelling Salesman Problems mit dem Dinner-Hopping Problem	
7	
2.3.2 Heuristiken	12
2.4 Recherche und Entscheidungsfindung	14
2.4.1 Verwendete Technologien und Entwicklungswerkzeuge	14
2.4.2 Diskussion von Diensten und deren Bewertung.....	16
3. Modellierung	23
3.1 Datenfluss.....	23
3.2 Use Cases	25
3.3 Klassenbeschreibung.....	26
3.3.1 Fachkonzeptklassen	26
3.3.2 Benutzeroberfläche	30
4. Implementierung	32
4.1 Routenplanungsalgorithmus	32
4.2 Umsetzung in Java	34
4.2.1 Heuristiken	34

4.2.2	Benutzeroberfläche	36
4.2.3	Kartendarstellung mit dem JXMapKit	39
4.2.4	Realisierung der Programmeigenschaften.....	40
4.2.5	Parsen einer Antwort des CloudMade-Geocodierers.....	41
4.2.6	Distanzbestimmung mit Osm2po	42
4.3	Beispieldatensatz	44
5.	Diskussion	48
5.1	Evaluation.....	48
5.1.1	Korrektheit des Routenplanungsalgorithmus	48
5.1.2	Laufzeiten des Routenplanungsalgorithmus	51
5.1.3	Berechnete Distanzen des Routenplanungsalgorithmus	54
5.2	Kritik	57
5.2.1	Annahmen zum Routenplanungsalgorithmus	57
5.2.2	Bestimmung der Hauptspeiseorte	58
5.3	Fazit.....	59
5.4	Ausblick	60
	Kurzzusammenfassung	61
	Literaturverzeichnis.....	62
	Anlagenverzeichnis	64
	Erklärung	65

Abbildungsverzeichnis

Abbildung 1 Es wird das Antwortzeitverhalten der Dienst für die Distanzbestimmung beschrieben.....	20
Abbildung 2 Es werden die Unterschiede der Berechneten Distanzen der Dienste für die Distanzbestimmung veranschaulicht.....	21
Abbildung 3 Komponenten des Planungssystems und Datenaustausch dieser.....	24
Abbildung 4 Use Case Diagramm des Planungssystems.....	25
Abbildung 5 Fachkonzeptklassen.....	27
Abbildung 6 Beschreibung des Einlesens einer CSV-Datei ohne Geokoordinaten mit entsprechender Geocodierung.....	29
Abbildung 7 Beschreibung der Distanzbestimmung.....	30
Abbildung 8 Klassen der Benutzeroberfläche.....	31
Abbildung 9 Dinner-Hopping Planner (links) und Viewer (rechts).....	37
Abbildung 10 „Properties“-Fenster mit Tabs für Input-Datei und Geocodierung.....	38
Abbildung 11 Datensatz mit neun Teams ohne Geokoordinaten.....	44
Abbildung 12 Input und Geocodierung.....	45
Abbildung 13 Distanzmatrix und Anzahl der gefundenen Lösungen (links). Das GUI zeigt Resultat fünf als Minimum mit der kleinsten Gesamtdistanz (rechts).....	46
Abbildung 14 Routen von Vorspeiseteams (oben links), Hauptspeiseteams (unten) und Nachspeiseteams (oben rechts).....	46
Abbildung 15 Ausgabe in der Konsole mit einer Route für jedes Team	47
Abbildung 16 Auswahl der Routen für die Teams 0, 5 und 3	47
Abbildung 17 Laufzeiten des Routenplanungsalgorithmus in Abhängigkeit von Parameter „changeMainCourseForRecursion“ für unterschiedliche Teamanzahlen.....	51
Abbildung 18 Laufzeit des Routenplanungsalgorithmus in Abhängigkeit von Parameter „solutions“ für unterschiedliche Teamanzahlen.....	52
Abbildung 19 Laufzeit des Routenplanungsalgorithmus in Abhängigkeit von Parameter „recursionDepth“ für unterschiedliche Teamanzahlen	53
Abbildung 20 Gesamtdistanz in Abhängigkeit der Teamanzahlen für unterschiedliche Konfigurationen der Parameter.....	54
Abbildung 21 Durchschnittliche Distanz eines Teams in Abhängigkeit der Teamanzahlen für unterschiedliche Konfigurationen.....	55

Abbildung 22 Durchschnittliche Distanz eines Teams in Abhangigkeit der Teamanzahlen fur unterschiedliche Konfigurationen der Parameter. Die Parameter wurden, fur die hier betrachteten kleineren Teamanzahlen, angepasst.	56
Abbildung 23 Differenz zwischen groter und kleinster ermittelter Gesamtdistanz pro Team fur die oben erwahnten Konfigurationen.	56
Abbildung 24 Varianz in den bestimmten Gesamtdistanzen fur kleinere Teamanzahlen. Es liegen die in Abbildung 22 definierten Konfigurationen zugrunde.	57

Abkürzungsverzeichnis

API	Application Programming Interface
CSV	Comma-Separated Values
DHP	Dinner-Hopping Problem
GIS	Geoinformationssystem
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
ID	Identifier
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
LVB	Leipziger Verkehrsbetriebe
mTSP	multiple Travelling Salesman Problem
MVC	Model-View-Controller
OSM	OpenStreetMap
REST	Representational State Transfer
SQL	Structured Query Language
SwingX	Swing Extensions
SWT	Standard Widget Toolkit
TSP	Travelling Salesman Problem
TSSP	Travelling Salesman Selection Problem
URL	Uniform Resource Locator
WILMA	Willkommens-Initiative für in Leipzig mitstudierende Ausländer
XML	Extensible Markup Language

1. Einleitung

1.1 Problemdarstellung

Die vorliegende Arbeit beschäftigt sich mit der Konzeption einer prototypischen Softwarelösung, welche durch die Integration von Geodaten ein Planungssystem zur Routenfindung zwischen verschiedenen hintereinander stattfindenden Veranstaltungsorten ermöglicht.

Geodaten werden hier als geografische Daten, das heißt Informationen zu Adressen, wie geografische Lage in Breiten- und Längengrad, die Entfernung zwischen zwei Adressen sowie Informationen über die Route, also den Weg von einer Adresse zu einer anderen, aufgefasst. Diese Daten sollen über verschiedene Services, wie Internetdienste oder lokale Datenquellen integriert werden.

Um den Ablauf einer solchen Veranstaltung und damit die genaue Problemstellung zu verdeutlichen, wird im Folgenden der konkrete Anwendungsfall näher erläutert.

Das „Dinner-Hopping“, eine Veranstaltung der „Willkommens-Initiative für in Leipzig mitstudierende Ausländer (WILMA)“ findet in regelmäßigen Abständen statt und dient der Erleichterung der Kontaktaufnahme Leipziger Austauschstudenten untereinander und zu deutschen Studenten. Dabei bewirken die Studenten in Zweierteams in ihrer Wohnung für einen Gang eines Drei-Gänge-Menüs vier andere Studenten. Bei den beiden anderen Gängen sind sie zu Gast bei ihren Kommilitonen. Die Herausforderung besteht darin, jedem Team möglichst kurze Wege von einem Gang bzw. Ort zum nächsten zu garantieren. Dabei muss jedes Team genau einmal selbst einen Gang in der eigenen Wohnung anrichten. Des Weiteren darf es keine doppelten Begegnungen der Teams während der einzelnen Gänge geben.

Zurzeit wird die Planung dieser Veranstaltungen per Hand erledigt, was ein sehr zeitaufwendiges Unterfangen darstellt, wenn man bedenkt, dass bis zu 39 Teams teilnehmen und die beschriebenen Nebenbedingungen erfüllt sein müssen. Um diesen Vorgang deutlich zu verkürzen, soll der in dieser Arbeit beschriebene Prototyp konzipiert werden. Da solche Veranstaltungen, auch als „Running-Dinner“ oder „Kitchen Run“ bekannt, in anderen Städten und von anderen Veranstaltern ebenfalls mit großem Interesse und hohen Teilnehmerzahlen durchgeführt werden, ist die Problemlösung für diese ebenfalls interessant und durchaus auf verschiedene Veranstaltungsorte übertragbar. Eine Recherche zu Konkurrenzprodukten bzw. ähnlichen Veranstaltungen findet sich in Abschnitt 2.2. Die Grobstruktur der Softwarelösung lässt sich folgendermaßen beschreiben:

Nach dem Import von Teilnehmerdaten und Veranstaltungsorten soll der Veranstalter durch das System bei der Routenplanung unterstützt werden. Ziel ist es, dem Veranstalter Routenplanungen zu liefern, wobei der Schwerpunkt auf möglichst kurzen Wegen zwischen den Veranstaltungsorten liegt. Des Weiteren sollen Schnittstellen für Visualisierung und Export der Routenplanung durch das Produkt zur Verfügung gestellt werden.

1.2 Methodisches Vorgehen

I. Anforderungs- und Machbarkeitsanalyse

Zu Beginn wird eine Risikoanalyse zum Aufdecken möglicher Probleme und Bedenken angefertigt. Es folgt eine erste Recherche zur Einarbeitung in die Thematik. Es sollen konkrete inhaltliche Vorstellungen über das Thema entwickelt werden. Identifiziert werden sollen hierbei mögliche Anbieter von geografischen Daten, Schnittstellen, die diese bereitstellen, sowie Auswertungsmöglichkeiten von zurückgelieferten Informationen. Auch rechtliche Aspekte, wie Nutzungseinschränkungen der verschiedenen Dienste, sind von Bedeutung. Außerdem wird für die Routenplanung ein Algorithmus für mögliche Routenkombinationen und deren Optimierung benötigt. Hier sind mögliche wissenschaftliche Lösungsansätze zu suchen. Dies beinhaltet in erster Linie die Einarbeitung in Konzepte und Terminologie des Themas. Die Recherche wird durch das Suchen von Konkurrenzprodukten und gegebenenfalls die Analyse und Abgrenzung zum eigenen Produkt erweitert. Eventuell besteht die Möglichkeit Anwendungen, die Teillösungen bereitstellen, zu finden, welche für die eigene Lösung hilfreich sind. Ziel ist der Recherchebericht.

Begleitend wird ständiger Kontakt mit dem Kunden gehalten, um das Umfeld des Auftrags sowie fachliche und technische Anforderungen aufzugreifen. Daraufhin findet eine erste Anforderungsanalyse statt, auf deren Grundlage Glossar und Lastenheft erstellt werden. Außerdem wird der Projektplan (Vorgehensbeschreibung) mit Terminplanung erstellt. In einem Kundengespräch wird das Lastenheft diskutiert und verfeinert, um schließlich das Pflichtenheft erstellen zu können. Dieses stellt den Abschluss der Anforderungsanalyse dar. Der Aufbau der Dokumente richtet sich stets nach [1].

II. Entwicklung geeigneter Verfahren

Zunächst werden die zu benutzenden Technologien und Programmiersprachen bewertet. Auf Grundlage der verwendeten Programmiersprachen wird ein Dokumentationskonzept erstellt.

Außerdem wird die Problemstellung, in diesem Fall die Entwicklung eines Algorithmus, für die Routenplanung analysiert und mögliche wissenschaftliche Probleme und Optimierungen diskutiert, die dem gegebenen Problem ähnlich sind. Schließlich werden auf das Problem übertragbare Aspekte für die Lösung übernommen. Auf Grundlage der Anforderungen, insbesondere angestrebter Qualität und Produktleistungen, werden die geeigneten Lösungen ausgewählt.

III. Entwurfsbeschreibung

Das zu konstruierende Softwareprodukt wird durch geeignete Modelle, wie Paket-, Klassen- und Sequenzdiagramme beschrieben. Damit wird der Datenfluss zwischen einzelnen Klassen, sowie die Umsetzung einzelner Anwendungsfälle definiert. Die Notation orientiert sich an [1]. Die Entwurfsbeschreibung liefert eine genaue Vorlage für die Implementierung.

IV. Implementierung des Prototyps

Es werden alle Mussfunktionen des Pflichtenheftes auf Grundlage der ausgewählten Verfahren realisiert. Das Softwareprodukt wird während der Entwicklung anhand eines Beispieldatensatzes des Kunden auf Funktionsfähigkeit getestet.

V. Evaluation

Zunächst wird die Güte des eingesetzten Verfahrens diskutiert, sowie die Korrektheit des Systems geprüft und dokumentiert. Außerdem werden Funktionen betrachtet, die aufgrund zeitlicher Restriktionen oder des Aufwandes nicht umgesetzt werden konnten. Auf deren Umsetzung in folgenden Projekten wird schließlich eingegangen.

1.3 Aufbau der Arbeit

Nach der einleitenden Beschreibung des Problems werden in Abschnitt 2 „Planung und Konzeption“ sowie bisherige Arbeiten zum vorliegenden Problem diskutiert. Außerdem erfolgen das Beschreiben der Anforderungen sowie die Formulierung der wissenschaftlichen Problemstellung und Grundlagen. Schließlich müssen geeignete Verfahren und Dienste ausgewählt werden.

Die Unterteilung des Planungssystems in einzelne Komponenten wird in Abschnitt 3 vorgenommen. Anhand dieser erfolgt eine Einteilung in Klassen und Beschreibung von deren Beziehungen durch einige Beispiele.

Der folgende Abschnitt geht auf die genaue Umsetzung des modellierten Systems ein. Zunächst wird der Routenplanungsalgorithmus verbal beschrieben. Daraufhin folgen Beispiele, die zeigen sollen, in welcher Weise einzelne Bestandteile des Planungssystems in der Programmiersprache Java umgesetzt wurden. Schließlich wird die Arbeitsweise des Systems an einem Beispieldatensatz gezeigt.

Abschnitt 5 beschäftigt sich hauptsächlich mit der Evaluation und Kritik des Routenplanungsalgorithmus. Laufzeiten und Distanzen werden an verschiedenen Konfigurationen des Algorithmus verdeutlicht. Schließlich werden die umgesetzten Produktfunktionen zusammengefasst und ein Ausblick über mögliche Verbesserungen und zusätzliche Funktionen gegeben.

Die während des Entwicklungsprozesses entstandenen Dokumente wie Pflichtenheft, Lastenheft, Recherchebericht usw. sind in Anhang A (CD) zu finden. Der ausführbare Prototyp wird zur Verfügung gestellt. Weitere Materialien sind dem Anlagenverzeichnis zu entnehmen.

2. Planung und Konzeption

2.1 Anforderungen

Die essenziellen Anforderungen an die Software sind im Weiteren beschrieben. Die genaue Beschreibung ist dem Pflichtenheft im Anhang zu entnehmen. Die Anforderungen sind in Klammern analog ihrer Beschriftung im Pflichtenheft gekennzeichnet.

Zunächst erfolgt der Import von Teilnehmerdaten aus einer von der WILMA zur Verfügung gestellten „Comma-Separated Values (CSV)“-Datei (/F10/). Hier sind vor allem die angegebenen Adressen wichtig. Diese müssen im Folgenden geocodiert werden (/F30/). Unter der Geocodierung versteht man das Umwandeln einer Adresse in Breiten- und Längengrad. Dies ermöglicht die Berechnung der kürzesten Entferungen zwischen den einzelnen Veranstaltungsorten (/F100/). Auf deren Grundlage erfolgt die Konzeption möglicher Routen der einzelnen Teams über die jeweils drei Veranstaltungsorte (drei Gänge) mit anschließender Filterung der Ergebnisse nach ausgewählten Kriterien, wobei die folgenden Nebenbedingungen (1) bis (8) erfüllt sein müssen (/F110/).

- (1) Ein Schwerpunkt der Routenberechnung liegt auf der Bestimmung kurzer Routen zwischen den drei Gängen bevorzugt für Teams, welche die Hauptspeise anrichten. Die Begründung liegt in der Tatsache, dass dieser Gang potenziell den meisten Aufwand für Vor- und Nachbereitung verursacht. Die Anfahrt zum ersten Gang bzw. die Abreise vom letzten Gang können vernachlässigt werden.
- (2) Es darf keine doppelten Begegnungen der Teams geben.
- (3) Der Gang des Gastes muss mit dem Gang des Gastgebers übereinstimmen. Richtet Team A beispielsweise den Hauptspeisegang aus und Team B ist bei diesem zu Gast, dann muss Team B für diesen Gang für die Hauptspeise eingeteilt sein.
- (4) Jedes Team richtet genau einen Gang selbst an.
- (5) Der Veranstaltungsort dieses Ganges ist der Heimatort des jeweiligen Teams.
- (6) Der Gastgeber bewirbt jeweils genau zwei weitere Teams.
- (7) Ein Team muss jeden Gang (Vor-, Haupt- und Nachspeise) genau einmal durchlaufen.
- (8) Bei den beiden restlichen Gängen ist es bei einem anderen Team zu Gast.

Schließlich müssen die optimierten Routen der einzelnen Teams auf einer oder mehreren Karten visualisiert werden (/F80/). Dabei ist die Anzeige auf statischen Karten ausreichend. Der Export der optimierten Routenplanung erfolgt ähnlich des Imports in einem vorgegebenen CSV-Format (/F130/), um dem Veranstalter die Weiterverarbeitung der Ergebnisse bzw. die Integration der Software in bestehende Systeme zu ermöglichen.

Als Wunschkriterium zu betrachten, ist die Unterstützung des E-Mail-Versandes der Routen an die einzelnen Teams (/FW140/) sowie die Berücksichtigung zusätzlicher Informationen der Teilnehmer bei der Routenplanung, wie bevorzugtes Anrichten eines Ganges oder Berücksichtigung von Essgewohnheiten.

Die Softwarelösung soll weder ausführliche Routeninformationen, also den Weg mit aufeinanderfolgenden Straßennamen bis zum Ziel, liefern, noch das Finden der optimalen Lösung für die Routenplanung garantieren.

Die Bereitstellung von Geodaten, wie Geokoordinaten von Adressen, wird über Internetdienste bzw. lokal installierte Services realisiert. Hierbei werden besonderer Wert auf eine möglichst lange Verfügbarkeit der Dienste und die einfache Erweiterbarkeit der Softwarelösung gelegt. Die Benutzeroberfläche muss intuitiv bedienbar, d. h. übersichtlich und unkompliziert, sein. Des Weiteren soll das Produkt die Daten von bis 39 Teams verarbeiten können.

Die Produktdaten umfassen damit die Teilnehmerdaten, die geocodierten Adressen, Entfernung der Veranstaltungsorte zueinander, Zwischenergebnisse der Routenberechnung und die finalen Routenplanungen. Die Technischen Anforderungen werden im Abschnitt 2.4.1 “Verwendete Technologien und Entwicklungswerkzeuge“ erläutert.

2.2 Konkurrenzprodukte und Dienstleister

Das „Dinner-Hopping“ wurde bereits von anderen Hochschulen und Organisationen als durchaus gute Methode zum Kennenlernen und zum Kontaktknüpfen erkannt. Beispiele sind im Nachfolgenden an der Universität Ilmenau und Mannheim bzw. mit der Organisation „RudiRockt.de“ gegeben. Außerdem ist das „Dinner-Hopping“-Problem als Planungsproblem interessant. Aus diesen Gründen ist es sinnvoll zu untersuchen, ob bereits Ansätze einer Lösung des Problems existieren.

An der TU Ilmenau veranstaltet die „Initiative Solidarische Welt Ilmenau“ ein „Kitchen Run“¹. Nach einer E-Mail-Anfrage wurden einige Eckdaten der dort verwendeten Software zugesendet. Sie verfügt über ein Nutzer-Interface zur Eingabe von Teilnehmerdaten und ein Administrator-Interface zur Verwaltung und Auslosung der Teams. Die Software basiert auf dem Ajax Magic Framework. Die Auslosung der Gänge zu den einzelnen Teams geschieht aufgrund der geäußerten Wünsche und ohne Beachtung der Adressen oder Distanzen zwischen diesen. Die Grundlage der zu erstellenden Lösung soll jedoch eine optimierte Routenberechnung anhand der Entferungen siehe Nebenbedingung (1) zwischen den Adressen sein. Damit besitzt die zu entwickelnde Software eine andere Problemstellung.

„RudiRockt.de“² ist eine „Running-Dinner“-Organisationsfirma. Der Kunde (WILMA) möchte jedoch unabhängig bleiben, sodass diese Lösung ebenfalls nicht akzeptabel wäre.

„Flying-Dinner.com“³ bietet ein kommerzielles Online-System an. „RudiRockt“ wird als Referenz-Partner genannt. WILMA arbeitet jedoch nicht kommerziell und möchte eine Open Source Lösung.

Der Verein internationaler Studentenpatenschaften der Universität Mannheim beschreibt in einem Artikel⁴ eine Softwarelösung für deren „Running-Dinner“, die der hier betrachteten ähnelt. Es wird beschrieben, dass die Entfernung zwischen zwei Orten eingeplant wird und doppelte Begegnungen verhindert werden. Eine Anfrage wurde jedoch nicht beantwortet, sodass keine weiteren Untersuchungen in diese Richtung möglich waren.

Das „Running-Dinner“ als Planungsproblem wurde an der Dualen Hochschule Stuttgart im Jahr 2006/2007 als Studienarbeitsthema⁵ ausgeschrieben. Allerdings ist hier nicht ersichtlich, ob das Thema bearbeitet wurde. Des Weiteren sollte die Lösung über ein Agentensystem erfolgen, was einen anderen Lösungsansatz als den in dieser Arbeit betrachteten darstellt.

1 <http://www.iswi.org/index.php?id=41> (10.10.2012) Infos zum „Kitchen Run“ an der TU Ilmenau

2 <http://www.rudirockt.de/> (10.10.2012)

3 <http://flying-dinner.com/> (10.10.2012)

4 http://www.unbeschreiblich-weiblich.net/Aktuell/Schlemmen_mit_Unbekannt.pdf (21.05.2012)

5 <http://www.lehre.dhbw-stuttgart.de/~reichard/index.php?site=newearbeiten> (21.05.2012)

Zusammenfassend lässt sich sagen, dass für den Kunden keine zufriedenstellende Lösung für sein Problem gefunden werden konnte. Aus diesem Grund findet im Folgenden die Entwicklung einer Softwarelösung statt.

2.3 Einordnung in den wissenschaftlichen Kontext

2.3.1 Vergleich des Travelling Salesman Problems mit dem Dinner-Hopping Problem

Die sich aus der obigen Einführung in die Thematik ergebende wissenschaftliche Problemstellung soll in diesem Abschnitt genauer erläutert werden. Dabei wird hier weniger auf die Integration von Geodaten eingegangen, sondern vielmehr die wissenschaftliche Grundlage des sich ergebenden Problems beschrieben.

Es geht vor allem um die Herausforderung, jedem Team eine Route, bestehend aus drei Veranstaltungsorten, zuzuordnen. Die einzuhaltenden Nebenbedingungen sind in Abschnitt 2.1 definiert. Ein Veranstaltungsort entspricht einer Adresse. Durch die Geocodierung wird die Adresse in Breiten- und Längengrad umgewandelt und auf diese Weise ein Veranstaltungsort definiert.

Als Entfernung zwischen Veranstaltungsorten wird die reine Luftlinie als ausreichend betrachtet. Es wird allerdings angestrebt, die kürzeste auf Verkehrswegen beruhende Strecke ebenfalls zu berücksichtigen. Auf die Umsetzung der Entfernungsbestimmung wird in Abschnitt 2.4.2 eingegangen. Im Folgenden ist ein Abriss des detaillierten Ablaufs der Veranstaltung gegeben.

Es beginnen jeweils drei Teams an einem Vorspeiseort. Das an diesem Ort beheimatete Team richtet die Vorspeise an. Jedes Team macht sich danach auf den Weg zum Hauptspeiseort inklusive des Hauptspeiseteams. Dabei dürfen sich die sich dort treffenden Teams nicht bereits am Vorspeiseort gesehen haben. Es sei noch einmal darauf hingewiesen, dass sich Hauptspeiseteams zu ihrem Hauptspeiseort bewegen müssen und nicht bereits an diesem warten können, da sie von einem Vorspeiseort aus starten müssen. In analoger Weise wird dies mit den Nachspeiseorten bzw. Teams fortgeführt.

Eine teilweise Formalisierung des Problems ist im Folgenden gegeben.

Gegeben ist eine Menge von Teams T . Da die Nebenbedingungen (7) und (8) für jedes Team gelten, wird von der Annahme ausgegangen, dass die Anzahl der Teams durch drei teilbar ist. Um jedem Team eine konsistente Route zuordnen zu können, muss die Anzahl der Teams größer gleich 9 sein.

$$T = \{ t \mid t \text{ ist ein Team} \wedge \forall t_1, t_2 \in T: t_1 \neq t_2 \} \text{ und } |T| \bmod 3 = 0 \text{ und } |T| \geq 9 \quad (2.1)$$

Außerdem wird die Menge der Veranstaltungsorte O definiert

$$O = \{ o \mid o \text{ ist ein Ort} \} \quad (2.2)$$

und es existiert eine surjektive Abbildung siehe (2.3) bzw. (2.4).

$$f: T \rightarrow O, d. h. \quad (2.3)$$

$$\forall o \in O: \exists t \in T: f(t) = o \quad (2.4)$$

Demnach können zwei Teams am selben Ort einen Gang ausrichten. Es gilt weiterhin, dass jedem Team genau ein Heimatort o zugeordnet wird.

Es existieren drei Gänge:

$$G = \{Vorspeise, Hauptspeise, Nachspeise\} \quad (2.5)$$

Jedem Team wird ein Gang zugeteilt, welchen es selbst anrichtet. Diese Zuteilung ist im Vorfeld nicht bekannt und erfolgt während der Routenplanung. Es ergibt sich die surjektive Abbildung:

$$g: T \rightarrow G \quad (2.6)$$

Dementsprechend werden folgende Definitionen festgelegt:

Ein Veranstaltungsort an dem die Vor-, Haupt- bzw. Nachspeise stattfindet, wird als Vor-, Haupt- bzw. Nachspeiseort bezeichnet. Die anrichtenden Teams sind entsprechend Vor-, Haupt- bzw. Nachspeiseteams.

Es ist darauf zu achten, dass zwei Teams, die am selben Ort beheimatet sind, nicht denselben Gang zugeteilt bekommen. Dies ist darauf zurückzuführen, dass zwei Teams nicht zur selben Zeit am selben Heimatort anrichten können.

$$\forall t_1, t_2 \in T: f(t_1) = f(t_2) \rightarrow g(t_1) \neq g(t_2) \quad (2.7)$$

Gegeben ist weiterhin eine symmetrische $n \times n$ Entfernungsmatrix

$$M = m_{i,j} = m_{j,i}; i, j \in \{1, \dots, n\} \quad (2.8)$$

Mit $m_{i,j}$ als Entfernung von Ort i zu Ort j .

Da ein Ort eine Adresse einer Stadt beschreibt, sind die folgenden Bedingungen erfüllt.

Es existiert immer eine Verbindung zwischen zwei Orten, sodass

$$\forall i, j : 0 < m_{i,j} < \infty; i, j \in \{1, \dots, n\} \quad (2.9)$$

Die Dreiecksungleichung ist erfüllt.

$$m_{i,j} \leq m_{i,k} + m_{k,j}; i, j, k \in \{1, \dots, n\}; i \neq j, j \neq k, k \neq i \quad (2.10)$$

Dieses Problem wird im Weiteren als „**Dinner-Hopping Problem (DHP)**“ bezeichnet.

Annahmen, die speziell im Lösungsansatz der Routenplanung der prototypischen Softwarelösung getroffen werden und nicht mit den hier beschriebenen übereinstimmen, sind in Abschnitt 5.2.1 „Annahmen zum Routenplanungsalgorithmus“ erläutert.

Es lassen sich einige grundlegende Merkmale des DHP erkennen:

- Es sind verschiedene Orte und deren Entfernung gegeben.
- Es existieren verschiedene Personen, die sich zwischen den Orten von einem Start- zu einem Zielort bewegen.
- Die zurückgelegte Strecke pro Person soll minimal sein.
- Städte werden von einer Person nur einmal besucht.

Auf Grundlage dieser Eigenschaften lassen sich einige Ähnlichkeit zum „Travelling Salesman Problem (TSP)“ bzw. zu Abwandlungen von diesem erkennen. Im Folgenden werden Gemeinsamkeiten und Unterschiede zwischen DHP und TSP aufgezeigt. Außerdem wird untersucht, ob das TSP sowie Lösungsansätze von diesem auf das DHP und dessen Lösung übertragbar sind.

Travelling Salesman Problem (TSP)

Die Definition des TSP im Städtekontext lautet:

„Das TSP oder auch **Problem des Handlungsreisenden** sucht eine Rundreise über n gegebene Städte. Jede Stadt wird genau einmal besucht. Die zurückgelegte Gesamtstrecke soll minimal sein.“ [2, S. 96]

Die allgemeine Formulierung des Problems ist folgende.

„Ein gerichteter stark zusammenhängender Graph $G = (N, A)$ ist gegeben. N ist die Menge der Knoten $1, 2, \dots, n$ ($Knoten \equiv Stadt$) und A ist die Menge der gerichteten Kanten ($\equiv Städteverbindungen$), wobei jeder Kante (i, j) Kosten $c_{i,j}$ zugeordnet werden. Gesucht ist ein gerichteter Hamilton-Kreis (Hamilton-Zyklus) Z von G mit minimalen Gesamtkosten

$$c(Z) := \sum_{(i,j) \in Z} c_{i,j} \quad (2.11)$$

.“ [3, S. 248] „Dabei ist ein Hamilton-Kreis eine Abfolge von Knoten und verbindenden Kanten (abwechselnd), die alle Knoten aus G genau einmal betrachtet.“ [3, S. 144]. c ist analog der oben definierten Entfernungsmatrix M mit den Werten $m_{i,j}$ zu betrachten.

Sofort lassen sich Gemeinsamkeiten zu den oben beschriebenen Merkmalen feststellen. Sowohl Orte als auch Entfernungen zwischen diesen sind gegeben. Es existiert eine Person, die sich von einem Start- zu einem Zielort bewegt. Die zurückgelegte Strecke dieser Person soll minimal sein und jede Stadt wird nur einmal besucht.

Die folgenden Eigenschaften werden in der allgemeinen Definition in [2, S. 96] jedoch nicht zwangsläufig vorausgesetzt. Das DHP verlangt, dass (a) die gegebene Entfernungsmatrix symmetrisch ist. (b) Außerdem muss die Dreiecksungleichung erfüllt sein. (c) Des Weiteren dürfen Entfernungen nicht unendlich sein. D. h., es gibt in jedem Fall eine Verbindung

zwischen zwei Orten. Die gegebene Definition ist demnach in Bezug auf das DHP sehr allgemein gehalten und schließt Fälle ein, die das DHP nicht erfüllen.

(d) Dazu kommt, dass beim DHP mehrere Handlungsreisende zu betrachten sind. (e) Weiterhin besteht eine Tour eines Reisenden nur aus drei Orten, wobei (f) diese Tour keine Rundreise darstellt, da (g) jedes Team an verschiedenen Orten startet und endet.

Es sind also Variationen des TSP gesucht, welche die Eigenschaften (a) bis (g) möglichst genau erfüllen bzw. durch diese charakterisiert werden.

Symmetrisches und Metrisches TSP

Ein TSP wird symmetrisch genannt, wenn der Graph G symmetrisch ist und Gleichung (2.12) gilt. vgl. [3, S. 248]

$$\forall (i, j) \in A: c_{i,j} = c_{j,i} \quad (2.12)$$

Ist für das symmetrische TSP die Dreiecksungleichung für alle Knoten $i, j, k \in V$ verfüllt, so spricht man von einem metrischen TSP. Bei dieser Variante des TSP werden Knoten des Graphen mit Punkten eines Raumes gleichgesetzt. Die Kantengewichte werden durch eine Metrische Distanz zwischen den entsprechenden Knoten beschrieben. vgl. [4, S. 6] Es bietet sich die Untersuchung des euklidischen TSP an, welchem die euklidische Metrik zugrunde liegt.

„Das Euklidische TSP ist durch eine Menge von Punkten in der Ebene definiert. Der korrespondierende Graph enthält einen Knoten für jeden Punkt und Kantengewichte werden durch die Euklidische Distanz zwischen den assoziierten Knoten beschrieben.“ [4, S. 6] Im euklidischen zweidimensionalen TSP wie in [5, S. 22] beschrieben, lässt sich die Entfernung zwischen zwei Punkten i und j mit den Koordinaten (x_i, y_i) und (x_j, y_j) in der xy-Ebene durch Gleichung (2.13) beschreiben.

$$d_{i,j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \quad (2.13)$$

Das euklidische TSP besitzt die Eigenschaften (a) und (b). (c) kann anhand dieser Definition nicht direkt angenommen werden, ist aber als Spezialfall durchaus denkbar. Des Weiteren besteht die Möglichkeit, die euklidische Metrik auf das DHP zu übertragen. Orte des DHP sind in Breiten- und Längengrad gegeben, die sich in die Koordinaten eines Punktes in der Ebene umrechnen lassen. Da Orte des DHP stets innerhalb einer Stadt liegen, sind Entferungen so klein, dass die Abstandsberechnung durch den euklidischen Abstand berechnet und die Erdkrümmung vernachlässigt werden kann. Es kann demnach ein euklidisches DHP angenommen werden. Die Eigenschaften (d) bis (g) werden beim euklidischen TSP nicht betrachtet, sodass die folgenden Varianten untersucht werden.

„Multiple TSP (mTSP)“

„Das **mTSP** stellt eine Verallgemeinerung des TSP dar. Die verallgemeinerte Variante kann wie folgt definiert werden. Gegeben ist eine Menge von Knoten (Städten) und n Handlungsreisende, die sich an einem Depot befinden. Das mTSP besteht darin, Touren für alle Handlungsreisende zu finden, die an diesem Depot beginnen und an diesem enden, sodass jeder Knoten genau einmal besucht wird und die Gesamtkosten zur Besichtigung aller Knoten minimiert sind.“ vgl. [6, S. 209]

Interessant sind hierbei die Variationen „multiple Depots“ und „single Depots“. Der „single Depots“-Fall entspricht dem gerade beschriebenen verallgemeinerten Fall. Im „multiple Depots“-Fall existieren mehrere Depots, wobei an jedem Depot eine gewisse Anzahl von Handlungsreisenden starten. Daraufhin sind zwei Varianten denkbar. Entweder kehren die Handlungsreisenden am Ende ihrer Tour zu ihrem Depot zurück, oder sie beenden die Tour an einem beliebigen anderen Depot. Dabei muss die anfängliche Anzahl der Handlungsreisenden an den Depots nach Beenden der Touren gleich bleiben. vgl. [6, S. 210]

Das mTSP geht auf Eigenschaft (d) ein. Es werden mehrere Handlungsreisende betrachtet, was im DHP ebenfalls der Fall ist. Des Weiteren können Routen an verschiedenen Depots enden, was die Eigenschaften (f) und (g) teilweise beinhaltet, und mehrere Handlungsreisende starten ihre Tour vom gleichen Ort, was ebenfalls eine Gemeinsamkeit zum DHP darstellt, da immer drei Teams ihre Route am Vorspeiseort beginnen. Allerdings wird schnell klar: Weil alle Handlungsreisenden ihre Tour am selben Ort beginnen, ist Eigenschaft (f) nicht vollständig enthalten.

“Euclidian Traveling Salesman Selection Problem (TSSP)”

„Hier soll bereits der euklidische Fall des TSSP betrachtet werden, sodass, angelehnt an das Euklidische TSP, n Städte in der Ebene und deren euklidische Distanzen gegeben sind. Es soll die kürzeste TSP-Tour gefunden werden unter der Bedingung, jede der n Städte genau einmal zu besuchen. Wichtiger ist, dass hierbei eine Anzahl $k < n$ von Städten festgelegt wird. Nun soll die kürzeste TSP-Tour der Teilmengen von k Städten gefunden werden.“ vgl. [7, S. 1]

Bei dieser Variante des TSP lässt sich die Anzahl der besuchten Städte festlegen, was Eigenschaft (e) angehen soll. Im Falle des DHP ist $k = 3$, die kürzeste Tour aus 3 Knoten ist gesucht. Der Unterschied dieser Variante zum DHP ist allerdings, dass hier die kürzeste Tour unter allen diesen Teilmengen gesucht ist. Das DHP sucht jedoch die kürzesten Routen, bestehend aus drei Knoten (3 Gänge), sodass diese Routen für jedes Team minimal sind, bevorzugt für die Hauptspeiseteams.

Zusammenfassend lässt sich sagen, dass keines der hier aufgeführten Probleme das DHP vollständig beinhaltet und es aus diesem Grund keine Lösung für eines der Probleme gibt, die das DHP vollständig löst.

Trotzdem lassen sich einige Gemeinsamkeiten feststellen, wie die Suche nach einem Weg mit minimaler Summe der Distanzen für den Handlungsreisenden, die Einbeziehung mehrerer Handlungsreisender und die Beschreibung der Beziehung zwischen Orten mit der euklidischen Metrik.

Probleme, wie die Ungleichheit der Anfangs- und Zielorte der Teams, sowie das Finden der minimalen Wegstrecke für jedes einzelne Team, konnten auch durch die Variationen des TSP nicht gelöst werden.

Im Folgenden soll auf Grundlage der Gemeinsamkeiten der beiden Probleme und der Lösungsmöglichkeiten des TSP ein Lösungsansatz für das DHP erarbeitet werden.

2.3.2 Heuristiken

In Abschnitt 2.3.1 wurden einige Gemeinsamkeiten zwischen TSP und DHP herausgearbeitet. Des Weiteren sind beides kombinatorische Optimierungsprobleme. Aus diesem Grund ist zu erwarten, dass Heuristiken zur Konstruktion von TSP-Touren ebenfalls auf andere Optimierungsprobleme, wie das DHP, Erfolg versprechend angewendet werden können. Deshalb werden an dieser Stelle die grundlegenden Heuristiken vorgestellt.

Die in [4] geschilderte „Standard Version“ ist die Nearest Neighbor Heuristik. Gegeben ist ein kompletter ungerichteter Graph K_n mit den Kantengewichten c_{uv} für jedes Paar u und v von Knoten und eine Knotenmenge V mit $V = \{1, 2, \dots, n\}$. vgl. [4, S. 74]

„Formuliert als Algorithmus entsteht die folgende Prozedur.

procedure nearest_neighbor

1. Select an arbitrary node j , set $l = j$ and $T = \{1, 2, \dots, n\} \setminus \{j\}$.
2. As long as $T \neq \emptyset$ do the following.
 - 2.1. Let $j \in T$ such that $c_{lj} = \min\{c_{li} \mid i \in T\}$.
 - 2.2. Connect l to j and set $T = T \setminus \{j\}$ and $l = j$.
3. Connect l to the first node (selected in Step (1)) to form a tour.

end of nearest_neighbor

Die Laufzeit liegt in $\Omega(n^2)$.“ [4, S. 73f]

Demnach wird immer der Knoten mit der kürzesten Entfernung zum Knoten, der das aktuelle Ende der Tour darstellt, gewählt und zur Tour hinzugefügt. Ein Nachteil besteht darin, dass zu Beginn auf der einen Seite kurze Kanten entstehen, jedoch immer wieder einzelne Städte während der Tour „vergessen“ werden, die später durch längere Kanten hinzugefügt werden müssen. vgl. [4, S. 75] In [4, S. 75] wird ebenfalls erwähnt, dass diese Heuristik oft für das Konstruieren des Anfangs von TSP-Touren Verwendung findet und als Grundlage für spätere verfeinernde Methoden dient.

Da das DHP im Grunde aus vielen einzelnen kurzen Routen von Teams besteht, kann die vorgestellte Heuristik als passende Grundlage angenommen werden. Sie wird im geschilderten Routenplanungsalgorithmus aus Abschnitt 4.1 vorwiegend genutzt.

Eine weitere Möglichkeit TSP-Touren zu konstruieren ist folgende. Es wird mit kleineren Touren mit k Knoten begonnen, die beispielsweise mit der vorgestellten Heuristik erzeugt werden können ($k = 1$ ist ebenfalls möglich). Neue Knoten werden daraufhin nach einem bestimmten Kriterium an beliebigen Stellen der Tour eingefügt. vgl. [4, S. 82] Im Folgenden einige ausgewählte Kriterien (siehe vgl. [4, S. 82]):

- Nearest Insertion: Füge den Knoten ein, welcher die kürzeste Entfernung zu einem Knoten der Tour besitzt.
- Farthest Insertion: Füge den Knoten ein, dessen minimale Distanz zu einem Knoten der Tour maximal ist.
- Random Insertion: Wähle den einzufügenden Knoten zufällig.
- Smallest Sum Insertion: Füge den Knoten ein, dessen Summe von Distanzen zu Knoten der Tour minimal ist. Dies ist gleichbedeutend mit: Wähle den Knoten mit der minimalen durchschnittlichen Distanz zu Knoten der Tour.

Für das DHP werden in dieser Arbeit die Prinzipien der Nearest, Random und Smallest Sum Insertion verwendet. Die Heuristiken müssen dabei an das DHP angepasst werden. Demnach können die gegebenen Definitionen nicht vollständig übertragen werden. Die obigen Kriterien beziehen sich beispielsweise stets auf das Einfügen von Knoten. In dieser Arbeit wird die Smallest Sum Insertion jedoch für die Auswahl von Hauptspeiseorten Verwendung finden.

Die Nearest Insertion scheint für das DHP angemessen. Die Begründung ist analog zur Nearest Neighbor Heuristik. Random Insertion berechnete zumindest für TSP-Instanzen sehr gute Ergebnisse. Dies wird in [4, S. 85] Diagramm 6.11 deutlich. Die Heuristik lieferte für die dortigen Probleminstanzen im Durchschnitt die zweitbeste Lösung.

Die genaue Umsetzung des Routenplanungsalgorithmus und die damit verbundene Anwendung der Heuristiken werden in Abschnitt 4.1 beschrieben.

2.4 Recherche und Entscheidungsfindung

Der Abschnitt „Recherche und Entscheidungsfindung“ befasst sich im Folgenden mit den in der Planungsphase des Projektes erschlossenen möglichen Technologien, Diensten und damit verbundenen Schnittstellen, um die Software umzusetzen. Es wird jeweils die Art des Dienstes bzw. der Technologie beschrieben, die benötigt wird. Daraufhin folgen Möglichkeiten der Realisierung sowie deren Bewertung hinsichtlich der gestellten Anforderungen. Schließlich wird die für das Softwaresystem geeignete Methode ausgewählt und verwendete Schnittstellen näher beschrieben.

Als allgemeine Anforderungen an die verwendeten Dienste sind nach dem Pflichtenheft vor allem die Beständigkeit /N40/ und der freie Zugang der Dienste /N60/ zu erwähnen.

2.4.1 Verwendete Technologien und Entwicklungswerkzeuge

Programmiersprache

Die Umsetzung des Planungssystems erfolgt mittels Java. Diese Entscheidung wurde aus verschiedenen Gründen getroffen. Zunächst einmal besitzt das Planungssystem keine maschinen- oder plattformabhängigen Anforderungen, die den zwingenden Gebrauch einer Sprache wie C++, die solche Eingriffe zulässt, rechtfertigen würde. Des Weiteren ist eine plattformunabhängige Implementierung, wie sie durch Java ermöglicht wird, für die zu erstellende Software von großem Vorteil. Eine Ausführung auf allen Systemen, für die eine Java Virtual Machine existiert, wird dadurch ermöglicht. Damit erweitert sich der Kreis potenzieller Endanwender.

Während der Recherche stellte sich heraus, dass viele Dienstleister von geografischen Daten, deren Integration ein Kernbestandteil des Planungssystems ist, ihre Dienste durch eine Java-„Application Programming Interface (API)“ bzw. die Integration ihrer Java-Bibliotheken bereitstellen. Hier sei das oben beschriebene Tool Osm2po, die CloudMade-API oder der JXMapViewer zu nennen. Während der Recherche fiel außerdem GeoTools das Open Source Java „Geographic Information System (GIS)“ Toolkit auf, welches für die Erstellung geografischer Informationssysteme große Bedeutung hat. Damit wird Java als geeignete Plattform erkannt, die ausreichendes Potenzial zur Realisierung des Planungssystems und Einbettung geografischer Daten in dieses bietet.

Die Objektorientierung ist ein weiterer Vorteil von Java hinsichtlich der Implementierung des Systems. Die Beschreibung von Objekten wie Teams, Orte, Routen oder Geocodierer bietet eine sinnvolle Modellierung des Gesamtsystems.

Die Berechnung und Optimierung von Routen spielt eine wichtige Rolle. Dabei wird eine gewisse Performanz von Algorithmen und Laufzeitumgebung gefordert. Hier haben sich seit Java 1.0 gravierende Fortschritte in der Art der Verarbeitung von Bytecode ergeben.

„Die Technik der Just-in-Time-Compiler (...) beschleunigt die Ausführung der Programme, indem er zur Laufzeit den Bytecode, also die Programmanweisungen der virtuellen Maschine, in Maschinencode der jeweiligen Plattform übersetzt.“ [8, S. 56] Weitere Optimierungen führten schließlich „zu einer Familie von virtuellen Maschinen, die heute unter dem Namen HotSpot bekannt ist.“ [8, S. 56] HotSpot⁶ überwacht die Ausführung zur Laufzeit und findet kritische Stellen. Die Java Virtuell Machine steuert damit gezielt Übersetzung und Optimierung. vgl. [8, S. 56]

Auf dieser Grundlage ist die Java-Plattform als durchaus performant einzustufen und für das Planungssystem als absolut ausreichend zu bewerten.

Entwicklungsumgebung

Die Entwicklungsumgebung umfasst das Java Standard Edition Development Kit Version 1.7 Update Release 5 (32-bit) und die Eclipse „Integrated Development Environment (IDE)“ in der 32-bit-Version 3.7.2 (Indigo) für Java Entwickler.

Die Eclipse IDE bietet den Vorteil selbst in Java geschrieben zu sein und ist somit auf verschiedenen Systemen lauffähig. Eclipse steht unter der Common Public License und ist als quelloffene Software frei zugänglich. Mit der Eclipse IDE ist eine produktive und effiziente Entwicklung möglich. Argumente für die Verwendung sind Code-Auto vervollständigung, vordefinierte Methodennamen, Syntax Hervorhebung für Schlüsselwörter, Code-Assistent mit Methodenvorschlägen, Nachverfolgung referenzierter Code-Deklarationen, Echtzeit Kompilation mit Funktionen zum Auffinden von Fehlern, wie falsche Input Parameter oder Rückgabewerte sowie potenziellen Fehlern wie undefinierte Variablen.

Zusätzlich wird für den Entwurf des „Graphical User Interface (GUI)“ das Eclipse-Plugin WindowBuilder verwendet. Dieses stellt eine Zusammenfassung eines „Standard Widget Toolkit (SWT)- und Swing-Designers dar. Durch den visuellen Designer, Layout-Tools sowie Drag-and-drop-Funktion wird das Zusammenfassen einzelner Komponenten zu komplexen Fenstern deutlich vereinfacht.

⁶ Weiterführende Informationen auf <http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html>

2.4.2 Diskussion von Diensten und deren Bewertung

Dienste zur Geocodierung

Nach Anforderung /F30/ ist eine Geocodierung von eingelesenen Adressen notwendig.

Als erste untersuchte Möglichkeit diese Funktionalität in das System einzubetten, wird der Geocodierer von CloudMade vorgestellt. CloudMade stellt sich selbst als Dienstleister von Daten für geografisch basierte Anwendungen und Services vor. Es werden Tools und APIs für Entwickler verschiedenster Web- und Mobilplattformen zur Verfügung gestellt. Die CloudMade-Dienste basieren auf OpenStreetMap-Daten und ergänzen diese mit alternativen Quellen. [9] Der Geocodierungsdienst wird als „Hypertext Transfer Protocol (HTTP)“-API realisiert. Dabei ist es möglich, die zu codierende Adresse ohne Formatierung als Suchstring (Freiform), als auch eine genauere Anfrage durch die Definition festgelegter Parameter (Strukturierte Suche) anzugeben. Negativ fällt auf, dass sich die Freiformsuche aktuell im Alpha-Stadium befindet. CloudMade definiert dieses Stadium als fehleranfällig und ohne Gewährleistung der Funktionstüchtigkeit des Dienstes. Die strukturierte Suche im Beta-Stadium befindlich, wird grundsätzlich als stabiler Service beschrieben, der allerdings Fehler und unerwartete Ergebnisse liefern kann. vgl. [10] Ebenso wie die OpenStreetMap-Daten ist der Dienst frei zugänglich. Es werden keine Einschränkungen bezüglich einer Begrenzung der Anfragen gemacht. Allerdings werden eine Registrierung und die Nutzung des dabei zur Verfügung gestellten API-Keys für die Nutzung vorausgesetzt.

Der Geocodierer von Microsofts Bing Spatial Data Service steht als zweite Möglichkeit zur Debatte. Der Bing Spatial Data Service stellt Daten über ein „Representational State Transfer (REST)“ Interface bereit. Dieses führt verschiedene Aufgaben über das Setzen von Parametern in einem HTTP-Request aus. Zum Ausführen einer Geocodierung müssen ein „Geocode job“ und ein Upload der zu codierenden Daten erfolgen. Daraufhin kann der Status der Anfrage erfragt und das Ergebnis heruntergeladen werden. Jeder dieser Schritte wird über einen eigenen HTTP-Request ausgeführt. Daten können separat von einer „Extensible Markup Language (XML)“-Datei eingelesen werden. Die Rückgabe erfolgt ebenfalls im XML-Format. Um den Dienst nutzen zu können, wird ein Bing Maps-Account benötigt. Über diesen wird ein Bing Maps-Key für die zu nutzende Anwendung erstellt, der nur mit dieser Anwendung kompatibel ist. Die Größe der zu bearbeitenden Datei darf 300 MB nicht überschreiten. Durch die Microsoft® Bing™ Maps Platform APIs’ Terms Of Use [11] (Abschnitt 3.2 d) wird der Nutzer darauf hingewiesen, dass Daten des Dienstes ausschließlich in Verbindung mit einer Bing Maps Karte genutzt werden dürfen.

Die dritte Möglichkeit ist die Geocodierung über Google. Hier wird die Google Geocoding API Version 2 bereitgestellt. Pro Anfrage wird genau eine Adresse als Parameter in der „Uniform Resource Locator (URL)“ übergeben. Die Adressangabe kann in Form eines Suchstrings ähnlich der CloudMade-Freiformsuche erfolgen. Ausgabeformate sind „JavaScript Object Notation (JSON)“ und XML. Der Dienst ist frei zugänglich. Ein API-Key wird nicht benötigt. Allerdings unterliegt der Google-Dienst einigen Beschränkungen. Die Anzahl der Anfragen pro Tag ist auf 2500 begrenzt. Weiterhin wird in den Google Maps/Google Earth APIs Terms of Service [12] Abschnitt 10.1.1 g ausdrücklich darauf hingewiesen, dass Daten, die dem Dienst entnommen werden, ausschließlich auf Google-Karten dargestellt werden dürfen. Die Anforderung eines frei nutzbaren Dienstes wird von allen drei Kandidaten erfüllt. Hierbei ist allerdings zu beachten, dass die obligatorische Nutzung eines API-Keys bereits eine starke Bindung der zu erstellenden Software an den Dienst bedeutet. Eine Nutzung der Software würde verbindlich mit einer Registrierung bei dem Bereitsteller des Dienstes einhergehen. Die Registrierung bzw. Erstellung eines Accounts erfordert meist die Angabe persönlicher Daten, wie Name, Adresse, Telefonnummer, E-Mail, nutzende Institution und genaue Angaben über die zugrunde liegende Software. Der Nutzer muss demnach, wenn er die Software verwenden möchte, allen Richtlinien des Dienstleisters zustimmen. Um derartige Abhängigkeiten auszuschließen, wird die Realisierung über einen derartigen Dienst nach Möglichkeit vermieden.

Der Bing Spatial Data Service verlangt nicht nur die Erstellung eines API-Keys, sondern ebenfalls die Erstellung eines Bing Maps for Enterprise Accounts. Es sind die oben bereits beschriebenen Angaben und die Bestätigung durch Bing notwendig. Aus diesem Grund wird der Bing Service für die Realisierung des Geocodierers keine Verwendung finden. Die Erstellung von Jobs und der Upload von zu geocodierenden Daten wirken überdimensioniert angesichts der wenigen zu codierenden Adressen des Planungssystems und vorwiegend für die Bewältigung größerer Datenmengen konzipiert. Außerdem ist die Beschränkung über die Terms of Use für das Planungssystem problematisch, wie im Abschnitt „Visualisierung von Routen“ in dieser Arbeit beschrieben wird.

Der CloudMade Dienst bietet den Vorteil, dass die Nutzung mit keinen größeren Einschränkungen verbunden ist. Weder die Anzahl der Anfragen noch die spätere Nutzung der Daten werden vorgeschrieben. Nachteile ergeben sich allerdings in der Zuverlässigkeit des Dienstes. Zumindest die Freiform-Suche kann aufgrund des frühen Entwicklungsstandes als nicht ausreichend für eine Nutzung betrachtet werden. Weiterhin ist die Voraussetzung eines API-Keys unerwünscht. Dennoch kann die strukturierte Suche, wenn auch im Beta-Stadium befindlich, in Betracht gezogen werden.

Der Geocodierer von Google liefert vollen Funktionsumfang und hohe Genauigkeit auch bei Freiformsuchen. Die Erstellung eines Accounts ist nicht notwendig. Diese Unabhängigkeit bringt allerdings größere Beschränkungen, was die maximal zu verarbeitende Datenmenge angeht, mit sich. Es sind 2500 Geocodierungsanfragen pro Tag möglich. Dies ist jedoch vollkommen ausreichend. Nach Anforderung /L90/ sollen alle Funktionen für maximal 39 Teams ausgelegt werden. Damit sind maximal 39 Geocodierungen notwendig, sodass selbst ein mehrmaliger Programmaufruf mit unterschiedlichen Adressen und ohne Cachen der Codierung in einer separaten Datei, um den mehrmaligen Programmaufruf zu beschleunigen und den Server zu entlasten, kein Problem darstellt. Die Codierung ist mit HTTP-Requests, der Freiformsuche und dem XML-Rückgabeformat sehr einfach gehalten und für die Realisierung des Planungssystems vollkommen ausreichend. Einziger Nachteil ist damit die Einschränkung durch die Terms of Service.

Aus Gründen der Genauigkeit der Geocodierung des Google-Dienstes wird das Planungssystem auf diesen zurückgreifen. Durch die Terms of Service stößt man hier auf Grenzen der Realisierbarkeit⁷. Für eine rein akademische Nutzung und prototypische Umsetzung des Planungssystems ist der Google-Service sicher geeignet. Für andere Anwendungsfälle wird ebenfalls der CloudMade-Dienst zur Verfügung gestellt. Dessen Nutzung ist allerdings mit Nachteilen, was Genauigkeit und Zuverlässigkeit angeht, behaftet.

Dienste zur Entfernungsbestimmung

Die Entfernungsbestimmung ist nach Anforderung /F100/ nötig. Das Produkt soll nach Produktleistung /L30/ die Entfernung per Luftlinie bestimmen können. Dies wird als ausreichend betrachtet.

Wie in 2.3.1 Abschnitt „Symmetrisches und Metrisches TSP“ beschrieben, kann die euklidische Metrik auf das DHP übertragen werden. Die Distanzbestimmung erfolgt anhand der dort aufgeführten Formel nach dem Satz des Pythagoras. Die Berechnung erfolgt demnach nicht über einen Dienst und wird aus diesem Grund hier nicht weiter betrachtet. Es ist zu bemerken, dass diese Art der Distanzbestimmung nicht in jedem Fall Ergebnisse liefert, bei denen von der Entfernung auf die letztendliche Fahrzeit geschlossen werden kann. Nach Produktleistung /L40/ ist die Möglichkeit einer derartigen Einschätzung jedoch vorteilhaft. Da zwischen zwei Orten unter Umständen Hindernisse existieren können, die nur mit größerem Umweg zu passieren sind, liefert eine Distanzbestimmung auf Grundlage des konkreten Verkehrsnetzes der betrachteten Stadt genauere Ergebnisse. Im konkreten Anwendungsfall ist beispielsweise die Weiße Elster in Leipzig nur über einige Verkehrstrassen überquerbar.

⁷ Dies wird im Unterabschnitt „Visualisierung von Routen“ genauer beschrieben.

Der Routing-Dienst von CloudMade ist eine Möglichkeit, solche Ergebnisse zu erhalten. Der CloudMade Routing Web Service⁸ bietet Echtzeit-Routing, sodass Ergebnisse zeitnah zur Verfügung stehen. Er arbeitet auf Grundlage von OpenStreetMap-Daten. Des Weiteren ist die CloudMade Java API⁹ frei zugänglich. Damit lässt sich der Dienst problemlos in Java-Projekte einbinden. Das API basiert auf HTTP-Requests des Routing Web Service. Es existiert die Möglichkeit, sowohl Routen per Auto als auch per Fahrrad zu berechnen. Ein API-Key wird für die Nutzung vorausgesetzt.

Ein zweiter alternativer Dienst wird mit Osm2po geboten. Osm2po bietet umfangreiche Möglichkeiten, Daten des „OpenStreetMap (OSM)“-Projektes aufzubereiten und auf diesen Routenberechnungen durchzuführen. Das Java basierte Tool fungiert sowohl als Konverter von OSM-Daten in Graph-Daten als auch in „Structured Query Language (SQL)“-Files für Geodatenbanken wie PostgreSQL. Auf den Graph-Daten ist ein Routing durch die interne Routing-Engine möglich. Das Tool beinhaltet einen einfachen HTTP basierten Routing-Dienst, der lokal installiert wird, kann jedoch auch direkt als Bibliothek eingebunden werden und ermöglicht ein Routing über eine Java-API. Dieser sogenannte Osm2po Core ist als Freeware verfügbar. Im aktuellen Release 4.5.2 wird neben Routen per Auto auch die Bestimmung von Fahrradrouten unterstützt. Alle für das Routing relevanten Daten werden lokal gehalten und bei Berechnungen in den Arbeitsspeicher kopiert. OSM-Daten liegen bei Anbietern wie GEOFABRIK oder CloudMade kostenfrei für die gesamte Erde vor und können über eine interne Schnittstelle je nach Bedarf geladen werden.

Entfernungsbestimmungen per CloudMade und Osm2po haben den Vorteil der Berücksichtigung von Verkehrswegen. Dies kann in Abhängigkeit der betrachteten Stadt für die Planung von Vorteil sein. Des Weiteren bieten die Dienste die Möglichkeit, Fahrradrouten zu berechnen. Dies wird in Produktleistung /L30/ als alternative Berechnungsmöglichkeit angegeben. Im Anwendungsfall liegt der Grund darin, dass das bevorzugte Verkehrsmittel der Studenten das Fahrrad ist. Routen per Auto sind hier weniger interessant. Als zweite Alternative wäre eine Berechnung mit den öffentlichen Verkehrsmitteln in Leipzig ebenfalls interessant. Hier konnte in der Recherche kein Dienst gefunden werden, der entsprechende Informationen bereitstellt.

Ein weiteres Unterscheidungskriterium der Dienste ist die Antwortzeit. Bei Osm2po werden Daten und Berechnungen lokal gehalten bzw. ausgeführt. Der CloudMade Routing Service wird auf Server-Seite angeboten und reagiert in Abhängigkeit von Netz- und Serverauslastung. Es ergeben sich deutliche Antwortzeitunterschiede siehe Abbildung 1.

⁸ <http://developers.cloudmade.com/projects/show/routing-http-api>

⁹ <http://developers.cloudmade.com/documentation/java-lib/index.html>

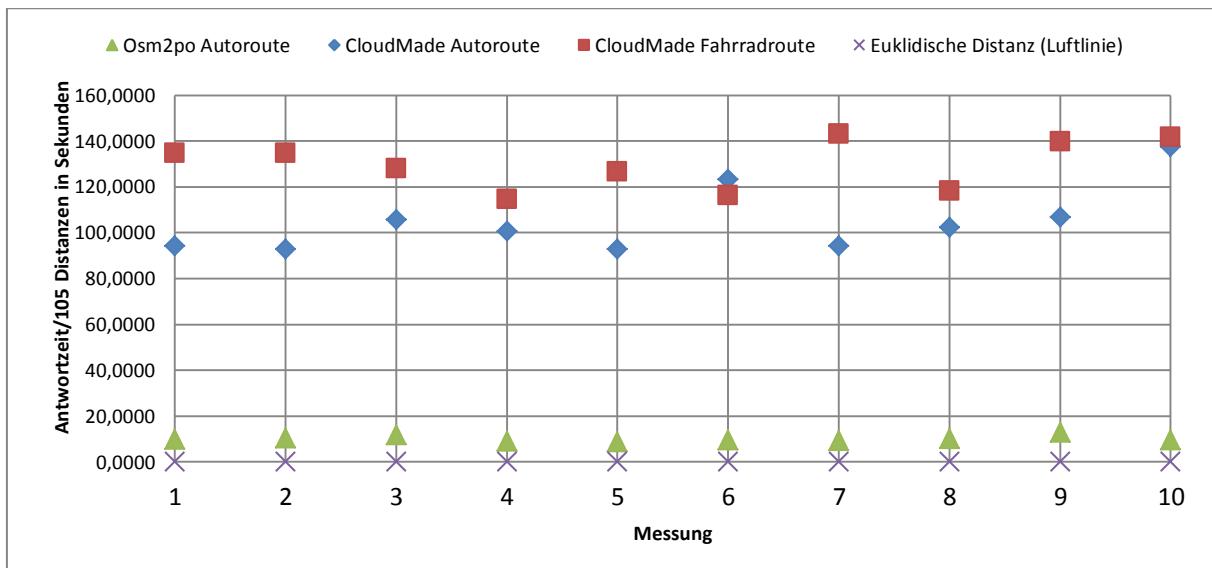


Abbildung 1 Es wird das Antwortzeitverhalten der Dienst für die Distanzbestimmung beschrieben. Der zugrunde liegende Datensatz beinhaltet 15 Teams. Dies entspricht $(15 * 14) / 2 = 105$ Distanzbestimmungen. Es wurden 10 unabhängige Messungen durchgeführt. Zeitpunkt der Messung ist der 04.07.2012.

In Abbildung 1 wird das Antwortzeitverhalten der Dienste über 10 unabhängige Messungen veranschaulicht. Es wurde ein Beispieldatensatz mit 15 Teams, dies entspricht 105 Distanzbestimmungen, zugrunde gelegt. Die Berechnung der Luftlinie liefert die schnellsten Ergebnisse. Wie bereits erläutert, sind diese Distanzen je nach Anwendungsszenario weniger aussagekräftig. Osm2po liefert als Routing Service durch die Ausnutzung lokaler Ressourcen und dementsprechend kurzer Verarbeitungszeiten sehr schnelle Ergebnisse mit einer durchschnittlichen Antwortzeit von 10 s. Hier wurden nur Autorouten bestimmt, da die Unterstützung von Fahrradrouten zum Zeitpunkt der Untersuchung erst in der Beta-Version vorlag. Als Referenzsystem auf Client-Seite wurde ein Intel Atom N280 mit 2 GB Arbeitsspeicher verwendet. Der CloudMade-Dienst für die Bestimmung von Autorouten ist mit durchschnittlich 105 s über zehn Mal langsamer, was durch die Anfrage und Antwortzeit, sowie Auslastung des Servers zu erklären ist. Bei der Berechnung von Fahrradrouten erhöht sich die durchschnittliche Antwortzeit um weitere 14 s. Dabei ist anzumerken, dass die Berechnung von Fahrradrouten die für das hier betrachtete Anwendungsszenario aussagekräftigsten Ergebnisse liefert.

Abbildung 2 verdeutlicht die Unterschiede der berechneten Distanzen der Dienste auf einem Beispieldatensatz mit 9 Teams. Die Routen sind aufsteigend nach ihrer durchschnittlichen Distanz geordnet.

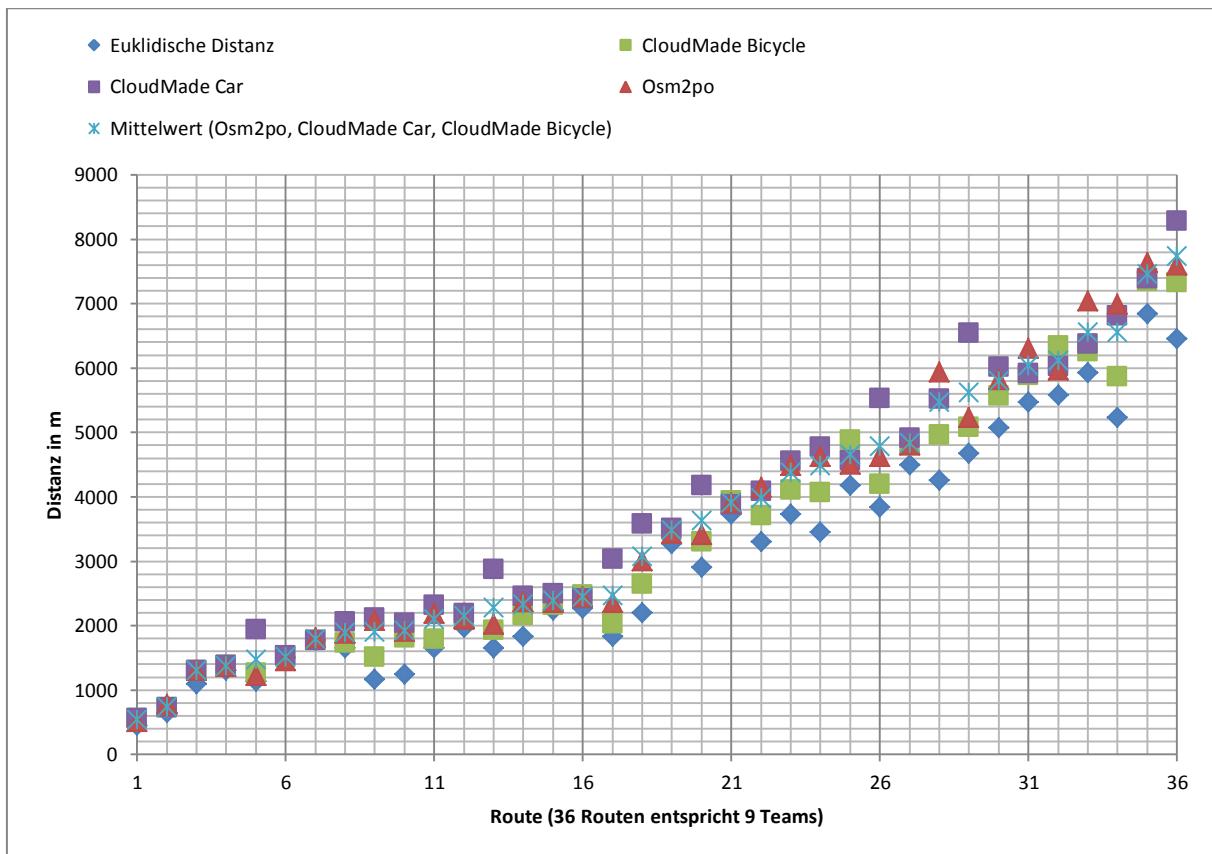


Abbildung 2 Es werden die Unterschiede der Berechneten Distanzen der Dienste für die Distanzbestimmung veranschaulicht. Die Routen sind aufsteigend nach ihrer durchschnittlichen Distanz geordnet. Zeitpunkt der Messung ist der 04.07.2012.

Es ist zu erkennen, dass die Distanzen innerhalb einer Route ab einer durchschnittlichen Länge von ca. 1,8 km teilweise stark variieren. Für kleinere Routen sind die berechneten Distanzen annährend konstant. Die maximale Abweichung der bestimmten Distanzen innerhalb einer Route ist bei Route 29 zu erkennen. Hier liegt eine Abweichung zwischen Luftlinie und CloudMade Autoroute von rund 2 km vor. Die Routen 26, 28, 34 und 36 zeigen ähnliches Verhalten. Diese Werte bestärken das Argument der geringen Aussagekraft einer Luftlinienbestimmung zum tatsächlichen Fahrweg.

Es ist zu erkennen, dass die Berechnung der Autorouten per CloudMade bis auf einzelne Ausnahmen, siehe Routen 28, 31 und 33, die größten Distanzen pro Route ergibt. Es folgen die Autorouten per Osm2po und schließlich die Fahrradrouten per CloudMade, welche nur in den Routen 25 und 32 größer als die Distanzen von Osm2po sind. Die Luftlinie liefert, wie zu erwarten, die kleinsten Ergebnisse. Der berechnete Mittelwert der Distanzen liegt sehr stark mit den errechneten Distanzen des Osm2po-Dienstes zusammen, sodass dieser für gute Näherungen an den tatsächlichen Fahrweg bzw. die Fahrzeit zu empfehlen ist.

Unverständlich ist dagegen die an vielen Routen auftretende Varianz der Distanzen zwischen Osm2po und CloudMade-Autorouten. Hier wäre eine größere Übereinstimmung zu erwarten gewesen, da beide Dienste auf Grundlage von OpenStreetMap-Daten arbeiten. Vor allem bei

größeren Distanzen ab 5,5 km dominiert vermehrt der Osm2po-Dienst. Allerdings sind, über alle Routen gesehen, einzelne Ausreißer mit deutlich größeren Distanzen des CloudMade-Dienstes zu erkennen. Da CloudMade zur Vervollständigung seines Routendienstes auch andere Datenquellen einschließt, kann hier eventuell ein Grund für diese Abweichung gesehen werden. Außerdem sind OSM-Daten auf viele unterschiedliche Weisen für ein Routing interpretierbar.

Zusammenfassend lässt sich sagen, dass jedes Verfahren für den konkreten Anwendungsfall Vorteile und Nachteile besitzt. Von der einfachen und schnellen Bestimmung der Euklidischen Distanz, über die schnelle, jedoch an lokale Ressourcen gebundene Berechnung über Osm2po bis zur zeitaufwendigen, aber genauere Ergebnisse liefernden Kommunikation mit dem CloudMade-Dienst wird eine breite Palette von Anwendungsbereichen abgedeckt. Aus diesem Grund werden sowohl Osm2po und CloudMade in Form von Diensten als auch die Bestimmung der Euklidischen Distanz im Planungssystem angeboten.

Visualisierung von Routen

Die Berechnung und Optimierung von Routen stellt das Herzstück des Planungssystems dar. Allerdings sind diese errechneten Daten ohne eine passende und übersichtliche Visualisierungsmöglichkeit für den Nutzer ohne Bedeutung. Die Visualisierung verschiedener Routen stellt durchaus eine Herausforderung dar, da zu bedenken ist, dass für jedes Team eine Route, bestehend aus drei Veranstaltungsorten, existiert. Bei bis zu 39 zu erwartenden Teams, verteilt auf einen verhältnismäßig kleinen Darstellungsbereich wie eine Karte der Stadt Leipzig, kann die Darstellung schnell unübersichtlich und für den Endanwender nutzlos werden. Nach Anforderung /F80/ muss eine Visualisierung der Ergebnisse stattfinden. Es wurden zwei in Betracht kommende Dienste identifiziert.

Zum einen besteht die Möglichkeit, die Google Static Maps API Version 2 zu verwenden. Dies ist eine reine HTTP-API. In einer URL werden alle Parameter, wie Geokoordinaten, zu verwendende Markierungs-Icons, deren Beschriftung und einzulezeichnende Pfade, angegeben. Auf Grundlage der angegebenen Geokoordinaten zeichnet der Google-Dienst alle Angaben auf den jeweiligen Google-Kartenausschnitt und gibt diesen zurück. Kartenausschnitte dürfen außerdem nur im Browser oder in der eigenen Web-Applikation angezeigt werden.

Die Vorteile bestehen darin, dass die Anfragen sehr einfach per URL gestellt werden können und Google die Aufgabe des Erstellens der Kartenausschnitte übernimmt. Von Nachteil ist allerdings, dass die Rückgabe in Form einer Bilddatei erfolgt. Das heißt, es kann nicht in den Kartenausschnitt gezoomt werden, um Details besser erkennen zu können. Des Weiteren wird die Länge der URL auf 2048 Zeichen begrenzt. Dies schränkt die Menge von Informationen

ein, die eingezeichnet werden können. Die Begrenzung der Anfragen setzt Google auf 25000 pro Tag fest.

Die zweite Möglichkeit besteht darin, Kartenmaterial direkt in die eigene Java-Anwendung zu integrieren. Dies kann mit Hilfe des Open Source JXMapViewers erfolgen. Dabei handelt es sich um eine Swing-Komponente, die auf Grundlage der SwingX-Erweiterung des Swing GUI Toolkit entwickelt wurde.

OpenStreetMap-Kartenmaterial wird dabei in Form von Tiles (Kartenausschnitten) von einem sogenannten OpenStreetMap Tile Server heruntergeladen und in einem Frame angezeigt. Der Tile Server hält eine Vektor-Karte der gesamten Erde bereit. Ein bestimmter Kartenausschnitt wird über definierte Parameter in der URL angefordert. Es ist möglich, die aktuell angezeigte Karte zu verschieben bzw. in sie hinein zu zoomen. Dabei werden neu hinzukommende Kartenausschnitte nachgeladen. Bereits geladene Ausschnitte werden zwischengespeichert. Nachteile sind, dass für das Einzeichnen von Markierungen und Pfaden nur teilweise vorgefertigte Klassen bzw. Methoden zur Verfügung gestellt werden. Das Einzeichnen muss demnach größtenteils durch eigenen Programmieraufwand erledigt werden. Allerdings sind hier keinerlei Einschränkungen der Anfragen bzw. einzueichnenden Informationen gegeben. Da dem Nutzer vielfältige Möglichkeiten der Visualisierung zur Verfügung gestellt werden sollen, ist es äußerst unpraktisch eine Einschränkung beim Einzeichnen von Pfaden oder Markierungen, wie bei der Google Static Maps API beschrieben, berücksichtigen zu müssen. Beispielsweise wäre ein Einzeichnen aller Routen und Orte nicht zu realisieren, da die mögliche Länge der URL zu kurz ist und damit zu wenige URL-Parameter übertragen werden können. Außerdem sind statische Karten sowie eine Anzeige ausschließlich im Browser für eine gute Bedienbarkeit des Programms nicht zu empfehlen. Aus diesem Grund wird hier auf den JXMapViewer unter Zuhilfenahme von OpenStreetMap-Daten zurückgegriffen.

3. Modellierung

3.1 Datenfluss

Das Planungssystem wird in fünf Komponenten unterteilt. Dies sind der Import/Export, der Geocodierer, der Distanzkalkulator, der Routenberechner und der Visualisierer. Jede dieser Komponenten trägt entscheidend zur Realisierung bei. Die Bestandteile des Planungssystems und der Datenaustausch zwischen ihnen werden in Abbildung 3 gezeigt. Die Beschreibung der verwendeten Dienste, bezeichnet mit Google-Geocodierer, CloudMade, OpenStreetMap Tile Server und Osm2po, erfolgte im Abschnitt 2.4.2.

Datenflussdiagramm 0.

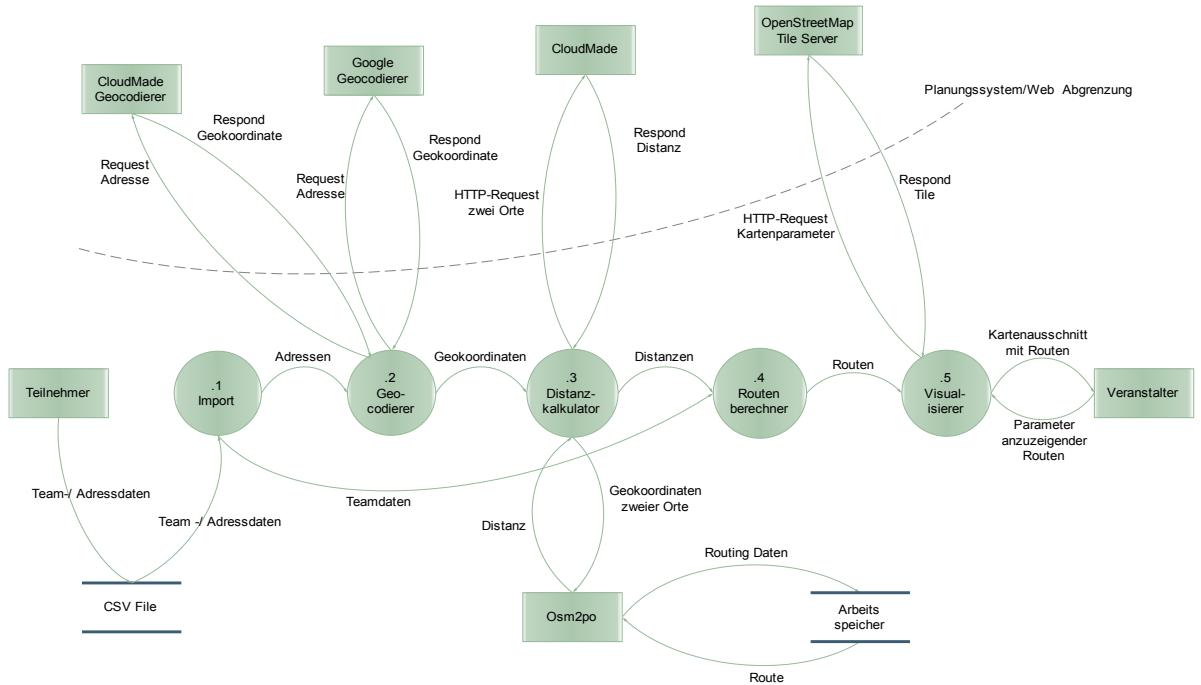


Abbildung 3 Komponenten und Datenaustausch der Komponenten des Planungssystems. Der Export, welcher mit dem Import als eine Komponente betrachtet wird, wurde aus Gründen der Übersichtlichkeit nicht eingetragen.

Der Teilnehmer trägt zusammen mit seinem Teammitglied Team- und Adressdaten in ein Onlineformular ein. Diese Daten werden intern in einer CSV-Datei gespeichert. Die Import-Komponente liest Team- und Adressdaten aus selbiger Datei ein und stellt sie für die weitere Verarbeitung zur Verfügung.

Das Planungssystem benötigt eine Möglichkeit, die für jedes Team angegebene Adresse in die entsprechende Geokordinate, d. h. Breiten- und Längengrad, umzuwandeln. Auf der Grundlage dieser Koordinaten wird später die Entfernungsbestimmung der Orte zueinander durchgeführt. Diese Aufgabe übernimmt der Geocodierer. Er erhält Adressdaten von der Importkomponente und gibt Geokoordinaten zurück.

Die Grundlage der späteren Routenplanung ist die Entfernungsbestimmung zwischen den Veranstaltungsorten. Es werden die Entferungen von jedem Veranstaltungsort zu jedem anderen Ort benötigt. Nur auf diese Weise können während der Routenplanung Entscheidungen getroffen werden, welche Routen optimal sind und welche Routen aufgrund ihrer Länge von der Lösung ausgeschlossen werden. Der Distanzkalkulator nimmt Geokoordinaten vom Geocodierer entgegen und berechnet die benötigten Entferungen.

Der Routenberechner ist die Kernkomponente des Planungssystems. Anhand der Distanzen des Distanzkalkulators und den Teamdaten des Imports werden hier mögliche Verteilungen von Teams auf die Vor-, Haupt- und Nachspeise sowie Routen zwischen den Teams berechnet.

Die Visualisierungskomponente nimmt die Routen des Routenberechners entgegen und stellt Möglichkeiten der Visualisierung dieser bereit. Der Veranstalter kann über das GUI Parameter der Darstellung von Routen anpassen.

3.2 Use Cases

Abbildung 4 zeigt die an das Pflichtenheft angelehnten Anwendungsfälle in einem Use Case Diagramm. Hier wird deutlich, welche einzelnen Aktionen von den Teilkomponenten bewältigt werden müssen. Der Aufbau orientiert sich grundsätzlich an den in Abbildung 3 gezeigten Teilkomponenten.

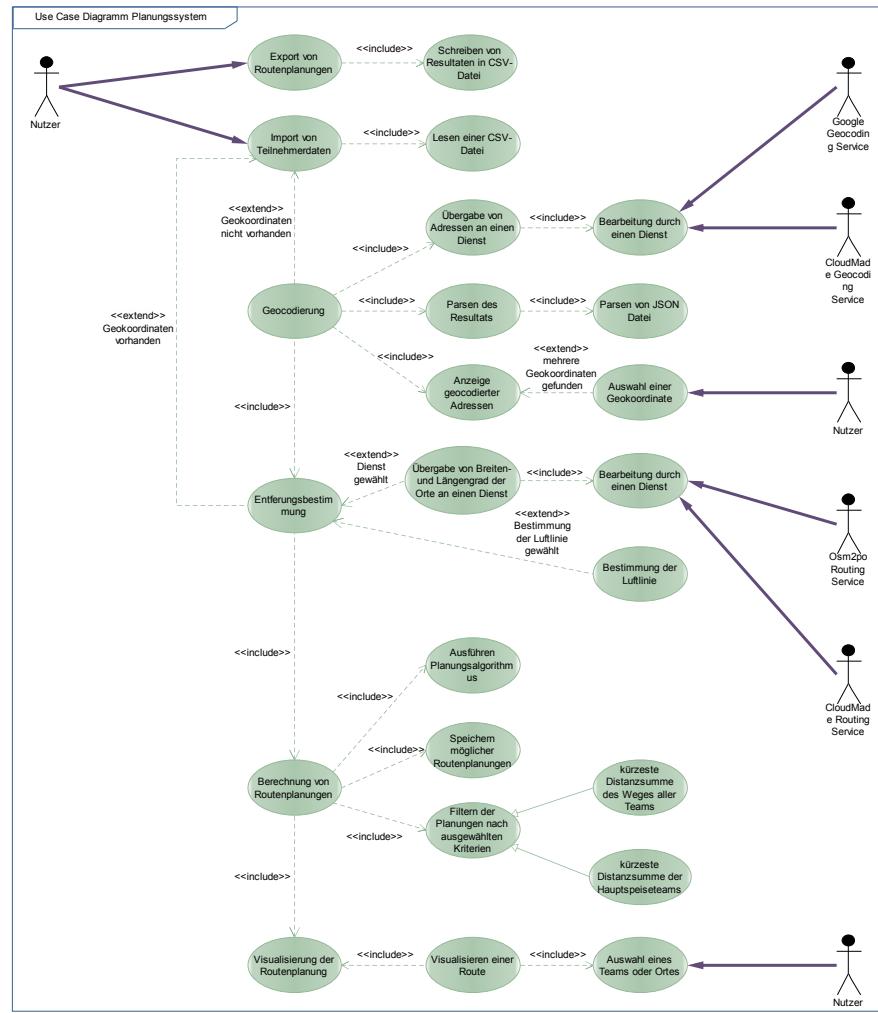


Abbildung 4 Use Case Diagramm des Planungssystems

Der Nutzer als Akteur stößt die Routenplanung an, indem er den Import von Teilnehmerdaten startet.

Neben dem in Abschnitt 3.1 beschriebenen Verhalten wird zusätzlich darauf hingewiesen, dass die Geocodierung ausschließlich ausgeführt wird, wenn die CSV-Datei keine Geokoordinaten

enthält. Außerdem werden von einem Dienst geocodierte Daten im JSON-Format zurückgeliefert und entsprechend geparsst. Des Weiteren beinhaltet der Prozess der Geocodierung die Anzeige der geocodierten Adressen. Werden mehrere Geokoordinaten für eine Adresse zurückgeliefert, so muss der Nutzer eine Entscheidung treffen.

Die Visualisierung zeigt einzelne Routen an. Die Auswahl der anzuzeigenden Routen wird ebenfalls vom Nutzer vorgenommen. Der Anwendungsfall Routenplanung zeigt, dass mehrere Routenplanungen berechnet werden, die gespeichert und nach bestimmten Kriterien gefiltert werden müssen, um dem Nutzer eine Auswahl der „besten“ Route zu erleichtern.

Des Weiteren wird die Export-Schnittstelle visualisiert. Hier müssen Daten, ähnlich dem Import, in einem vorgegebenen CSV-Format exportiert werden können.

3.3 Klassenbeschreibung

Die sich aus Abschnitt 3.1 und 3.2 ergebenden Klassen werden in der Klassenbeschreibung näher erläutert. Die Beschreibung beginnt mit den Fachkonzeptklassen. Diese realisieren die grundlegenden Funktionen des Planungssystems. Im zweiten Abschnitt werden die Benutzeroberfläche und deren Klassen genauer gezeigt. Außerdem sind Beispiele des genauen Ablaufs durch Sequenzdiagramme gegeben.

3.3.1 Fachkonzeptklassen

Die Fachkonzeptklassen und deren Beziehungen untereinander sind in Abbildung 5 zu sehen. Den einzelnen Teilkomponenten wird jeweils eine Klasse bzw. ein Interface zugeordnet. Die beschriebenen Dienste realisiert das Planungssystem ebenfalls durch eigene Klassen, die das vorgegebene Interface realisieren.

Die Zuordnung erfolgt folgendermaßen.

- Import und Export: Klasse „CSVHandler“
- Geocodierer: Interface „Geocoder“
- Geocodierer Dienste: „GoogleGeocoder“ und „CloudMadeGeocoder“
- Distanzkalkulator: Interface „DistanceCalculator“
- Distanzkalkulator Dienste: “CloudMadeDistanceCalculator”,
“Osm2poDistanceKalkulator”, “EuklidDistanceCalculator” (Luftlinie)
- Routenberechner: Klasse „TeamRouteConstructor“
- Visualisierer: Klasse „MapView“

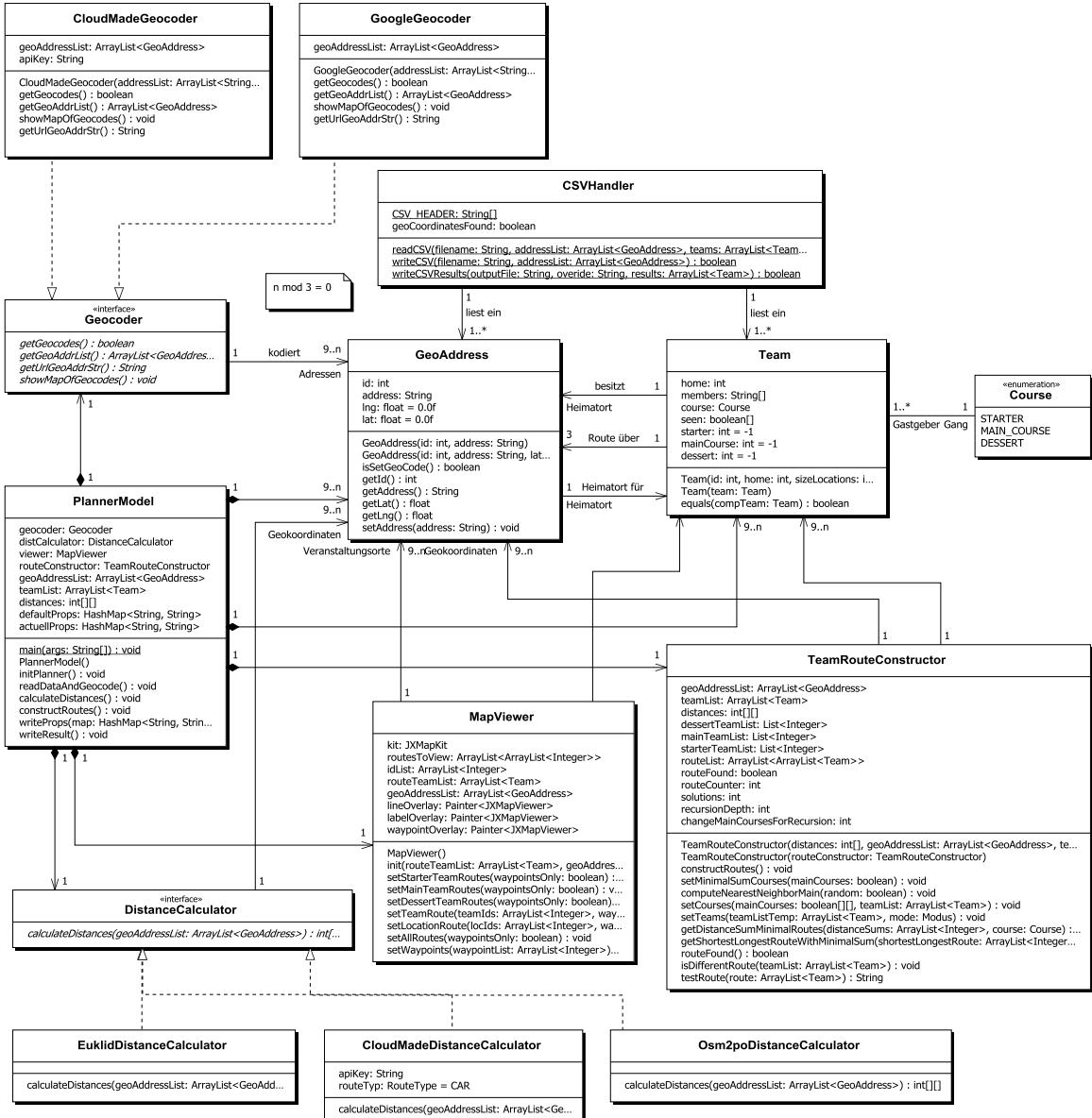


Abbildung 5 Fachkonzeptklassen

Das Gesamtsystem wird durch die Klasse „PlannerModel“ realisiert. Hier werden die einzelnen Komponenten aufgerufen und zurückgelieferte Daten gespeichert, um sie der jeweils nächsten Komponente übergeben zu können (siehe Datenfluss Abbildung 3). Das Benutzerinterface, im nächsten Abschnitt beschrieben, wird nach dem Model-View-Controller Interaktionsmuster realisiert. Die „PlannerModel“ Klasse entspricht hierbei dem Model. Die Klasse „Course“ als Aufzählung bildet die Aufteilung der Teams in Vorspeise- („STARTER“), Hauptspeise- („MAIN_COURSE“) und Nachspeiseteams („DESSERT“) ab. Die Veranstaltungsorte mit Adressname und Geokoordinate werden jeweils in einer Instanz der Klasse „GeoAddress“ gespeichert. Die Teams mit Teammitgliedern und Routen sind in Objekten der Klasse Team gekapselt.

Das Model hält Referenzen auf jede Teilkomponente und Methoden, um diese aufzurufen. Die Methode „constructRoutes()“ erzeugt beispielsweise eine Instanz der Klasse

„TeamRouteConstructor“ und berechnet die Routenplanungen. Die wichtigsten Attribute sind die „geoAddressList“ und die „teamList“. Erstere enthält alle Veranstaltungsorte. Die „teamList“ enthält alle Teams. Der Heimatort eines Teams hat den gleichen Index in der „geoAddressList“ wie das zugehörige Team in der „teamList“. Die Teilkomponenten referenzieren diese Listen und führen Berechnungen auf ihnen aus. Weiterhin existieren die Attribute „defaultProps“ und „actuellProps“ zum Speichern der Programm-Eigenschaften. Diese sollen in eine „Properties“-Datei gespeichert werden und somit für einen erneuten Programmaufruf zur Verfügung stehen.

Ein Team-Objekt besteht aus seinem „Identifier (ID)“, dem Namen der Teammitglieder, dem Gang, den dieses Team anrichtet („course“), einer Liste von Teams, die dieses Team während seiner Route sieht („seen“) sowie dem Vorspeise- („starter“), Hauptspeise- („mainCourse“) und Nachspeiseort („dessert“), welchen dieses Team auf seiner Route passiert.

Ein „GeoAddress“-Objekt wird durch seine ID, den Adressnamen sowie Breiten- und Längengrad des jeweiligen Ortes gekennzeichnet.

Der „CSVHandler“ wird mit Hilfe von statischen Methoden realisiert, da er ausschließlich zum Lesen und Schreiben von Dateien dient.

Das Interface „Geocoder“ stellt Methode zur Anfrage und zum Parsen von Geocodierungen („getGeocodes()“), zum Zurückgeben der Orte mit Geokoordinaten (getGeoAddrList()), sowie zum Anzeigen der Geokoordinaten („showMapOfGeocodes()“) bereit. Der „CloudMadeGeocoder“ benötigt außerdem ein Attribut zum Speichern des API-Keys.

Der „DistanceCalculator“ generiert aus der „geoAddressList“ ein zweidimensionales Integer-Array mit den benötigten Distanzen (Angabe in Metern). Zum problemlosen Hinzufügen neuer Dienste wird hier ein Interface zur Verfügung gestellt. Hilfsmethoden für Berechnungen sind in den implementierenden Klassen in Abbildung 5 nicht eingetragen.

Der „TeamRouteConstrucor“ besitzt Attribute und Methoden zum Speichern der Routen („routeList“) und Zurückgeben von Informationen über berechnete Ergebnisse, wie „routeFound“ und „routeCounter“. Außerdem müssen die Parameter des Algorithmus festgelegt werden. Dies geschieht über die Attribute „solutions“, „recursionDepth“ und „changeMainCoursesForRecursion“.

Die Visualisierung durch die Klasse „MapView“ benötigt das JXMapKit als Swing Komponente, um die Kartendarstellung zu realisieren. Die aktuell zu zeichnende Routenplanung wird in der „routeTeamList“, einzelne anzuzeigende Teams in der „idList“ abgelegt. Von großer Bedeutung sind die über die OpenStreetMap-Karte zu legenden Overlays. Für das Einzeichnen der Orte dient das „WaypointOverlay“. Routen werden durch das „LineOverlay“ eingezeichnet und Ortsbezeichnungen über das „LabelOverlay“. Die einzelnen

Setter-Klassen zeichnen beispielsweise Routen von Vorspeiseteams („setStarterTeamRoutes()“), einzelne Routen der Teams („setTeamRoute()“) oder einzelne Orte („setWaypoints()“) ein.

Das Sequenzdiagramm in Abbildung 6 zeigt den Ablauf des Einlesens und Geocodierens.

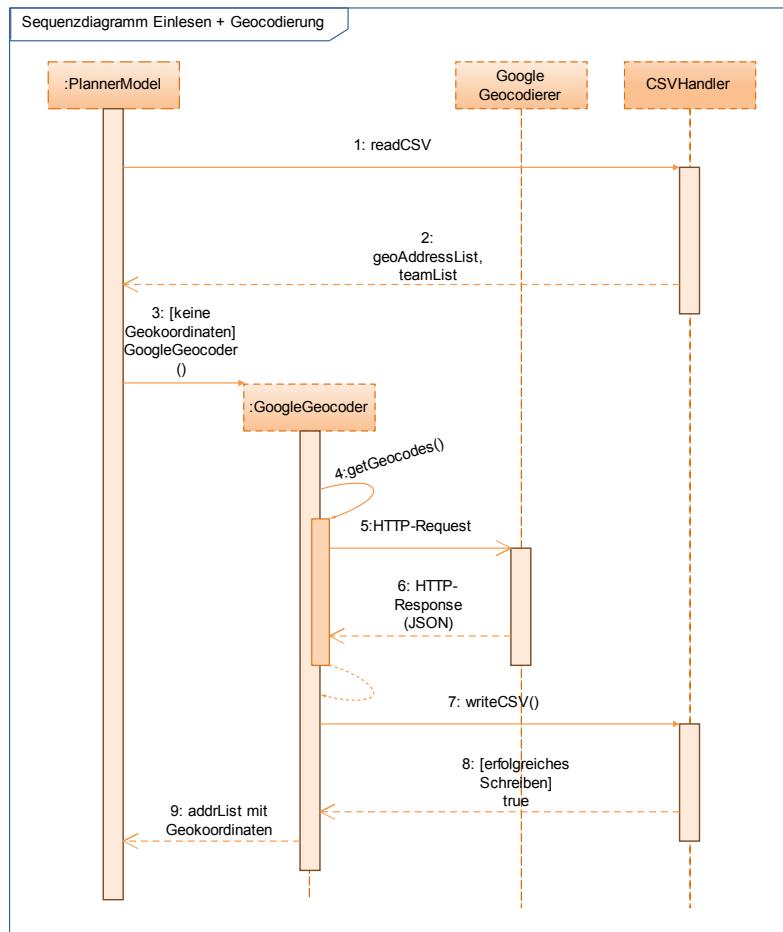


Abbildung 6 Beschreibung des Einlesens einer CSV-Datei ohne Geokoordinaten mit entsprechender Geocodierung.

Die Schritte 1 und 2 zeigen das Einlesen der CSV-Datei. Die folgenden Schritte werden ausgeführt, wenn keine Geokoordinaten in der Datei gefunden werden. In Schritt 3 wird der „GoogleGeocoder“ für die Geocodierung instanziert. Als Nächstes muss über die Methode „getGeocodes()“ der Google-Dienst über einen HTTP-Request angefragt werden (Schritte 4-6). Das Ergebnis wird im JSON-Format zur Verfügung gestellt. Nach dem Parsen des Ergebnisses wird eine CSV-Datei mit den Geokoordinaten erzeugt (Schritte 7-8). Damit ist es nicht nötig, die Adressen erneut zu geocodieren. Schließlich ist die fertige Liste der Veranstaltungsorte mit allen Geokoordinaten zurückzugeben (Schritt 9).

Abbildung 7 beschreibt die Bestimmung von Distanzen unter Verwendung des Osm2po-Routing-Service durch ein Sequenzdiagramm. In Schritt 1 wird in der „PlannerModel“-Klasse der „Osm2poDistanceCalculator“ instanziert. Es folgt der Aufruf der

Methode „calculateDistance()“ auf dem entstandenen Objekt (Schritt 2). Hier wird der Osm2po-Dienst über die eingebundene Java-Bibliothek angefragt. Es werden jeweils die Geokoordinaten zweier Orte übergeben (Schritt 3). Die Distanz zwischen diesen wird schließlich zurückgegeben und dem „PlannerModel“ übergeben (Schritte 4-5).

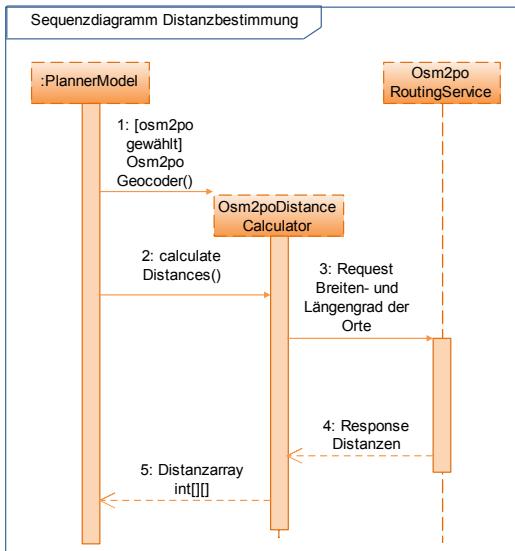


Abbildung 7 Beschreibung der Distanzbestimmung

3.3.2 Benutzeroberfläche

Die Benutzeroberfläche ist nach dem „Model-View-Controller (MVC)“-Ansatz realisiert. „Mit Model-View-Controller (MVC) wird ein Interaktionsmuster in der Präsentationsschicht von Software beschrieben.“ [13, S. 511] Der MVC-Ansatz ist ein Konzept, das die Interaktionen des Benutzers sauber von den dadurch veränderten Daten und der Darstellung trennen soll. vgl. [13, S. 513] „Das Beobachter-Muster bildet die Basis von MVC. (...) Das Muster wird angewandt, um Änderungen an einem Model an die Objekte zu kommunizieren, die das Model darstellen, die sogenannten Views.“ [13, S. 511f]

Durch den MVC-Ansatz ist es möglich, zu einem Model mehrere Views und mehrere Controller zur Verfügung zu stellen. In der Regel hat jeder View einen spezifischen Controller. „Dabei ist der *Controller* für die Verarbeitung der Eingaben (zum Beispiel Mausklicks) zuständig und für deren Kommunikation an das Modell. Das *Model* ist passiv, es wird vom Controller befragt und modifiziert. Der *View* befragt das Modell, um auf dieser Grundlage seine Darstellung anzupassen.“ [13, S. 513] Das Beobachter-Muster (Observer-Pattern) wird in Java wie folgt realisiert. Das Model wird von der Klasse Observable abgeleitet. View und wenn nötig Controller implementieren das Interface Observer.

Klassen, welche für die Realisierung der Benutzeroberfläche zuständig sind, werden im Klassendiagramm in Abbildung 8 beschrieben. Die Klasse „PlannerModel“ verbindet die Fachkonzeptklassen mit den Klassen der Benutzeroberfläche. Ihr Inhalt wurde bereits in Abbildung 5 gezeigt. Neben dem Model sind zwei Views mit zugehörigem Controller vorhanden. Das „PlannerView“ mit zugehörigem „PlannerController“ ist für die Anzeige sowie Verarbeitung von Nutzereingaben, welche die eigentliche Routenplanung betreffen, zuständig. „PropertiesView“ und „PropertiesController“ sind für die Präsentation sowie Manipulation von Programmeinstellungen verantwortlich. Das Model als zu überwachende Klasse wird von der Klasse Observable abgeleitet. Die Views implementieren das Interface Observer. Controller und Views halten Referenzen auf das Model, um jederzeit auf die aktuellen Daten des Planers zugreifen zu können. Das View referenziert den zugehörigen Controller und umgekehrt. Die Controller stellen Methoden zum Entgegennehmen verschiedener Eingabe-Events wie „ActionEvents“ oder „WindowEvents“ zur Verfügung. Der „PropertiesController“ speichert außerdem die aktuell eingestellten Programmeigenschaften. Werden sie im Menü übernommen, so sollen sie im Model als neue Eigenschaften gesetzt und in eine Properties-Datei geschrieben werden. Die Views dagegen initialisieren das GUI, indem Swing Komponenten in Containern angeordnet werden. Hier sind Check-Boxen, Auswahl-Boxen, Buttons, Listen oder Label zu nennen.

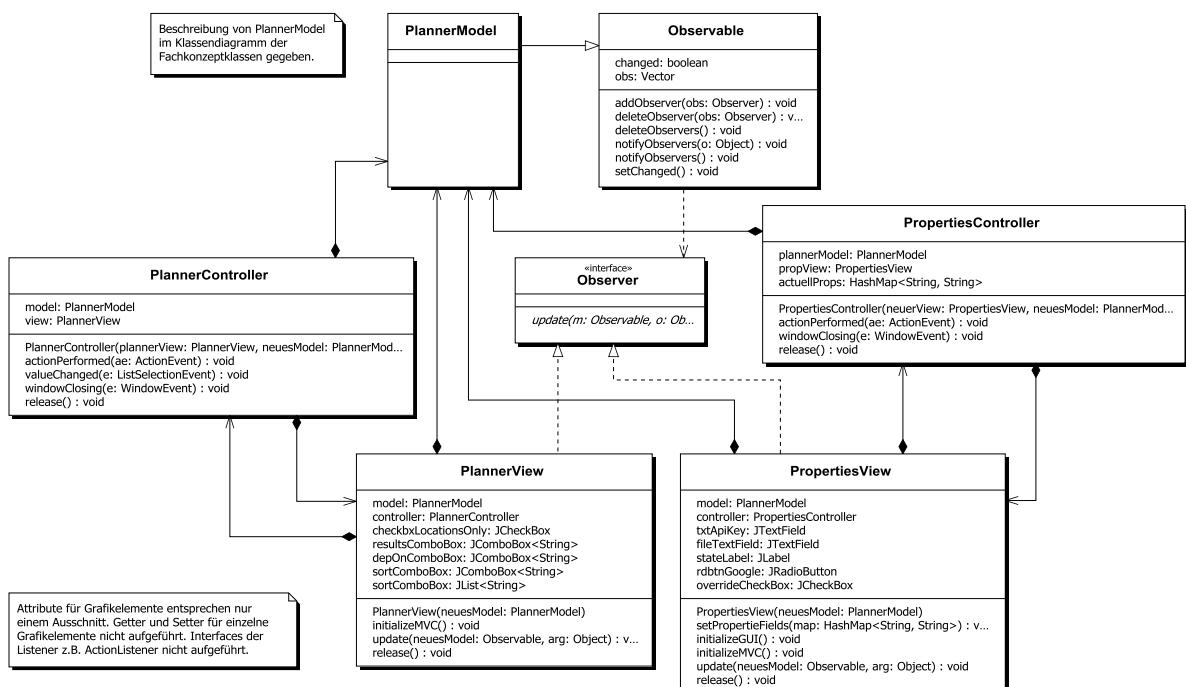


Abbildung 8 Klassen der Benutzeroberfläche

4. Implementierung

4.1 Routenplaneralgorithmus

Der Algorithmus zur Berechnung der Routen für Vor-, Haupt- und Nachspeiseteams wird an dieser Stelle erläutert. Annahmen, die speziell im Lösungsansatz der Routenplanung dieser Arbeit getroffen werden, sind in Abschnitt 5.2.1 beschrieben.

Zunächst ist es von großer Bedeutung, die Nebenbedingung (1) aus Abschnitt 2.1 noch einmal genauer zu betrachten. Um diese Bedingung zu erfüllen, wird die Berechnung der Routen für Hauptspeiseteams im ersten Schritt durchgeführt. Dies wird wie folgt realisiert:

1.1 Es erfolgt eine Vorauswahl der Orte¹⁰, welche die geringste Entfernung zu allen anderen Orten aufweisen. Die Annahme ist dabei, dass für diese Orte im späteren Verlauf die potenziell kürzesten Routen entstehen. Für jeden Ort i wird die Summe s_i über die Distanzen zu allen anderen Orten bestimmt. Das Prinzip wurde von der Smallest Sum Insertion Heuristik (siehe Abschnitt 2.3.2) übernommen. Der Unterschied zum beschriebenen Verfahren besteht darin, dass Orte nicht zu einer Tour hinzugefügt, sondern in die Menge der Hauptspeiseorte aufgenommen werden.

Mit der definierten Entfernungsmatrix in Abschnitt 2.3.1 ergibt sich Gleichung (4.1).

$$s_i = \sum_{l=0}^n m_{i,l} = \sum_{l=0}^n m_{l,i}; i \in \{1, \dots, n\} \quad (4.1)$$

Dabei werden die $n/3$ Orte mit den kleinsten Distanzen ausgewählt und provisorisch als Hauptspeiseorte festgelegt. Diese Orte definieren die Menge H . Die $2n/3$ verbleibenden Orte definieren eine Menge Z . Durch dieses Verhalten werden die Auswahlmöglichkeiten der Hauptspeiseorte minimiert.

1.2 Um eine Route über Vor-, Haupt- und Nachspeise zu erstellen, werden im Folgenden für jeden in Schritt 1.1 ausgewählten Ort $h \in H$ die zwei am nächsten zu ihm gelegenen Folgeorte $z_1, z_2 \in Z, z_1 \neq z_2$ ausgewählt. Das Wählen eines Folgeortes mit kleinster Entfernung entspricht der Nearest Neighbor Heuristik bzw. der Nearest Insertion (siehe Abschnitt 2.3.2). Es ist darauf zu achten, dass bereits ausgewählte Folgeorte für andere Orte aus H nicht mehr infrage kommen.

¹⁰ Es ist zu beachten, dass jedem Team genau ein Ort zugeordnet ist. Ein Ort spezifiziert also ebenfalls das Team, welches dort beheimatet ist.

1.3 In diesem Schritt werden die eigentlichen Vor-, Haupt- und Nachspeiseorte festgelegt. Außerdem erfolgt die endgültige Festlegung der Routen der Hauptspeiseteams. Jede Gruppe, bestehend aus h und den zwei zugewiesenen Orten z_1 und z_2 , wird so zu einer Route verbunden, dass für die Summe s der Distanzen gilt:

$$s = \min(m_{a,b} + m_{b,c}), a, b, c \in \{z_1, z_2, h\} \text{ und } a, b, c \text{ verschieden} \quad (4.2)$$

b stellt den „mittleren“ Ort der Route dar. Erst jetzt wird b als Hauptspeiseort und das beheimatete Team als Hauptspeiseteam definiert. Die Route von b wird von a nach b über c festgelegt. Die angrenzenden Orte a und c werden zufällig als Vor- und Nachspeiseorte, die dort beheimateten Teams als Vor- und Nachspeiseteam definiert. Damit sind die Routen für die Hauptspeiseteams festgelegt.

Um Nebenbedingung (2) auch in den folgenden Schritten aufrechterhalten zu können, werden folgende Maßnahmen ergriffen. Das Hauptspeiseteam aus b wird in den beiden Orten a und c seiner Route als gesehen markiert. Im Hauptspeiseteam werden wiederum Vor- und Nachspeiseteam aus a bzw. c als gesehen markiert. Im Hauptspeiseteam werden die durchlaufenen Orte in Feldern für Vor-, Haupt und Nachspeiseort vermerkt.

In Schritt zwei erfolgt die Festlegung der Routen für Vor- und Nachspeiseteams unter Einhaltung von Nebenbedingung (2). Aus diesem Grund wird in jedem Team-Objekt vermerkt, welche Teams auf der Route bereits gesehen wurden. Die Berechnung der Routen für Vorspeiseteams wird wie folgt realisiert.

2.1 Die Vorspeiseteams benötigen zuerst einen Folgeort, an dem sie die Hauptspeise einnehmen. Hier kommt wiederum die Nearest Neighbor Heuristik zum Einsatz. Es wird jeweils der Hauptspeiseort mit der kürzesten Entfernung ausgewählt. Wird ein Hauptspeiseort für ein Vorspeiseteam ausgewählt, so steht er den anderen Vorspeiseorten nicht mehr zur Auswahl zur Verfügung. Er wird also aus der zugehörigen Liste entfernt. **Aus diesem Grund ist die letztendliche Verteilung der Vorspeiseorte auf die Hauptspeiseorte stark von der Reihenfolge abhängig, in der die Vorspeiseorte durchlaufen werden.** Um flexible Ergebnisse zu erhalten und für eine möglichst faire Verteilung, wird diese Reihenfolge zufällig gewählt. Dies entspricht dem Prinzip der Random Insertion (siehe Abschnitt 2.3.2). Die Menge der Hauptspeiseorte wird dagegen für jeden Vorspeiseort sequenziell durchlaufen. Zusätzlich muss darauf geachtet werden, dass das Hauptspeiseteam nicht bereits vom Vorspeiseteam gesehen wurde, um (2) zu erfüllen. In diesem Fall wird der entsprechende Hauptspeiseort während der Auswahl übersprungen.

2.2 Als Nächstes wird den Vorspeiseteams ein Nachspeiseort zugeteilt. Dieser Schritt erfolgt weitgehend analog zu Schritt 2.1, allerdings werden jetzt die Nachspeiseorte ausgewählt, welche die kürzeste Entfernung zu den in Schritt 2.1 bestimmten Hauptspeiseorten

besitzen. Weiterhin ist zu beachten, dass das Vorspeiseteam auf seiner bisherigen Route sowohl an seinem Heimatort ein Hauptspeiseteam bewirkt hat, als auch am Hauptspeiseort zu Gast bei einem Hauptspeiseteam war. Das heißt, alle Nachspeiseorte, die diese beiden Teams zu Gast haben, werden für eine Auswahl ausgeschlossen.

Die Schritte 2.3 und 2.4 befassen sich mit der Berechnung von Routen der Nachspeiseorte. Diese verläuft analog den Schritten 2.1 und 2.2. Eine Beschreibung kann aus den vorherigen Schritten abgeleitet werden. Bereits an den Schritten 2.1 und 2.2 ist zu erkennen, dass die Möglichkeiten, einen Nachfolgeort auszuwählen, durch die zunehmende Menge bereits gesehener Teams pro Team in jedem Schritt abnehmen. Aus diesem Grund besteht die Möglichkeit, dass in einem Schritt kein passender Folgeort für ein Team gefunden wird. Eine erste Maßnahme, dieses Problem zu beheben, ist eine erneute Berechnung mit abgeänderter Reihenfolge des Durchlaufens der Orte¹¹ durchzuführen. Außerdem werden die Schritte 2.1 bis 2.4 durch rekursive Aufrufe mit Backtracking folgendermaßen realisiert:

Da die einzelnen Schritte analog verlaufen, werden sie in ein und derselben Java-Methode implementiert. Die Schritte werden als Zustände codiert und der Methode als Parameter übergeben, um anzusehen, von welchen Heimatorten zu welchen Folgeorten eine Berechnung stattfinden soll. Des Weiteren wird jeweils die aktuelle Teamliste mit den Zuständen der einzelnen Teams in Form einer Kopie übergeben.

Wird nun im aktuellen Zustand auch nach mehrmaliger Berechnung mit unterschiedlicher Reihenfolge der Orte¹² keine konsistente Zuordnung von Folgeorten gefunden, so wird in den vorherigen Zustand zurückgekehrt. Nach diesem Backtracking erfolgt schließlich wiederum eine Neuberechnung von Folgeorten. Für die Anzahl dieser Neuberechnungen pro Zustand kann ein fixer Wert festgelegt werden. Das Backtracking erfolgt rückwirkend bis Schritt 2.1. Für die Anzahl der Neuberechnung von Hauptspeiserouten kann ebenfalls ein fester Wert angegeben werden. Außerdem besteht die Möglichkeit, den Algorithmus nach einer festgelegten Anzahl berechneter Lösungen zu terminieren.

4.2 Umsetzung in Java

4.2.1 Heuristiken

Die Umsetzung der in Abschnitt 2.3.2 erwähnten Heuristiken wird anhand der Methode „`setMinimalSumCourses()`“ gezeigt. In erster Linie wird die Smallest Sum Insertion erläutert.

11 Siehe fett gedruckte Markierung Schritt 2.1

12 Siehe fett gedruckte Markierung Schritt 2.1

Das Verhalten der Nearest Neighbor Heuristik bzw. Nearest Insertion lässt sich ableiten. Die Methode wird zur Berechnung von Schritt 1.1 des Routenplanungsalgorithmus genutzt. Die Zeilen 5 bis 12 befassen sich mit der Bestimmung der Distanzsummen. Zur Speicherung wird zunächst das Array „sumDistances“ initialisiert. Das Attribut „distances“ entspricht dem berechneten Distanzen-Array eines „DistanzCalculator“-Objektes. Über die zwei „For“-Schleifen werden alle Distanzen, die von einem bestimmten Ort ausgehen aufsummiert. Die Variable „countMainCourse“ wird auf $n/3$ gesetzt, wobei n die Anzahl der Orte ist, da genau $n/3$ Hauptspeiseorte auszuwählen sind. Die Zeilen 20 bis 34 befassen sich mit der Auswahl der Orte mit kürzester Distanzsumme und entsprechen dem Verhalten der Nearest Insertion mit dem Unterschied, dass hier die kürzesten Distanzsummen gesucht sind. Dafür wird das Array mit Distanzsummen kopiert. Im kopierten Array werden im Folgenden die Distanzen ausgewählter Hauptspeiseorte in Zeile 31 auf den größtmöglichen Integer-Wert gesetzt. Damit werden diese Orte bei einem erneuten Durchlauf nicht mehr ausgewählt. Sie werden sozusagen aus der Liste gelöscht. Das Array wird $n/3$ -mal durchlaufen (Zeile 20). Die Variable „min“ hält stets den Index des Ortes mit der zuletzt bestimmten kleinsten Distanzsumme. Die kleinste Distanzsumme eines Durchlaufs wird in der Variablen „temp“ gespeichert. Nach Durchlauf des Arrays ist die kleinste Summe für diesen Durchlauf gefunden. Der zugehörige Ort wird in einem zweidimensionalen Boolean-Array „mainCourses“ in der Diagonalen auf den Wert „true“ gesetzt. Dieses Array entspricht einer quadratischen Matrix. Die Länge entspricht der Anzahl der Orte. Es wird für weitere Berechnungen im Laufe des Algorithmus benötigt.

```

/**
 * Hauptspeiseteams sollen nach Möglichkeit die Routen mit kleinstem
 * zurückgelegten Weg erhalten.
 * Deshalb erfolgt hier eine Vorauswahl dieser.
 * Vorgehen: Berechnung der Summe aller Distanzen eines Ortes zu allen anderen
 * Orten.
 * Sei n die Anzahl der Veranstaltungsorte bzw. -teams.
 * Auswahl der n/3 Orte mit den kleinsten Summen. Diese Orte werden für eine
 * Vorauswahl der Hauptspeiseorte (provisorisch) ausgewählt.
 * Annahme: Da diese Orte die kleinsten Entfernung zu allen anderen
 * Orten besitzen, entstehen für sie die potenziell kleinsten Routen.
 * @param mainCourses zweidimensionales Array, Orte mit kleinsten Summen
 * sind in der Diagonale true gesetzt. Der Index entspricht der home ID des
 * jeweiligen Teams.
 */
1 public void setMinimalSumCourses( boolean[][] mainCourses ) {
2
3     //Array zum Speichern von Distanzsummen
4     int[] sumDistances = new int[this.distances.length];
5
6     //Aufsummieren der Einzeldistanzen pro Veranstaltungsort
7     for( int i = 0; i < sumDistances.length; i++ ){
8
9         for( int j = 0; j < sumDistances.length; j++ ){
10
11             sumDistances[i] += this.distances[i][j];

```

```

12
13         }
14     }
15
16     //Anzahl auszuwählender Hauptspeiseorte = 1/3 der Gesamtanzahl von Orten
17     int countMainCourse = sumDistances.length / 3;
18
19     //Kopieren des Summen-Arrays
20     int[] minSum = Arrays.copyOf( sumDistances, sumDistances.length );
21
22     //Auswählen der n/3 Orte mit den kleinsten Distanzsummen.
23     for( int k = 0; k < countMainCourse; k++ ){
24
25         //temp = aktuell kleinste Summe
26         int temp = minSum[0];
27         //min = Index von temp
28         int min = 0;
29
30         //Durchlauf aller Orte
31         for( int l = 1; l < minSum.length; l++ ){
32
33             //Prüfe, ob kleinste Summe größer als aktuelle Summe
34             if( temp > minSum[l] ){
35
36                 //Aktualisiere kleinste Summe und Index
37                 min = l;
38                 temp = minSum[l];
39
40             }
41
42         }
43         //Setzen der Distanzsumme des ausgewählten Ortes auf maximal Wert.
44         //=> dieser wird nicht erneut ausgewählt.
45         minSum[min] = Integer.MAX_VALUE;
46
47         //Länge entspricht Anzahl der Orte in beiden Dimensionen
48         //Ausgewählter Ort wird in der Diagonale auf true gesetzt.
49         mainCourses[min][min] = true;
50     }
51
52 }
```

Das Prinzip der Random Insertion, in Abschnitt 2.3.2 erläutert, wird durch die Codezeile
`Collections.shuffle(mainList);`
realisiert. Die „mainList“ entspricht der Liste der Vorspeiseteams aus Schritt 2.1 des Routenplanungsalgorithmus. Diese Liste wird zufällig permutiert, damit die Vorspeiseteams in unterschiedlichen Kombinationen durchlaufen werden.

4.2.2 Benutzeroberfläche

Die Umsetzung der grafischen Benutzeroberfläche wird anhand von Screenshots beschrieben. Abbildung 9 zeigt das Hauptfenster des Planers, bezeichnet als „Dinner-Hopping Planner v1.0“ und das Fenster der Kartendarstellung mit der Bezeichnung „Dinner-Hopping Viewer v1.0“.

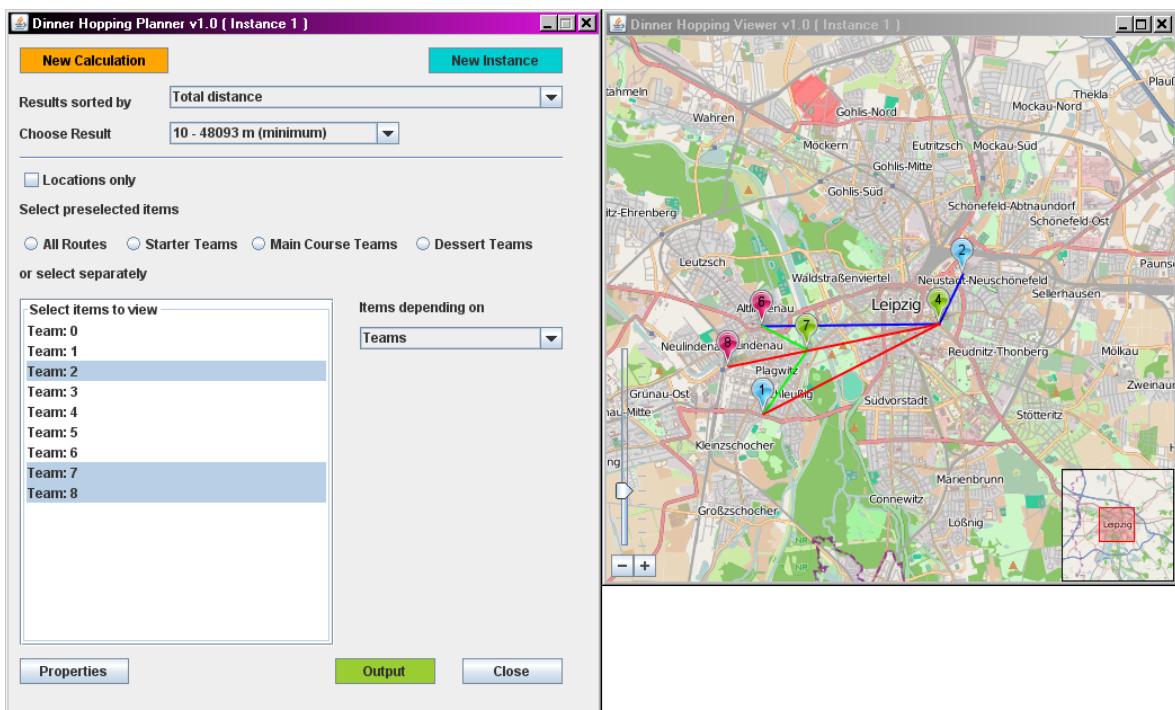


Abbildung 9 Dinner-Hopping Planner (links) und Viewer (rechts)

Das Hauptfenster ist in einen oberen Teil, welcher die Filterung und Auswahl von Routenplanungen behandelt und einen unteren Teil, der sich mit Orten und Teams befasst, die im Viewer angezeigt werden sollen, unterteilt. Die einzelnen Komponenten sind in ein „ContentPane“ eingebettet, welches sich wiederum im zugrunde liegenden „JFrame“ befindet. Der obere Teil des Fensters enthält:

- „New Calculations“-Button: Durchführung einer neuen Routenberechnung
- „New Instance“-Button: Öffnen einer neuen Instanz des Planers mit dem aktuell berechneten Datensatz
- „Results Sorted By“-Auswahlbox: Filterung der Resultate nach zurückgelegter Gesamtstrecke oder längster Strecke der berechneten Routenplanungen
- „Choose Result“-Auswahlbox: Auswahl einer berechneten Lösung

Der untere Teil gliedert sich in:

- „Locations Only“-Checkbox: Zeige ausschließlich Orte im Viewer
- „Select preselected items“: Anzeige vorausgewählter Teams im Viewer
- „Select separately“-Liste: Liste („JList“) zum Auswählen einzelner Teams oder Orte
- „Items depending on“-Auswahlbox: Liste beruht auf Teams oder Orten
- „Properties“-Button: Anzeige des Fensters mit Programmeigenschaften
- „Output“-Button: Schreiben der aktuell ausgewählten Routenplanung in eine CSV-Datei

Die „JList“ ist in ein „JScrollPane“ eingebettet.

Zur Realisierung der Kartendarstellung wird das „JXMapKit“ als Swing-Komponente zu einem „JFrame“ hinzugefügt. Die dadurch entstandene Karte beinhaltet die Hauptkarte, eine kleinere Übersichtskarte in der rechten unteren Ecke sowie die Zoom-Leiste am linken Rand. Das Einzeichnen von Inhalten geschieht über eigens definierte „Painter“ (Overlays). Das durch den „Properties“-Button aufgerufene Menü dient zum Einstellen der Programmeigenschaften. Dieses wurde mit einem „JTappedPane“ erstellt. Jeder Tab enthält wiederum ein „ JPanel“ mit den individuellen Komponenten. Abbildung 10 zeigt zwei Tabs. Der obere Tab befasst sich mit dem Import. Im Textfeld wird die Input-Datei angegeben. Durch das Betätigen von „Apply File“ wird geprüft, ob die Datei vorhanden ist. Eine Statusanzeige erfolgt darunter. Der untere Tab zeigt die Möglichkeiten der Geocodierung. Mit den Radiobuttons ist die Auswahl eines Geocodierers möglich. Der für den CloudMade-Dienst erforderliche API-Key wird im „JTextField“ eingegeben und über den „Apply Key“-Button validiert. Der Status wird über das mit „State:“ bezeichnete „JLabel“ ausgegeben.

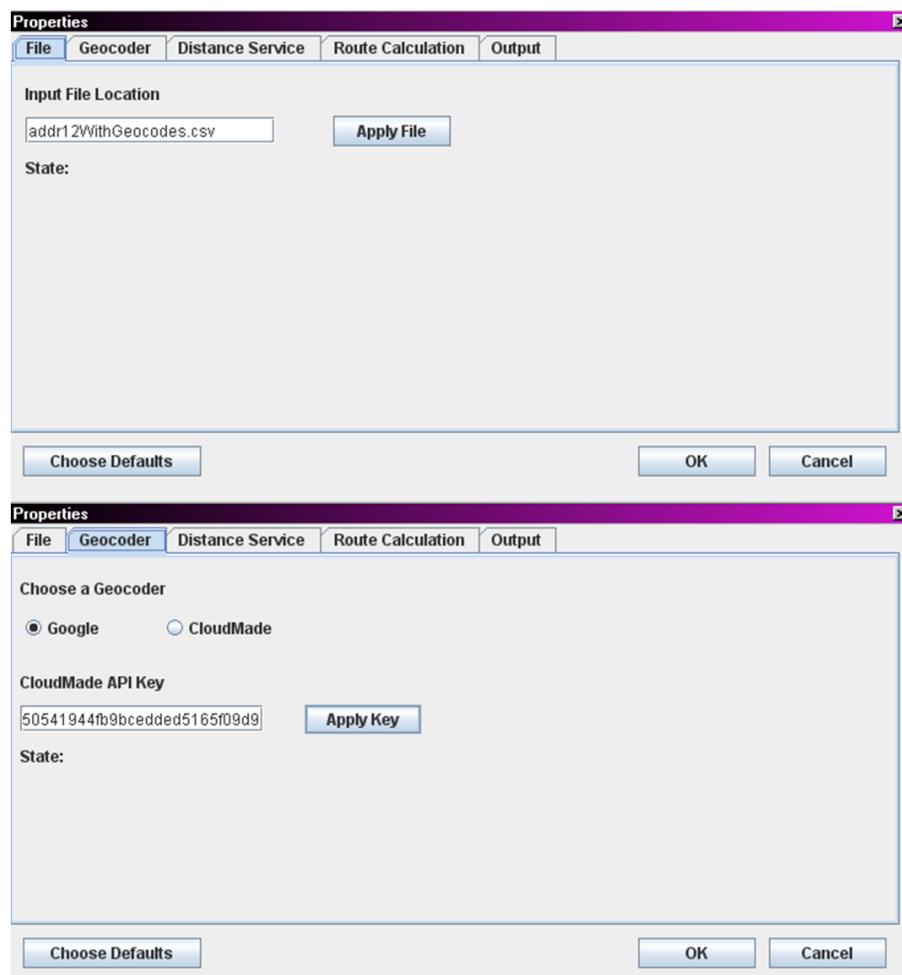


Abbildung 10 „Properties“-Fenster mit Tabs für Input-Datei (oben) und Geocodierung (unten)

4.2.3 Kartendarstellung mit dem JXMapKit

Um die Routen einzelner Teams im Viewer anzuzeigen, wurde die folgende Methode implementiert.

```

1 /**
2 * Zeige Routen der ausgewählten Teams.
3 * @param teamIds IDs (Indices) der Teams, die angezeigt werden sollen.
4 * @param waypointOnly zeige ausschließlich Orte
5 */
6 public void setTeamRoute(ArrayList<Integer> teamIds, boolean waypointOnly ){
7
8     //Liste mit Routen, die im Moment angezeigt werden.
9     //Eine Route besteht aus drei Veranstaltungsorten.
10    this.routesToView.clear();
11
12    //Liste mit Team IDs deren Routen angezeigt werden sollen.
13    this.idList = teamIds;
14
15    //Route des Teams mit jeweiliger ID wird ausgelesen.
16    for( int id : teamIds ){
17
18        ArrayList<Integer> actuellRoute = new ArrayList<Integer>();
19        //Vorspeiseort wird ausgelesen und zur aktuellen Route hinzugefügt
20        actuellRoute.add( this.routeTeamList.get(id).starter );
21        //Hauptspeiseort wird ausgelesen
22        actuellRoute.add( this.routeTeamList.get(id).mainCourse );
23        //Nachspeiseort wird ausgelesen
24        actuellRoute.add( this.routeTeamList.get(id).dessert );
25        //Hinzufügen der aktuellen Route zum Routen-Array
26        routesToView.add(actuellRoute);
27    }
28
29    //Zeichne aktuelle Wegpunkte, Routen und Label
30    //Zeige nur Orte => Overlay der Orte + Overlay der Ortsbezeichnungen setzen
31    //cp = CompoundPainter fügt mehrere Painter zu einem zusammen
32    if( waypointOnly ) cp.setPainters( waypointOverlay, labelOverlay );
33
34    //zeige Routen => Overlay der Orte + Overlay der Ortsbezeichnungen +
35    //Overlay der Orte
36    else cp.setPainters( lineOverlay, waypointOverlay, labelOverlay );
37
38    //Füge Painter zum JXMapKit hinzu
39    kit.getMainMap().setOverlayPainter(cp);
40
41 }
42
43 }
```

Die Parameter der Methode sind die IDs der Teams, von welchen die Routen angezeigt werden sollen. Außerdem wird durch „waypointsOnly“ signalisiert, ob komplette Routen oder ausschließlich die besuchten Orte anzuzeigen sind. Zeile 5 leert die Liste mit bisher angezeigten Routen („routesToView“). Zeile 8 setzt das Attribut, welches Team-IDs hält, auf die übergebene ID-Liste. Im Folgenden wird über die ID-Liste iteriert und die Route für das Team mit der jeweiligen ID zusammengestellt. Dazu wird auf die „routeTeamList“, welche die aktuelle Routenplanung mit allen Teams enthält, zugegriffen. Die erstellte Route wird der Liste „routesToView“ hinzugefügt. Die Zeilen 27 und 31 beschäftigen sich mit dem Setzen der Overlays, die auf die Karte gezeichnet werden sollen. Da im „JXMapKit“ („kit“) nur ein

„Painter“ (Overlay) zu einem Zeitpunkt gesetzt werden kann, wird hier ein „CompoundPainter“ („cp“) verwendet, welcher mehrere „Painter“ zusammenfügt. Im „CompoundPainter“ wird je nach Wert von „waypointsOnly“ das Routen-Overlay, Marker-Overlay oder Label-Overlay gesetzt. Schließlich muss in Zeile 34 dem „JXMapKit“ der „CompoundPainter“ hinzugefügt werden. Die jeweiligen Overlays sind damit im Viewer sichtbar.

4.2.4 Realisierung der Programmeigenschaften

Die folgende Methode beschreibt das Schreiben der Programmeigenschaften in eine Java-Properties-Datei.

```


    /**
     * Schreiben der übergebenen Properties in die Planner.properties Datei.
     * @param map Properties
     */
1 public void writeProps( HashMap<String, String> map ) {
2
3     //Schreiben eines Charakter-Streams (java.io.Writer)
4     Writer writer = null;
5
6     try{
7
8         //Erzeugen der Property-Datei
9         writer = new FileWriter( "Planner.properties" );
10        // (Key, Value) Liste der Programmeigenschaften
11        //(java.util.Properties)
12        Properties prop1 = new Properties();
13        prop1.setProperty( "Geocoder", map.get("Geocoder") );
14        prop1.setProperty( "Distance", map.get("Distance") );
15        prop1.setProperty( "Route", map.get("Route") );
16        prop1.setProperty( "APIKEY", map.get("APIKEY") );
17        prop1.setProperty( "Input File", map.get("Input File") );
18        prop1.setProperty( "Output File", map.get("Output File") );
19        prop1.setProperty("Override", map.get("Override"));
20        //Schreiben der Eigenschaften in die erzeugte Datei
21        prop1.store( writer, "--Dinner-Hopping Planner Properties--" );
22
23    }catch ( IOException e ){
24        e.printStackTrace();
25    }finally{
26        try {
27            writer.close();
28        }catch ( Exception e ) {
29            e.printStackTrace();
30        }
31    }
32 }

```

Als Parameter erhält diese Methode eine „HashMap“. Der Key entspricht der Bezeichnung der Eigenschaft als String. Das Value ist der Wert-String. Zeile 4 initialisiert einen „Writer“ zum Schreiben eines Character-Streams. In Zeile 9 wird schließlich ein „FileWriter“ instanziert. Der Konstruktor erstellt eine Datei mit dem angegebenen Dateinamen, falls diese nicht bereits existiert. Als Nächstes wird ein „Properties“-Objekt erstellt. Dieses besteht aus einer

Eigenschaftsliste. Die einzelnen Werte der „HashMap“ werden an die Eigenschaftsliste mithilfe der „setProperty()“ Methode übergeben. Der „store()“-Methode des „Properties“-Objektes wird das „FileWriter“-Objekt übergeben und die Eigenschaftsliste in die vom „FileWriter“ erstellte Datei geschrieben (Zeile 21). Damit können konfigurierte Eigenschaften in eine externe Datei übernommen und bei einem erneuten Programmstart wieder zur Verfügung gestellt werden.

4.2.5 Parsen einer Antwort des CloudMade-Geocodierers

Im Folgenden ist ein Ausschnitt aus der Methode „getGeocodes()“ der Klasse „CloudMadeGeocoder“ gegeben. Es wird der Verbindungsauflauf sowie das Auslesen der Antwort des CloudMade-Dienstes verdeutlicht.

```

1 //Verbindung herstellen (maximal 3 Verbindungsversuche)
2 while(attempt < 3) {
3
4     try{
5         //Verbindungsobjekt erstellen
6         URLConnection con = url.openConnection();
7         //Nutze das Objekt für den Input
8         con.setDoInput(true);
9         //Verbindung herstellen
10        con.connect();
11        //inputStream, der von der Verbindung liest
12        inputStreamJson = con.getInputStream();
13        break;
14
15    } catch(IOException ioe){
16        System.out.println("Connection Error "+(attempt+1)+"! ");
17    }
18    attempt++;
19
20 }
21 //Programm beenden, bei 3 fehlgeschlagenen Verbindungen
22 if(attempt == 3) {
23     System.out.println("Internet connection error or server not available!");
24     return false;
25 }
26
27 //String mit Antwort des Dienstes
28 String jsonTxt = "";
29 //Antwort als String übernehmen
30 try {
31     jsonTxt = IOUtils.toString( inputStreamJson );
32 } catch (IOException e) {
33     System.out.println("Can't convert source to string!");
34     return false;
35 }
36
37 //Umwandlung in JSON
38 JSONObject json = (JSONObject) JSONSerializer.toJSON( jsonTxt );
39
40 //Erzeuge ein JSONArray features
41 JSONArray features = json.getJSONArray("features");
42
43 //Anzahl der Geocodierungsresultate
44 int results = features.size();
```

```

45
46 //Nimm das erste Resultat und erzeuge aus diesem ein JSONObject.
47 JSONObject obj = (JSONObject) features.get(0);

```

In Zeile 6 wird ein Verbindungsobjekt zum CloudMade-Client über ein im Vorfeld erstelltes URL-Objekt erzeugt. Das URL-Objekt enthält die Geocodierungsanfrage an den CloudMade-Dienst. Über die Methode „connect()“ des Verbindungsobjektes wird eine Verbindung hergestellt. Das Rückgabeformat wurde im Vorfeld durch einen URL-Parameter auf JSON gesetzt. Die HTTP-Antwort wird in Zeile 12 zunächst auf einen „InputStream“ umgeleitet. Über diesen kann das Ergebnis im Folgenden entgegengenommen werden. Wird hier eine „Exception“ geworfen, so konnte der HTTP-Request nicht ausgeführt werden und die äußere „While“-Schleife wird erneut durchlaufen. In Zeile 2 ist erkennbar, dass maximal 3 Verbindungsversuche unternommen werden, bevor die Geocodierung abgebrochen wird (Zeile 22). Bei korrekter Erstellung des „InputStreams“ wird die Schleife durch das „break“ (Zeile 13) verlassen. Zeile 30 bis 35 wandeln den zurückgelieferten Stream in ein String-Objekt um. Dieses bildet die Grundlage zum Parsen der JSON-Antwort mit der an dieser Stelle genutzten JSON Library v.2.4¹³. Über den bereitgestellten JSON-Serialisierer wird ein JSON-Objekt (Zeile 38) aus der JSON-Antwort (String aus Zeile 31) erstellt. Je nach Aufbau der Antwort können daraufhin einzelne Bestandteile ausgelesen werden. Beispielsweise enthält die Antwort des CloudMade-Dienstes ein Array namens „features“, das einzelne Geocodierungsergebnisse liefert. Aus diesem Grund wird in Zeile 41 ein JSON-Array, identifiziert durch seinen Namen „features“, erstellt. In Zeile 47 wird das erste Objekt dieses Arrays, was dem ersten Resultat der Geocodierungs-Anfrage entspricht, angefragt und wiederum in ein JSON-Objekt umgewandelt, um weiter verarbeitet werden zu können. Auf diese Weise ist es möglich, über die einzelnen Geocodierungsergebnisse zu iterieren. Die erzeugten Objekte werden danach auf die enthaltenen Geokoordinaten untersucht, was hier nicht dargestellt ist.

4.2.6 Distanzbestimmung mit Osm2po

Der Osm2po-Dienst benötigt ein lokal gespeichertes Graph-File, um auf diesem Berechnungen durchführen zu können. Der Ablauf einer Distanzbestimmung wird an dieser Stelle kurz erläutert. Die Methode „calculateDistances()“ nimmt die Liste der Veranstaltungsorte entgegen. Zunächst wird ein Array zum Abspeichern berechneter Distanzen initialisiert. Zeile 5 zeigt die Erstellung eines „File“-Objektes. Hier muss das angesprochene Graph-File initialisiert werden. Durch die Erstellung des Graph-Objektes (Zeile 7) wird der Graph in den

¹³ Basierend auf der Arbeit von Douglas Crockford siehe <http://www.json.org/java/>

Arbeitsspeicher geladen. Daraufhin erfolgt das Initialisieren des Routers. Hier wird der „DefaultRouter“ initialisiert. Dabei wird der Dijkstra-Algorithmus als Grundlage für das Routing gesetzt. Als Nächstes werden die einzelnen Parameter in Form eines „Properties“-Objektes festgelegt (Zeile 12 bis 16). Im Beispiel wird der kürzeste Pfad zum Ziel gesucht, Einbahnstraßen werden ignoriert und als Heuristik wird der A*-Algorithmus (Erweiterung des Dijkstra Algorithmus mit Verkürzung der Laufzeit) verwendet. Ab Zeile 19 wird über die einzelnen Orte iteriert, um die Entfernung von jedem Ort zu jedem anderen zu berechnen (zwei „For“-Schleifen). Da die entstehende Matrix symmetrisch ist, wird nur eine Hälfte dieser berechnet (Zeile 22).

```

/**
 * Berechnung von Distanzen von jeder geocodierten Adresse zu jeder anderen.
 * @param geoAddressList
 * @return Integer-Matrix mit Distanzen
 */
1 public int[][] calculateDistances( ArrayList<GeoAddress> geoAddressList ) {
2     //initialisiert Array zum Halten der zu berechnenden Distanzen
3     int[][] distances = new
4         int[geoAddressList.size()][geoAddressList.size()];
5     //Erzeuge Objekt des Graph-Files
6     File graphFile = new File("../lib/hh/hh_2po.gph");
7     //Lade den Graphen in den Speicher
8     Graph graph = new Graph(graphFile);
9     //Wähle den DefaultRouter (Dijkstra with specials)
10    DefaultRouter router = new DefaultRouter();
11
12    //mögliche Parameter für den DefaultRouter
13    Properties params = new Properties();
14    params.put("findShortestPath", false); //kürzester Pfad
15    params.put("ignoreRestrictions", false);
16    params.put("ignoreOneWays", false); //keine Einbahnstraßen
17    params.put("heuristicFactor", "1.0"); // 0.0 Dijkstra, 1.0 good A*
18
19    //Distanzen zwischen allen Orten
20    for(int j = 0;j < geoAddressList.size(); j++){
21
22        for(int i = 0; i < geoAddressList.size() ; i++){
23            if(i <= j) continue;
24
25            //Finde am nächsten gelegenen Vertex zur Geokoordinate
26            int sourceId = graph.findClosestVertexId(
27                geoAddressList.get(i).getLat(),
28                geoAddressList.get(i).getLng());
29            int targetId = graph.findClosestVertexId(
30                geoAddressList.get(j).getLat(),
31                geoAddressList.get(j).getLng());
32            //Nutze den zwischengespeicherten Graph mehr als einmal
33            router.reset();
34
35            //Suche nach einer Route
36            router.traverse(graph, sourceId, targetId,
37                            Float.MAX_VALUE, params);
38
39            double totalKm = 0.0;
40            //Zielvertex gefunden!
41            if (router.isVisited(targetId)){
42
43                //Erzeuge den Pfad bestehend aus IDs der einzelnen Segmente
44                int[] path = router.makePath(targetId);
45
46                //Summiere Distanzen der einzelnen Segmente der Route

```

```

41         for (int k = 0; k < path.length; k++) {
42
43             //Segmente mit aktueller ID
44             RoutingResultSegment rrs =
45                 graph.lookupSegment(path[k]);
46             //Länge des aktuellen Segmentes
47             double km = rrs.getKm();
48
49             totalKm += km;
50
51         }
52
53         //Füge Länge in symmetrische Distanzmatrix ein
54         distances[j][i] = (int) (totalKm * 1000);
55         distances[i][j] = (int) (totalKm * 1000);
56
57     }
58 }
59 return distances;
60 }
```

Die Methode „findClosestVertexId()“ findet das zu den übergebenen Geokoordinaten am nächsten gelegene Vertex im Graphen. Nach dem Zurücksetzen des Routers, was nötig ist, um den Router mehrmals für unterschiedliche Koordinaten verwenden zu können, wird der Graph schließlich nach Routen durchsucht (Zeile 31). Mit „isVisited(targetId)“ wird überprüft, ob das Zielvertex gefunden wurde und die Route berechnet werden konnte. In Zeile 38 wird der Pfad bis zum Zielvertex mit allen dazwischenliegenden Vertices erzeugt. Dieser wird benötigt, um Parameter, wie die Länge jedes einzelnen Streckenabschnitts, bestimmen zu können. Die Zeilen 41 bis 50 zeigen das Aufsummieren der einzelnen Längen der Streckenabschnitte einer Route. Schließlich wird die errechnete Summe in das Distanz-Array eingetragen (Zeile 54f).

4.3 Beispieldatensatz

1	Zeitattempel,Teilnehmer 1,Teilnehmer 2,Straße,Hausnummer,Telefon,Email,Sonstiges
2	08-08-12,"Max Mustermann", "Eric Baumgarten", "Erich-Zeigner-Allee", "5", "0341 93443458", "max.mustermann@web.de",
3	08-08-12, "Hans Treu", "Johannes Eichner", "Erich-Zeigner-Allee", "200", "0160455445694", "Joh96@gmx.com", "1 mal Vegetarier"
4	,"Lisa Schleußig", "Peter Gefreu", "Eisenbahnstr.", "1", 03449389058, "PL78@yahoo.de",
5	08-08-12, "Julia Kern", "Sabine Unruhe", "Eisenbahnstr.", "150", "0365 6549848464", "Sabine-Unruhe@web.de", eine vegetarierin
6	,"Kora Meissen", "Frank Raubein", "Johannisplatz", "5", "036548915135", "Meissen.89@studserv.uni-leipzig.de",
7	08-08-12, "Franz Zimmermann", "Peter Graumann", "Jahnallee", "5", "01755498494654", "ideserveit@yahoo.de",
8	,"Sivia Höffner", "Eric Trist", "Marktstraße", "5", 01804484684864, "silvia-H@ich.de", "ohne zwiebel und ohne pasta"
9	,"Ilse Meier", "Michael Jackson", "Karl-Heine-Straße", "1", "065874565558", "ilse-Michael@kabelmail.de",
10	08-08-12, "Uwe Baumann", "Gerald Zimmller", "Karl-Heine-Straße", "122", "075801818686", "ZimmllerG88@studserv.uni-leipzig.de",
11	

Abbildung 11 Datensatz mit neun Teams ohne Geokoordinaten

Der hier vorgestellte Beispieldatensatz (siehe Abbildung 11) umfasst neun Teams. Im „Properties“-Fenster wird im Tab „Geocoder“ Google ausgewählt. Der „Distance Service“ ist „Linear Distance“. Nach Drücken des „New Calculation“-Buttons im GUI erscheint die in Abbildung 12 gezeigte Konsolenausgabe. Es sind die einzelnen Teams mit ID, Adressname, Geocodierung und Teammitgliedern zu sehen. Daraufhin wird die Geocodierung gestartet. Dies

ist nicht notwendig, wenn der entsprechende Datensatz bereits Geokoordinaten enthält. Die ersten sieben Adressen werden ohne Probleme geocodiert. Für die Adresse „Karl-Heine-Straße, 1“ werden allerdings zwei mögliche Geokoordinaten gefunden. Daraufhin erscheint das Browserfenster aus Abbildung 12 rechts oben. Hier muss der Nutzer den am besten zur angegebenen Adresse passenden Ort wählen. Die jeweilige ID dieses Ortes nimmt die Konsole als Eingabe entgegen.

```
-----Input-----
Read CSV-File addr.csv!
Team: 0 Address: Erich-Zeigner-Allee, 5
      Members: Max Mustermann, Eric Baumgarten

Team: 1 Address: Erich-Zeigner-Allee,200
      Members: Hans Treu, Johannes Eichner

Team: 2 Address: Eisenbahnstr.,1
      Members: Lisa Schleußig, Peter Gefreu

Team: 3 Address: Eisenbahnstr.,150
      Members: Julia Kern, Sabine Unruhe

Team: 4 Address: Johannisplatz,5
      Members: Kora Meissen, Frank Raubein

Team: 5 Address: Jahnallee,5
      Members: Franz Zimmermann, Peter Graumann

Team: 6 Address: Marktstraße,5
      Members: Sivia Höffner, Eric Trist

Team: 7 Address: Karl-Heine-Straße,1
      Members: Ilse Meier, Michael Jackson

Team: 8 Address: Karl-Heine-Straße,122
      Members: Uwe Baumann, Gerald Zimmersler

No geocoords found! Geocoding necessary!
-----Google geocoding-----
Address: Erich-Zeigner-Allee, 5 successful geocoded! Latitude: 51.334396 Longitude: 12.33969
Address: Erich-Zeigner-Allee,200 successful geocoded! Latitude: 51.31988 Longitude: 12.334504
Address: Eisenbahnstr.,1 successful geocoded! Latitude: 51.34633 Longitude: 12.39509
Address: Eisenbahnstr.,150 successful geocoded! Latitude: 51.344864 Longitude: 12.41603
Address: Johannisplatz,5 successful geocoded! Latitude: 51.336918 Longitude: 12.38785
Address: Jahnallee,5 successful geocoded! Latitude: 51.342846 Longitude: 12.363873
Address: Marktstraße,5 successful geocoded! Latitude: 51.33641 Longitude: 12.3341675

More than one result found!
0: Karl-Heine-Straße,1; 51.33203, 12.347935
1: Karl-Heine-Straße,1; 51.332542, 12.336187

Choose location that is nearest to point of interest!
e
```

Abbildung 12 Input und Geocodierung. Zuerst werden Adressen aus der Input-Datei eingelesen (links). Danach wird die Geocodierung gestartet (unten links). Für die Adresse "Karl-Heine-Straße, 1" werden mehrere Geokoordinaten gefunden. Hier muss der Nutzer den am besten geeigneten Veranstaltungsort aus der statischen Google Karte (oben rechts) auswählen.

Schließlich werden die restlichen Adressen geocodiert (siehe Abbildung 13 links). Daraufhin wird eine CSV-Datei mit den entsprechenden Geokoordinaten erstellt. Diese CSV-Datei ist mit der ursprünglichen identisch bis auf die Ergänzung der Geokoordinaten der Heimatorte am Ende jeder Zeile eines Teams. Eine erneute Geocodierung wird eingespart.

Im nächsten Schritt werden die Distanzen auf Grundlage der euklidischen Distanz ermittelt. Diese sind in der symmetrischen Distanzmatrix in Abbildung 13 links erkennbar. Zeile bzw. Spalte eins steht dabei für Team 1, Zeile bzw. Spalte 2 für Team 2 usw.

Nun werden die einzelnen Routen durch den „Route constructor“ bestimmt. Dabei entstehen hier 16 Lösungen. Schließlich zeigt das GUI des Planungssystems die in Abbildung 13 rechts dargestellte Ausgabe. Es ist zu erkennen, dass Resultat vier das Minimum, beruhend auf der Gesamtdistanz aller Teams, darstellt. Dieses Resultat wird automatisch ausgewählt. Außerdem öffnet sich die Kartenansicht (Viewer).

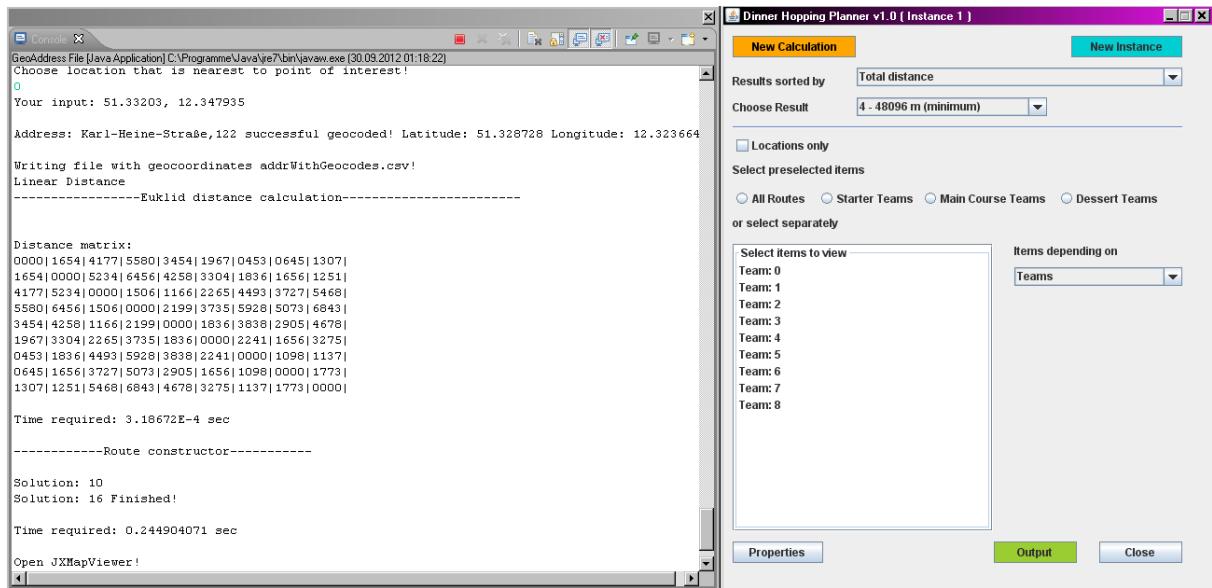


Abbildung 13 Distanzmatrix und Anzahl der gefundenen Lösungen (links). Das GUI zeigt Resultat fünf als Minimum mit der kleinsten Gesamtdistanz (rechts).

Werden in dem GUI die Radio-Buttons „Starter Teams“, „Main Course Teams“ und „Dessert Teams“ in dieser Reihenfolge betätigt, so erscheinen im Viewer nacheinander die Karten aus Abbildung 14 links oben, unten und rechts oben.

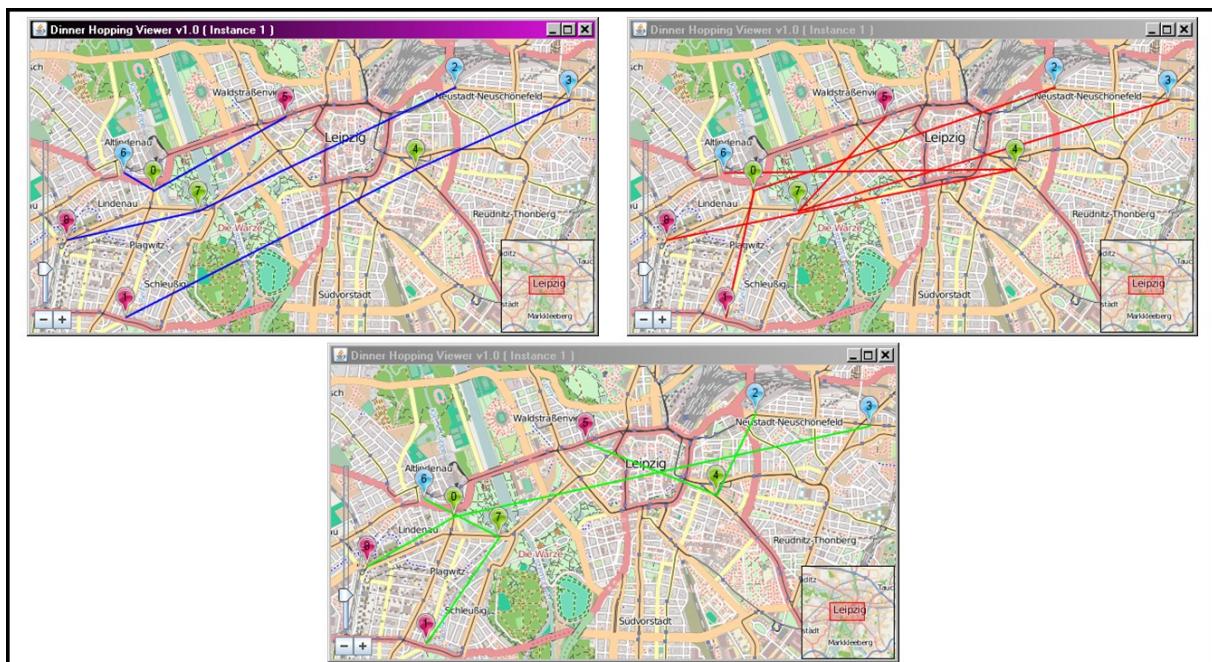


Abbildung 14 Routen von Vorspeiseteams (oben links), Hauptspeiseteams (unten) und Nachspeiseteams (oben rechts)

Durch Betätigen des Output-Buttons wird eine CSV-Datei mit den einzelnen Routen der Teams ausgegeben. Die Konsolenausgabe aus Abbildung 15 zeigt den Verlauf der einzelnen Routen ähnlich der Ausgabe in die Output-Datei. Hierbei entspricht Team 0 bzw. der Ort 0 dem Team 0 bzw. Ort 0 im Viewer usw.

Beispielsweise verläuft die Route von Team 0 von Veranstaltungsort 3 zum Heimatort 0 und endet am Nachspeiseort 8. Team 0 ist demnach ein Hauptspeiseteam. Daraus folgt, dass die Route von Team 0 auf Abbildung 14 unten zu sehen ist. Abbildung 16 zeigt außerdem die Auswahl einzelner Routen.

```
-----Output-----
Writing output file test.csv!

Team: 0 Members: Max Mustermann, Eric Baumgarten
Route: start 3 -> main 0 -> dessert 8
Team: 1 Members: Hans Treu, Johannes Eichner
Route: start 2 -> main 0 -> dessert 1
Team: 2 Members: Lisa Schleußig, Peter Gefreu
Route: start 2 -> main 7 -> dessert 8
Team: 3 Members: Julia Kern, Sabine Unruhe
Route: start 3 -> main 4 -> dessert 1
Team: 4 Members: Kora Meissen, Frank Raubein
Route: start 2 -> main 4 -> dessert 5
Team: 5 Members: Franz Zimmermann, Peter Graumann
Route: start 3 -> main 7 -> dessert 5
Team: 6 Members: Sivia Höffner, Eric Trist
Route: start 6 -> main 0 -> dessert 5
Team: 7 Members: Ilse Meier, Michael Jackson
Route: start 6 -> main 7 -> dessert 1
Team: 8 Members: Uwe Baumann, Gerald Zimmmer
Route: start 6 -> main 4 -> dessert 8
Writing file test.csv successful!
```

Abbildung 15 Ausgabe in der Konsole mit einer Route für jedes Team

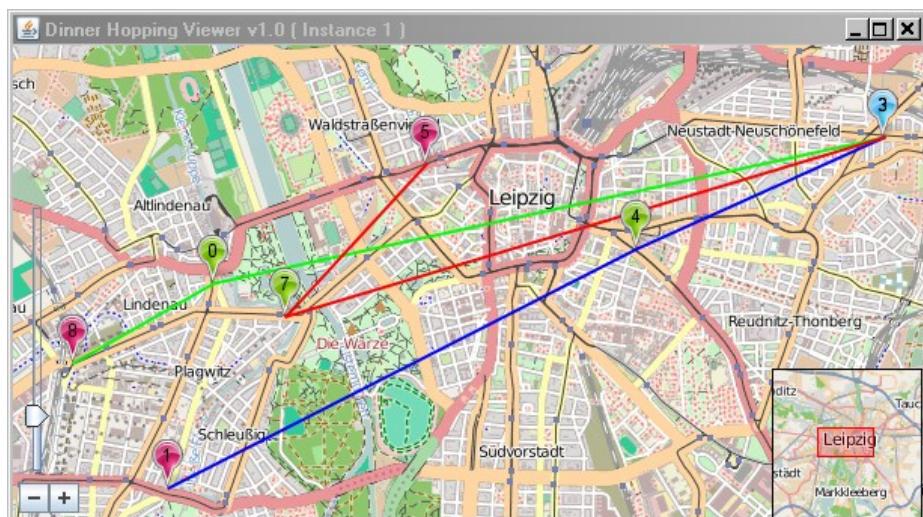


Abbildung 16 Auswahl der Routen für die Teams 0, 5 und 3

5. Diskussion

5.1 Evaluation

5.1.1 Korrektheit des Routenplanungsalgorithmus

Dieser Abschnitt zeigt die Korrektheit des Routenplanungsalgorithmus (siehe Abschnitt 4.1), indem aufgezeigt wird, dass die zu erfüllenden Nebenbedingungen aus Abschnitt 2.1 in jedem Teilschritt erfüllt sind. Im Folgenden bezeichnet n die Anzahl der Teams bzw. Veranstaltungsorte.

Schritt 1

Schritt 1 beschäftigt sich mit der Planung der Routen der Hauptspeiseteams und dem Setzen von Vor-, Haupt- und Nachspeiseorten.

Nebenbedingung (1) wird in diesem Schritt umgesetzt.

Durch die Verwendung der Nearest Neighbor Heuristik und der Smallest Sum Insertion werden kurze Routen für Hauptspeiseteams erwartet. Eine Garantie ist hier nicht gegeben. Es ist demnach nicht sichergestellt das kurze oder optimale Routen entstehen. Durch den mehrmaligen Durchlauf dieses Schrittes und dem Auswerten und Filtern der Ergebnisse sollen möglichst gute Näherungen an eine optimale Lösung erreicht werden. Es sei noch einmal darauf hingewiesen, dass kein Anspruch auf eine optimale Lösung besteht.

Schritt 1.1

Es werden $n/3$ Orte ausgewählt, die provisorisch als Hauptspeiseorte festgelegt werden.

Es existieren drei Arten von Teams: Vor-, Haupt- und Nachspeiseteams. Da die Bedingungen (7) und (4) gelten, müssen immer gleich viele Teams jeder Art vorhanden sein. Daraus folgt, dass bei n Teams jeweils $n/3$ Teams jeder Art vorhanden sein müssen.

Daraus folgt: Die Auswahl von $n/3$ Teams als Hauptspeiseteams verletzt die Nebenbedingungen nicht.

Schritt 1.2

Es werden jedem provisorisch ausgewählten Ort zwei Orte zugeordnet. Diese Tripel von Orten werden später einer Route eines Hauptspeiseteams zugeordnet. Ein Ort wird nur genau einem Hauptspeiseort zugewiesen.

Die Route eines Teams führt immer über drei Orte von einem Vorspeiseort über einem Hauptspeiseort zu einem Nachspeiseort (7).

Daraus folgt: Die Zuteilung von jeweils zwei Orten zu genau einem provisorischen Hauptspeiseort verletzt die Nebenbedingungen nicht.

Schritt 1.3

Dieser Schritt beinhaltet das Verbinden der Tripel von Orten zu einer Route für ein Hauptspeiseteam, sodass die kürzest mögliche Route zwischen den drei Orten entsteht. Der mittlere Ort wird als Hauptspeiseort festgelegt, die beiden anderen zufällig als Vor- bzw. Nachspeiseort.

Die genannte Nebenbedingung aus Schritt 1.2 wird nicht verletzt. Es entsteht eine Route, die (7) erfüllt. Des Weiteren richtet das Hauptspeiseteam seinen Gang am Hauptspeiseort an ((4) und (5)). An den beiden anderen Orten ist es zu Gast (8). Durch die Festlegung von Vorspeise- und Nachspeiseort wird ebenfalls festgelegt, dass die dort ansässigen Teams (Heimatort) Vorspeise- bzw. Nachspeiseteams sind. Diese richten an ihrem Heimatort entsprechend die Vor- oder Nachspeise an. Damit ist ((4) und (5)) für alle Teams erfüllt. Da die Routen der Hauptspeiseteams bei einem Vorspeiseort beginnen und in einem Nachspeiseort enden, ist sichergestellt, dass der Gang des Gastes mit dem Gang des Gastgebers übereinstimmt (3).

Nebenbedingung (2) wird durch die in Schritt 1.3 getroffenen Maßnahmen stets gewährleistet. Durch die beschriebenen Felder für Vor-, Haupt- und Nachspeiseort in jedem Teamobjekt wird sichergestellt, dass jedes Team nur genau einen Vor-, Haupt- und Nachspeiseort durchläuft. Des Weiteren kann stets überprüft werden, welche und wie viele Teams am jeweiligen Ort zu Gast sind. Damit kann (6) stets aufrechterhalten werden.

Schritt 2

Dieser Schritt beschäftigt sich mit dem Festlegen der Routen für die Vorspeiseteams und Nachspeiseteams.

Durch Nutzen der Nearest Neighbor Heuristik und Random Insertion wird (1) behandelt.

Schritt 2.1

In diesem Schritt wird jedem Vorspeiseteam ein Hauptspeiseort zugeordnet. Dies geschieht nach der Nearest Neighbor Heuristik. Die Vorspeiseteams werden dabei zufällig durchlaufen (Random Insertion). Ausgewählte Hauptspeiseorte werden aus der Liste möglicher Folgeorte entfernt.

Jedes Vorspeiseteam muss einen Hauptspeiseort als zweiten Ort seiner Route durchlaufen. Diese Zuordnung wird nun durchgeführt. Damit ist (3), (7) und (8) erfüllt. Dabei wird darauf geachtet, dass ein Vorspeiseteam das jeweilige Hauptspeiseteam nicht bereits gesehen hat, indem die Liste mit bereits gesehenen Teams überprüft wird. Sollte dies der Fall sein, dann wird der jeweils nächste Hauptspeiseort für eine Auswahl geprüft. Damit ist die Nebenbedingung (2) erfüllt. Am Ende dieses Schrittes wurden jedem Vorspeiseort ein Hauptspeiseteam und das Vorspeiseteam selbst zugeordnet. Jedem Hauptspeiseort wurden ein Vorspeiseteam sowie das Hauptspeiseteam selbst zugeteilt. Jedem Nachspeiseteam wurden ein Hauptspeiseteam und das ansässige Nachspeiseteam selbst zugeordnet. Damit ist (6) erfüllt.

Schritt 2.2

In diesem Schritt sollen die Routen der Vorspeiseteams komplettiert werden. Es wird jedem Vorspeiseteam ein Nachspeiseort, ausgehend von dem im vorherigen Schritt gewählten Hauptspeiseort, zugewiesen. Die Auswahl erfolgt wiederum nach der Nearest Neighbor Heuristik.

Ein Vorspeiseteam muss einen Nachspeiseort als dritten Ort seiner Route durchlaufen. Dies wird hier durchgeführt. Damit ist (3), (7) und (8) erfüllt. Jede Zuordnung eines Folgeortes geht mit einer Überprüfung der Liste mit bereits gesehenen Teams einher. Damit ist die Bedingung, (2) keine doppelten Begegnungen, zu jeder Zeit erfüllt. Gibt es keine konsistente Zuordnung, so wird der aktuelle Schritt abgebrochen und in den vorherigen Schritt zurückgekehrt, um diesen neu zu berechnen. Es gibt demnach zu keinem Zeitpunkt eine inkonsistente Aufteilung der Teams. Nach diesem Schritt wurden jedem Nachspeiseort ein Vorspeiseteam, ein Hauptspeiseteam sowie das Nachspeiseteam selbst zugeordnet. Damit ist (6) weiterhin erfüllt. Außerdem ist an jedem Hauptspeise- sowie Vorspeiseort genau ein Platz für ein Gast-Team frei. Es ist demnach möglich, (6) in den Folgeschritten konsistent zu halten.

Die Schritte 2.1 und 2.2 werden für die Nachspeiseteams in analoger Weise durchlaufen. Schritt 2.3 besteht allerdings darin, einen Vorspeiseort für das aktuelle Nachspeiseteam zu finden.

Daraus folgt: Auch hier liegt zu keinem Zeitpunkt eine inkonsistente Zuordnung der Teams vor. Werden die Schritte 1.1 bis 2.4 vollständig durchlaufen, so sind die Nebenbedingungen (2) bis (8) erfüllt. Bei einem nicht vollständigen Durchlauf wurde keine konsistente Routenplanung gefunden. Das Zurückliefern von inkonsistenten Routenplanungen als Ergebnis des Algorithmus ist nicht möglich. Durch die Auswahl der Folgeorte mit der kürzesten Entfernung wird hier zumindest heuristisch versucht, kurze Routen (1) zu erzeugen. Optimale Routen sind bei diesem Ansatz nicht zu erwarten.

5.1.2 Laufzeiten des Routenplanungsalgorithmus

Der Routenplanungsalgorithmus kann durch drei Parameter beeinflusst werden. „changeMainCoursesForRecursion“ (C) gibt an, wie oft die Routen für die Hauptspeiseteams neu berechnet werden sollen. „recursionDepth“ (B) legt die Anzahl der Neuberechnungen von Routen pro Zustand fest. „solutions“ (A) steht für eine maximale Anzahl von zu berechnenden Lösungen pro Konfiguration der Hauptspeiseorte. Daraus folgt, dass sich die maximale Anzahl der zu berechnenden Lösungen aus $A * C$ ergibt. Das Referenzsystem bestand aus einem Intel Atom N280 (1.66 GHz) mit 2 GB Arbeitsspeicher. Für die Laufzeitanalyse wurden jeweils zwei Parameter auf einen fixen Wert gesetzt, während der dritte Parameter in Schritten erhöht wurde. Die genutzte Konfiguration wird in der Form (A, B, C) angegeben.

Abbildung 17 zeigt die Laufzeit in Abhängigkeit von Parameter C. Die gewählte Konfiguration lautet (1, 1000, C).

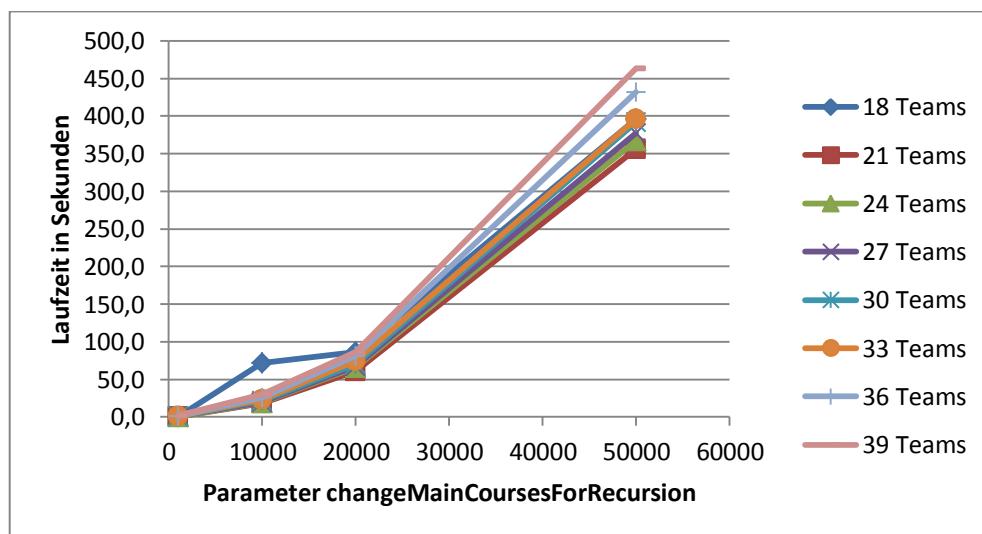


Abbildung 17 Laufzeiten des Routenplanungsalgorithmus in Abhängigkeit von Parameter „changeMainCourseForRecursion“ für unterschiedliche Teamanzahlen. Für C gleich 50000 bei 18 Teams wurden nach 15 min 2000 Lösungen gefunden. Der entsprechende Messpunkt wurde aus Gründen der Übersichtlichkeit nicht eingetragen.

Die Auswahl dieser Anfangskonfiguration erfolgte aus folgender Beobachtung. Die Laufzeiten für Teamzahlen größer 15 liegen mit (1, 1000, 1000) meist im Millisekunden-Bereich, während kleinere Teamzahlen bereits Minuten benötigen (15 Teams = 5 Minuten). Bei einer Konfiguration von (1, 1000, 20000) wurden nach 15 Minuten 1000 Lösungen von maximal 20000 zu erwartenden Lösungen geliefert, sodass die Laufzeit auf mehrere Stunden zu schätzen ist. Mögliche Begründungen und Lösungen dieses Problems werden während der nachfolgenden Betrachtungen gegeben. Um die Entwicklung für größere Teamzahlen besser darstellen zu können, werden die Teamzahlen erst ab 18 Teams betrachtet.

Für C gleich 1000 liegen die berechneten Laufzeiten für alle Teams unter 2 s. Ein Verzehnfachen des Parameters führt bereits zu einem zwanzig- bis dreißigfachen Wert der Laufzeit. Wird der Parameter daraufhin auf 20000 verdoppelt, so verdreifacht sich die Laufzeit, was in etwa dem Wachstum im vorherigen Schritt entspricht. Eine weitere Erhöhung des Parameters auf 50000 zeigt einen fünf bis sechsmal höheren Wert für die Laufzeit. Die Abbildung lässt eine ständige Erhöhung des Wachstums erkennen. Jedoch wäre ein konstantes Wachstum zu vermuten, da ausschließlich die Anzahl der Neuberechnungen von Routen für Hauptspeiseteams steigt. Die darauf folgende Rekursion ist in allen Fällen die gleiche Prozedur. Allerdings dauert es mit fortschreitender Neuberechnung länger, eine neue Lösung für die jeweilig aktuelle Konfiguration der Hauptspeiseteams zu finden. Für C gleich 50000 und A gleich 1 sind bereits 50000 Lösungen zu bestimmen. Eine neue Lösung bedeutet hierbei, dass diese noch nicht vorhanden ist und die Nebenbedingungen erfüllt. Dies ist eine mögliche Begründung dieses Wachstums.

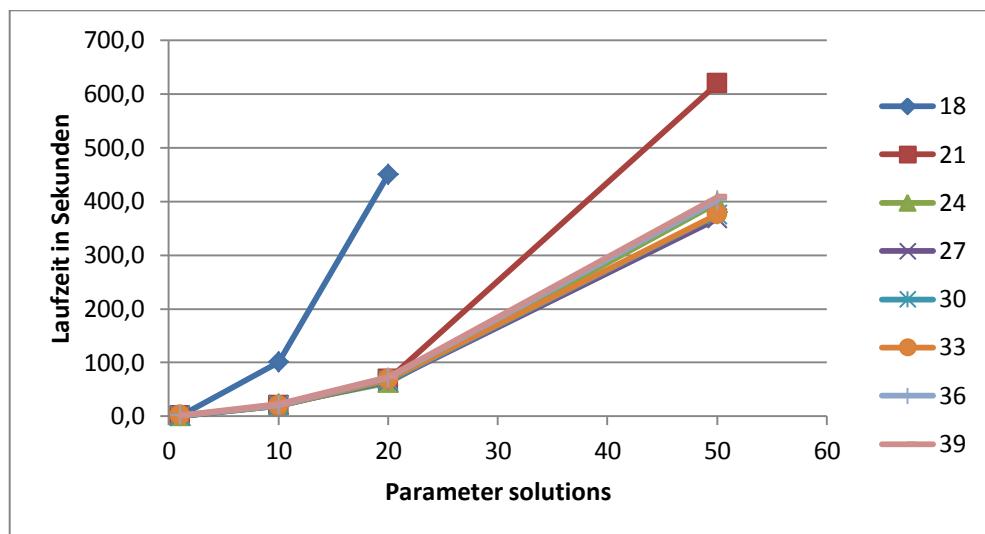


Abbildung 18 Laufzeit des Routenplanungsalgorithmus in Abhängigkeit von Parameter „solutions“ für unterschiedliche Teamanzahlen. Für C gleich 50 bei 18 Teams wurden nach 30 min 30000 Lösungen gefunden. Der entstehende Messpunkt wurde aus Gründen der Übersichtlichkeit nicht eingetragen.

Als Nächstes wird der Parameter A untersucht (Konfiguration (A, 1000,1000)). Abbildung 18 zeigt, dass es keine bedeutenden Unterschiede der Laufzeiten für Teamanzahlen von 24 bis 39 Teams gibt. Das Wachstum entspricht hier in etwa dem in der vorherigen Abbildung beschriebenen. Für 21 Teams allerdings ist für Werte von $A > 20$ ein deutlich höheres Wachstum zu beobachten. Für 18 Teams und A gleich 1 ist die Laufzeit mit denen der restlichen Teamzahlen einheitlich. Daraufhin ergibt ein Verzehnfachen von A bereits eine hundertsfache Laufzeit, während sich für die restlichen Teamzahlen maximal eine vierzigfache Laufzeit erkennen lässt.

Das beschriebene Verhalten kann folgendermaßen erklärt werden. Je geringer die Anzahl der Teams ist, umso weniger Möglichkeiten existieren die Teams in einer Weise anzugeordnen, sodass die Nebenbedingungen (1) bis (8) erfüllt sind. Gibt es weniger Möglichkeiten, so muss, jedes Mal, wenn keine konsistente Zuordnung von Orten zu Gängen gefunden wird, ein Backtracking durchgeführt werden. Die Anzahl dieser Schritte steigt damit bei kleineren Teamzahlen bedeutend an. Dieses Verhalten kann abgefangen werden, indem C für kleinere Teamzahlen verkleinert wird oder bereits nach einer geringeren Anzahl von Lösungen zur nächsten Konfiguration der Hauptspeiseteams gewechselt wird (A kleiner wählen).

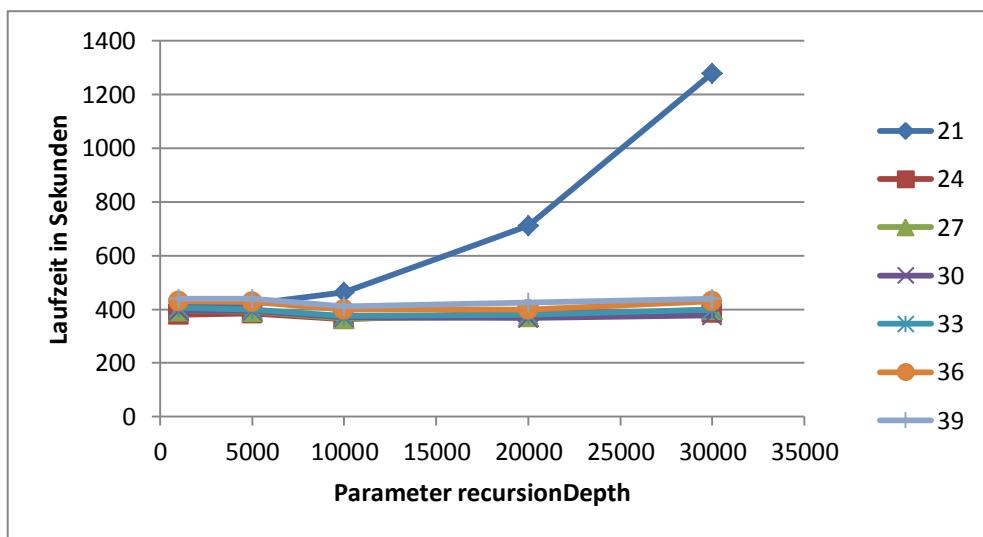


Abbildung 19 Laufzeit des Routenplanungsalgorithmus in Abhängigkeit von Parameter „recursionDepth“ für unterschiedliche Teamanzahlen

Die Untersuchung von Parameter B wurde mit der Konfiguration (10, B, 5000) durchgeführt (siehe Abbildung 19). Zunächst sollen die Laufzeiten für Teamzahlen größer als 21 beschrieben werden. Auffällig ist hierbei, dass es bei einer Verdopplung von B von 5000 auf 10000 zu einer Verringerung der Laufzeit um etwa 25 bis 30 s kommt. Anschließend steigt die Laufzeit bei Erhöhen von B wieder an.

Hier liegt die Ursache vermutlich darin, dass bei einer höheren Neuberechnung von Routen pro Zustand im Intervall $5000 < B < 10000$ vermehrt neue Lösungen gefunden werden. Dies führt zu einem schnelleren Wechsel zur nächsten Konfiguration der Hauptspeiseteams und somit zum schnelleren Terminieren des Algorithmus. Für kleinere Teamzahlen ergab sich ab Werten von $B > 5000$ ein deutliches Wachstum der Laufzeit. Dies ist, wie bei Parameter A beschrieben, auf die geringere Anzahl von Lösungen für kleinere Teamzahlen zurückzuführen. Dabei wird bei steigenden Werten von B immer mehr Zeit verwendet, um in einer Konfiguration der Hauptspeiseteams nach neuen Lösungen zu suchen. Angenommen, die aktuell betrachtete Konfiguration bietet nur wenige Lösungen, dann wird der Algorithmus in

diesem „Minimum an Lösungen“ so lange weitersuchen, bis alle Rekursionsschritte pro Zustand (B) durchlaufen sind. Dieses Verhalten kostet enorm viel Zeit. Durch Herabsetzen von B für kleinere Teamzahlen kann die Laufzeit hier deutlich verringert werden.

5.1.3 Berechnete Distanzen des Routenplanungsalgorithmus

In diesem Abschnitt soll auf die Qualität der Lösungen in Abhängigkeit der in Abschnitt 5.1.2 beschriebenen Parameter eingegangen werden. Die Qualität wird hierbei an der berechneten Gesamtdistanz, also der Summe der zurückgelegten Distanzen aller Teams, gemessen.

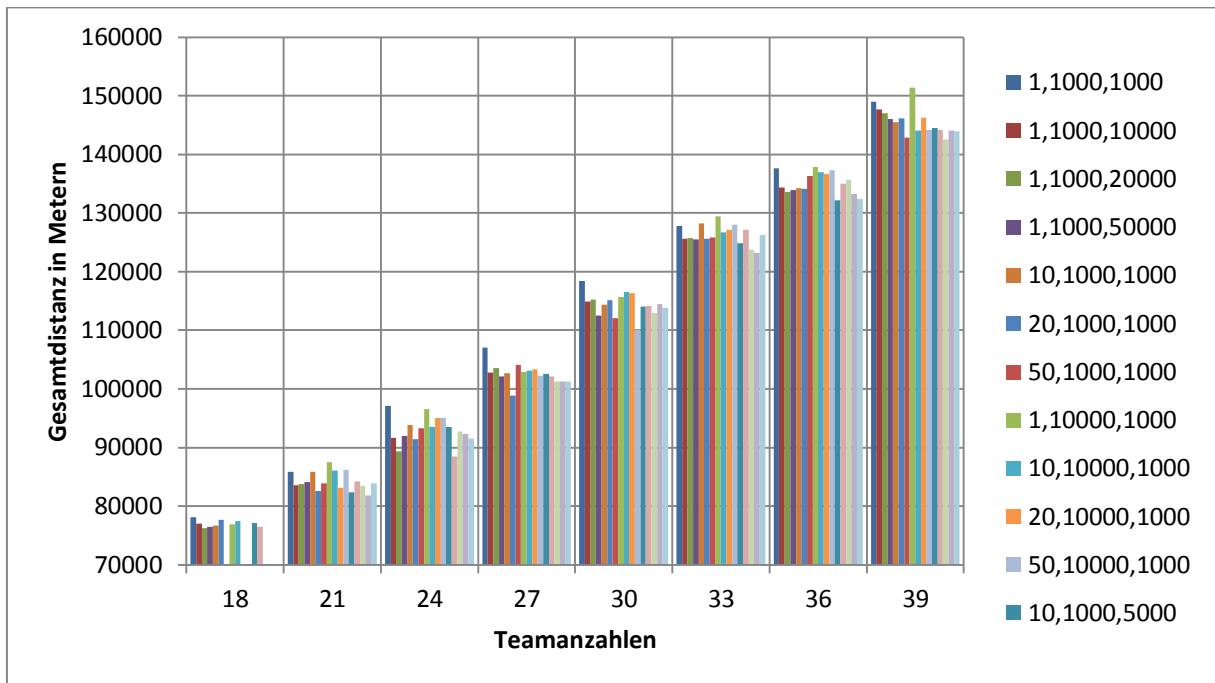


Abbildung 20 Gesamtdistanz in Abhängigkeit der Teamanzahlen für unterschiedliche Konfigurationen der Parameter. Konfigurationen, die für die jeweilige Teamanzahl keine Distanzen in akzeptabler Zeit lieferten, wurden leer gelassen.

Abbildung 20 zeigt die Gesamtdistanz in Abhängigkeit von der Teamanzahl für unterschiedliche Konfigurationen der Parameter. Die Konfigurationen entsprechen den bereits in Abschnitt 5.1.2 verwendeten.

Es ist zu bemerken, dass für kleine Werte von A, B und C, siehe Konfiguration (1, 1000, 1000), für alle Teamzahlen höhere Distanzen entstehen als bei höheren Werten. Die ersten vier Konfigurationen, welche Parameter C erhöhen, zeigen vor allem bei 39 Teams eine deutliche Abstufung. Höhere Werte für C berechnen demnach meist kürzere Distanzen. C wirkt sich unmittelbar auf die Länge der Distanzen aus. Für Teamzahlen von 21 bis 27 Teams bringt ein Wert von B gleich 20 die kürzesten Ergebnisse. Für größere Teamzahlen zeichnet sich eine Verkürzung der Distanzen bei höheren Werten für B ab. Neben einzelnen Minima, z. B. bei 18

Teams mit Konfiguration (1, 1000, 20000) oder bei 27 Teams mit (20, 1000, 1000) berechneten vor allem die fünf jeweils zuletzt eingezeichneten Konfigurationen relativ niedrige Distanzen. Bei A gleich 10 und C gleich 5000 wurde hier B erhöht. Für 33, 36 und 39 Teams wurden jeweils Minima erzielt. Für größere Werte von B zeichnen sich bei größeren Teamzahlen ab 36 kürzere Gesamtdistanzen ab.

Aufgrund der Beobachtungen zu Parameter C ist zu vermuten, dass eine Erhöhung dieses Parameters, für höhere Teamzahlen als den hier untersuchten, zu einer höheren Qualität der Lösung beiträgt. Dies ergibt sich aus der höheren Anzahl von möglichen Konfigurationen von Hauptspeiseteams bei höheren Teamzahlen. Bei den Parametern A und B ist nur zu erahnen, dass eine weitere Erhöhung auch zu einer Verkürzung der Routen beiträgt. Dies muss entsprechend weiterführend untersucht werden.

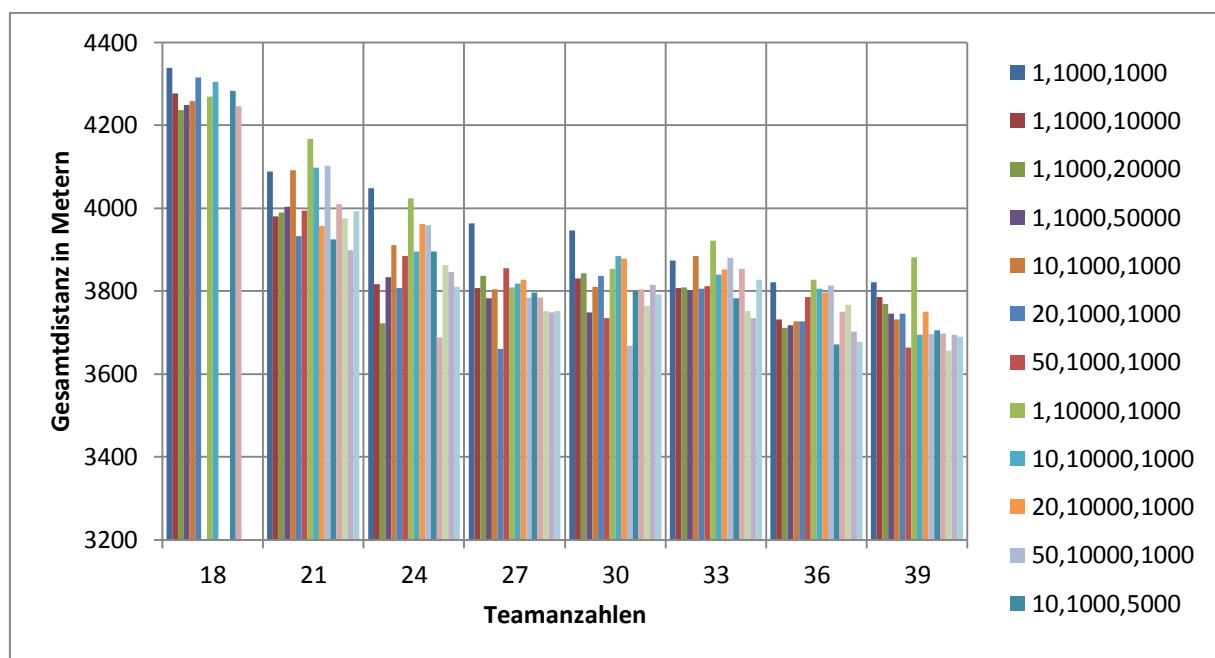


Abbildung 21 Durchschnittliche Distanz eines Teams in Abhängigkeit der Teamanzahlen für unterschiedliche Konfigurationen. Konfigurationen, die für die jeweilige Teamanzahl keine Distanzen in akzeptabler Zeit lieferten, wurden leer gelassen.

In Abbildung 21 wurden die durchschnittlichen Distanzen eines Teams für unterschiedliche Teamzahlen bei unterschiedlichen Konfigurationen dargestellt. Um die durchschnittliche Distanz eines Teams zu erhalten, wurden die Gesamtdistanzen durch die jeweilige Teamanzahl geteilt. Es ist zu sehen, dass die Distanzen für größere Teamzahlen geringer werden. Von 18 bis 24 Teams ist eine deutliche Abstufung zu erkennen. Abbildung 22 zeigt das Verhalten für kleinere Teamzahlen, hier in einer anderen Abbildung dargestellt, da andere Konfigurationen zugrunde gelegt wurden. Das beschriebene Verhalten ist ebenfalls sichtbar. Eine Begründung könnte wiederum die zunehmende Anzahl von Lösungen bei steigenden Teamzahlen liefern.

Damit wären ebenfalls kürzere Distanzen pro Team möglich. Für 27 bis 39 Teams ist nur ein leichtes Abnehmen zu verzeichnen, sodass für höhere Teamzahlen keine deutliche Abnahme, als die bei 39 Teams zu sehenden Distanzen, zu erwarten ist.

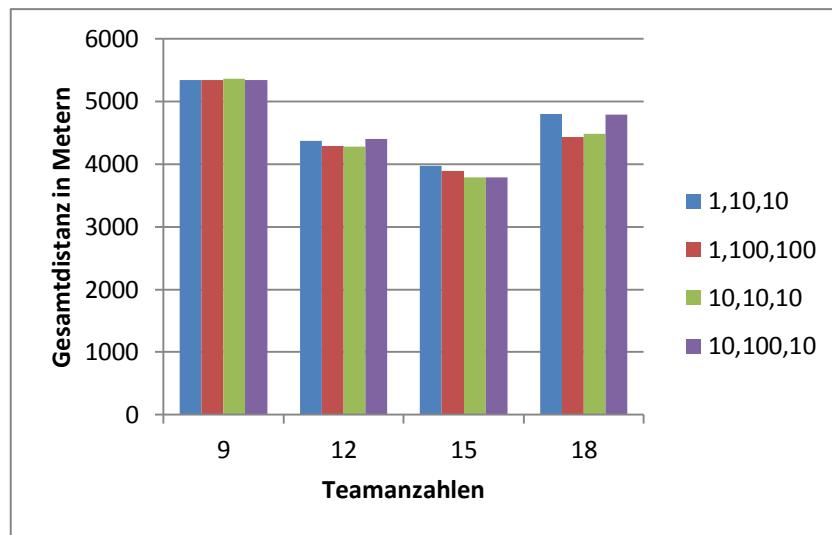


Abbildung 22 Durchschnittliche Distanz eines Teams in Abhängigkeit der Teamanzahlen für unterschiedliche Konfigurationen der Parameter. Die Parameter wurden, für die hier betrachteten kleineren Teamanzahlen, angepasst.

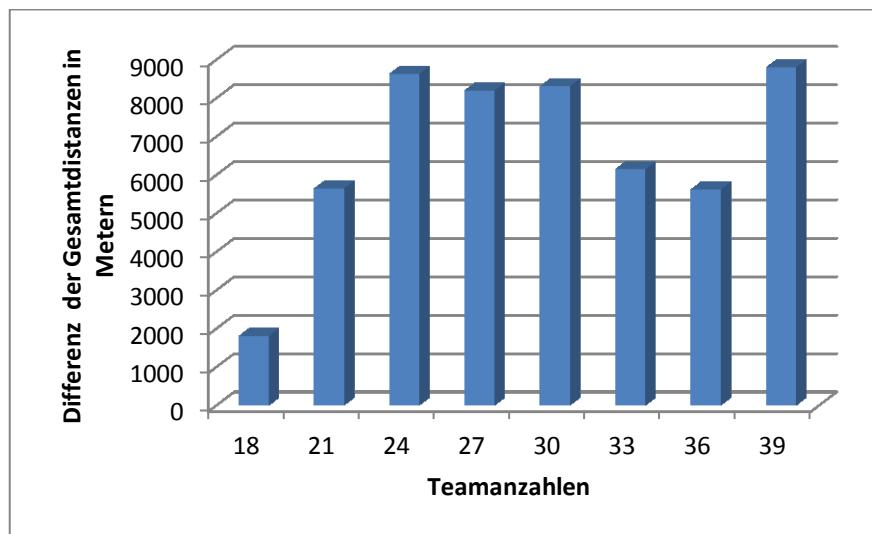


Abbildung 23 Differenz zwischen größter und kleinster ermittelter Gesamtdistanz pro Team für die oben erwähnten Konfigurationen.

Um die Differenz zwischen den berechneten Gesamtdistanzen pro Teamanzahl einschätzen zu können, wurde in Abbildung 23 die Varianz der Gesamtdistanzen abgetragen. Bei kleineren Teamanzahlen liegen die berechneten Distanzen näher beieinander. Mit steigender Teamanzahl ist eine Erhöhung der Varianz zu erkennen. Abbildung 24 zeigt dieses Verhalten auch für kleinere Teamanzahlen. Damit könnte hier eine mehrmalige Ausführung der Routenplanung von Vorteil sein, um zufällig entstandene hohe Werte erkennen und ausschließen zu können.

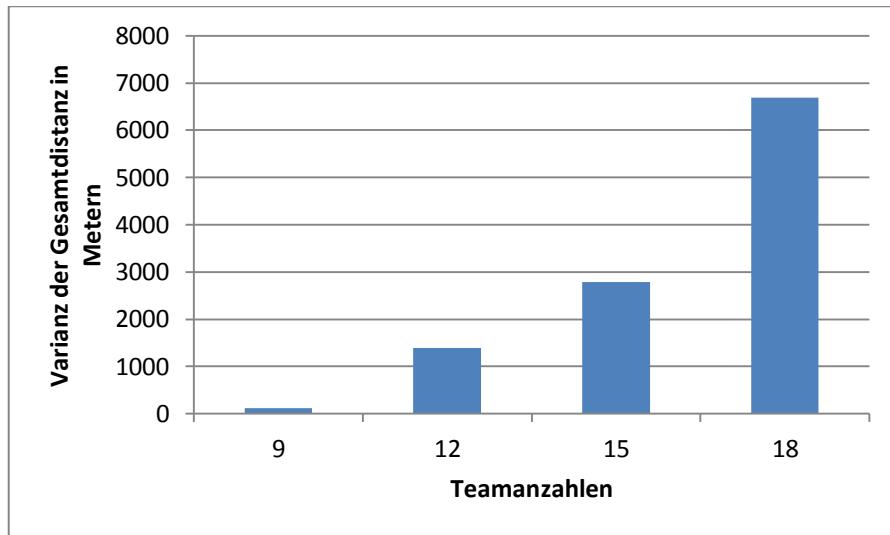


Abbildung 24 Varianz in den bestimmten Gesamtdistanzen für kleinere Teamanzahlen. Es liegen die in Abbildung 22 definierten Konfigurationen zugrunde.

5.2 Kritik

5.2.1 Annahmen zum Routenplanungsalgorithmus

Der Algorithmus zur Routenberechnung wurde in Abschnitt 4.1 beschrieben. Es ergeben sich einige Diskussionsansätze.

Zunächst geht der Routenplanungsalgorithmus davon aus, dass anders als in der formalen Definition des DHP in Abschnitt 2.3.1 beschrieben, gilt

$$O = \{ o \mid o \text{ ist ein Ort} \wedge \forall o_1, o_2 \in O: o_1 \neq o_2 \}, \quad (5.1)$$

$$|T| = |O| \quad (5.2)$$

und dass stets eine bijektive Abbildung, siehe (5.3), existiert.

$$f: T \rightarrow O \quad (5.3)$$

Demnach können zwei Teams nicht am selben Ort beheimatet sein. Es ist natürlich dennoch möglich, für zwei Teams den gleichen Ort anzugeben, jedoch kann es in diesem Fall passieren, dass die beiden Teams zur selben Zeit am selben Ort einen Gang anrichten. Dieses Problem wird im Routenplanungsalgorithmus nicht behandelt.

Des Weiteren ist zu bemerken, dass der Algorithmus einige Möglichkeiten der Erzeugung von Routen nicht betrachtet. Dies sind die folgenden Fälle:

- Zwei Hauptspeiseteams starten von demselben Vorspeiseort
- Zwei Hauptspeiseteam enden in demselben Nachspeiseort

- Zwei Nachspeiseteams starten von demselben Vorspeiseort
- Zwei Vorspeiseteams enden in demselben Nachspeiseort
- Zwei Nachspeiseteams gehen zum gleichen Hauptspeiseort
- Zwei Vorspeiseteams gehen zum gleichen Hauptspeiseort

Wird beispielsweise für ein Vorspeiseteam v_1 ein Hauptspeiseort h ausgewählt, so steht h für das danach betrachtete Vorspeiseteam v_2 nicht mehr zur Verfügung. Dieses Problem kann behoben werden, indem h nicht sofort aus der Liste möglicher Folgeorte gelöscht wird und im Beispiel für weitere Vorspeiseorte zugänglich bleibt. Hier muss allerdings darauf geachtet werden, dass es maximal zwei Vorspeiseteams geben kann, die bei h zu Gast sind, da maximal drei Teams an einem Ort einen Gang einnehmen. Zudem ist nicht klar, ob auf diese Weise insgesamt kürzere Routen entstehen.

5.2.2 Bestimmung der Hauptspeiseorte

Des Weiteren bringt der Ansatz zur Bestimmung der Hauptspeiseorte einige Nachteile mit sich. Die Verteilung der Veranstaltungsorte hat einen großen Einfluss auf die entstehenden Längen der Routen. Sind die Veranstaltungsorte beispielsweise in einzelnen Clustern angeordnet, so wäre es sinnvoll, die Hauptspeiseteams, verteilt auf die einzelnen Cluster, zuzuweisen. Daraufhin würden die Routen dieser Teams hauptsächlich innerhalb eines Clusters verlaufen. In einem solchen Fall wären diese Routen unter Umständen die Kürzeren. Der hier beschriebene Algorithmus nimmt darauf allerdings keine Rücksicht und ordnet Teams im Zentrum aller Veranstaltungsorte an. Dieses Verhalten wird aufgrund der Smallest Sum Insertion hervorgerufen. Es sei allerdings darauf hingewiesen, dass die Zuteilung in Clustern nicht in jedem Fall optimal wäre. Damit Routen ausschließlich innerhalb eines Clusters zugewiesen werden können, müsste die Anzahl der Teams innerhalb eines Clusters mindestens 9 betragen und durch drei teilbar sein. Ist dies nicht der Fall, so müssten die Routen einzelner Teams zusätzlich über andere Cluster verlaufen, wobei wiederum sehr weite Distanzen zu überbrücken wären.

5.3 Fazit

Nach dem Fertigstellen der prototypischen Softwarelösung werden im Folgenden die wichtigsten Anforderungen noch einmal genannt und erläutert, ob diese erfolgreich umgesetzt werden konnten. Muss-Anforderungen sind mit „F“ und Kann-Anforderungen mit „FW“ gekennzeichnet.

Alle Produktfunktionen aus den Muss-Anforderungen wurden erfolgreich implementiert. Dies ist vor allem am Aufbau des Systems durch die einzelnen in Abschnitt 3.1 beschriebenen Komponenten zu erkennen. Die Import/Export-Komponente erfüllt die Funktionen /F10/, /FW70/, /F120/ sowie /F130/ vom Dateiimport über die Anzeige der importierten und exportierten Daten bis zum Export der berechneten Routenplanungen. Der Geocodierer ist für die Funktionen /F20/ bis /F60/ von der Übergabe der Adressen an einen Dienst bis zur Evaluierung der Adressen durch den Nutzer verantwortlich. Die Entfernungsbestimmung (/F110/) wird über den Distanzkalkulator realisiert. Der Routenberechner übernimmt die Bestimmung der Routen sowie die Aufbereitung dieser durch eine Filterung der möglichen Routenplanungen. Damit ist Funktion /F110/ erfüllt. Schließlich können die Routen auf verschiedene Weisen visualisiert werden, was sowohl /F80/ als auch /FW90/ abdeckt. Die Kann-Anforderung „Unterstützung des E-Mail-Versandes“ (/FW140/) wurde nicht umgesetzt. Die Produktleistungen wurden ebenfalls zufriedenstellend verwirklicht. Hier sei zum Beispiel die Einhaltung aller Nebenbedingungen bei der Routenplanung genannt. Ein Manko ist, dass die benötigte Zeit für eine Route bei der Planung nicht berücksichtigt wird, wie in /L40/ beschrieben. Des Weiteren werden einige Möglichkeiten der Erzeugung von Routen nicht betrachtet.

Die Qualitätsanforderungen wurden nach Tests mit dem Auftraggeber als eingehalten angesehen. Der Komponentenaufbau des Systems trägt zur leichten Änderbarkeit bei. Die Benutzeroberfläche ist gut verständlich und leicht erlernbar. Negativ fiel hier die Geocodierung auf, welche nicht vollständig in die Benutzeroberfläche integriert wurde. Hier müssen Abstriche bei der Bedienbarkeit verzeichnet werden.

Bei der Modellierung der Benutzeroberfläche wurde großen Wert auf eine intuitive Bedienbarkeit sowie Übersichtlichkeit gelegt. Durch die Bereitstellung mehrerer Dienste für die Komponenten Geocodierung und Distanzkalkulator ist die Funktionstüchtigkeit des Systems auch auf längere Zeit hin gewährleistet. Die Visualisierungskomponente kann ebenfalls andere Kartendienste zur Darstellung verwenden. Eine Änderung erfordert jedoch einen Eingriff in den Quellcode.

Zusammenfassend lässt sich sagen, dass das System den Anforderungen genügt. Vor allem der Routenplanungsalgorithmus sowie die Benutzeroberfläche lassen mehrere Schwächen des Systems erkennen. In Anbetracht der kurzen Entwicklungszeit konnten jedoch verwertbare Ergebnisse erzielt werden, sodass das System für den realen Einsatz geeignet ist.

5.4 Ausblick

In Form eines Prototyps realisiert, zeigt die Software nur eine Möglichkeit auf, ein derartiges Planungssystem mit seinen fünf Komponenten zu realisieren. Es lässt sich einiges Potenzial für Verbesserungen erkennen. Außerdem sind Ideen für nützliche zusätzliche Funktionalitäten entstanden, die aufgrund der begrenzten Entwicklungszeit nicht umgesetzt werden konnten. Im Folgenden sollen einige Verbesserungsvorschläge und Ideen benannt werden.

Der Hauptansatzpunkt für Verbesserungen ist der Routenplanungsalgorithmus. Hier wurde ein mögliches Konzept der Konstruktion von Routen vorgestellt. Wie bereits im Abschnitt 5.2 „Kritik“ bemerkt, könnte für eine Auswahl von Hauptspeiseorten und zur Verkürzung der Routen auf Cluster-Algorithmen zurückgegriffen werden. Außerdem besteht die Möglichkeit, andere Heuristiken als die hier beschriebenen zu verwenden. Hier sei beispielsweise die Farthest Insertion genannt, welche in [4, S. 82] beschrieben ist. Sie lieferte für TSP-Touren im Durchschnitt die besten Ergebnisse. Gleiches gilt für Schritt 2 des Algorithmus. Außerdem wurden Annahmen getroffen, die einige Möglichkeiten von Routenplanungen ausschließen. Diese sollten in zukünftigen Entwicklungen mit betrachtet werden. Eine verbesserte Version des Algorithmus wurde bereits erstellt, konnte allerdings aus Zeitgründen nicht mehr in dieser Arbeit vorgestellt werden.

Des Weiteren ist die Fortschrittsanzeige des Programms in der Konsole nicht als komfortabel zu bezeichnen. Hier bietet sich eine Anzeige in der grafischen Benutzeroberfläche an. Auch die Geocodierung der Orte könnte auf diese Weise verbessert werden. Die Auswahl eines Ortes aus mehreren möglichen Geocodierungen muss im Moment in der Konsole erfolgen.

Aus den Kann-Anforderungen des Pflichtenheftes sind einige Funktionalitäten zu nennen, die im Rahmen dieser Arbeit nicht umgesetzt werden konnten. Dies betrifft die Anzeige der Teilnehmerdaten (/FW70/). Diese werden ebenfalls ausschließlich in der Konsole angezeigt. Weiterhin ist die Unterstützung beim E-Mail Versand der fertigen Routenplanungen an die einzelnen Teams zu nennen. Hierbei sollen die E-Mails für den Versand mit Routenabfolge und Zusatzinformationen vorbereitet werden (/FW140/).

Aus Gesprächen mit dem Auftraggeber während der Entwicklung stellte sich heraus, dass die Einbeziehung des Nahverkehrs, in Leipzig wäre dies der „Leipziger Verkehrsbetriebe (LVB)“,

in die Distanzbestimmung eine nützliche Funktionalität wäre. Viele Studenten wählen dieses Transportmittel. Eine Unterstützung des Nahverkehrsnetzes durch einen Dienst konnte in der Recherche allerdings nicht festgestellt werden, sodass hier keine Informationen für eine mögliche Realisierung gegeben werden können.

Außerdem wäre eine genaue Angabe der Routenabschnitte und Fahrhinweise für eine Route von einem Veranstaltungsort zum folgenden denkbar. Dies ist ohne Probleme mit den aktuell implementierten Diensten möglich. Für den Osm2po und CloudMade Routing Service sind entsprechende Methoden in den genutzten Java-Bibliotheken vorhanden.

Kurzzusammenfassung

Die WILMA („Willkommens-Initiative für in Leipzig mitstudierende Ausländer“) veranstaltet in regelmäßigen Abständen ein „Dinner-Hopping“. Dabei sind Teams gegeben, die an bestimmten Orten beheimatet sind. Jedes Team richtet entweder eine Vor-, Haupt oder Nachspeise an. Ziel ist es, jedem Team eine Route, bestehend aus drei Orten zuzuteilen, sodass jedes Team ein komplettes 3-Gänge-Menü einnimmt. Dabei müssen bestimmte Nebenbedingungen eingehalten werden.

Ziel dieser Arbeit war die Entwicklung einer prototypischen Softwarelösung, die den Nutzer bei der Planung einer solchen Veranstaltung unterstützt. Die Software liest hierfür Teilnehmerdaten ein und geocodiert die enthaltenen Adressen der Teams. Auf Grundlage der Entfernung zwischen einzelnen Veranstaltungsorten werden möglichst kurze Routen für jedes Team bestimmt. Nach dem Errechnen mehrerer möglicher Planungen können diese verglichen und Routen auf vielfältige Weise visualisiert werden.

Die Entwicklung umfasste zunächst das Aufdecken der theoretischen Grundlagen zur Lösung des Problems. Hierbei wurde das Travelling Salesman Problem mit seinen Lösungsmöglichkeiten auf die vorliegende Problematik übertragen. Es folgte die Auswahl möglicher Dienste für Geocodierung, Entfernungsbestimmung und Visualisierung. Schließlich wurde die Modellierung sowie Implementierung des Prototyps mit Entwicklung eines geeigneten Algorithmus für die Routenplanung durchgeführt und die erstellte Lösung evaluiert.

Literaturverzeichnis

- [1] H. Balzert, Lehrbuch der Software-Technik, 2. Auflage, Bd. 1. Software-Entwicklung, Heidelberg, Berlin: Spektrum Akademischer Verlag, 2000, 3-8274-0480-0.
- [2] U. Schöning, Algorithmen - kurz gefasst / Uwe Schöning, Heidelberg, Berlin: Spektrum, Akad. Verl., 1997, 3-8274-0232-8.
- [3] L. Suhl und T. Mellouli, Optimierungssysteme. Modelle, Verfahren, Software, Anwendungen, Berlin, Heidelberg: Springer-Verlag, 2009, 978-3-642-01579-3.
- [4] G. Reinelt, The Travelling Salesman. Computational Solutions for TSP Applications, Berlin, Heidelberg: Springer-Verlag, 1994, 3-540-58334-3.
- [5] W. Schmitting, „Das Traveling-Salesman-Problem: Anwendungen und heuristische Nutzung von Voronoi-/Delaunay-Strukturen zur Lösung euklidischer, zweidimensionaler Traveling-Salesman-Probleme/ Walter Schmitting“, Universität- und Landesbibliothek Düsseldorf, 2000, 3-00-004089-7. [Online]. Available: <http://docserv.uni-duesseldorf.de/servlets/DocumentServlet?id=2314>. [Zugriff am 02.06.2012].
- [6] T. Bektas, „The multiple traveling salesman problem: an overview of formulations and solution procedures“, *Omega*, Bd. 34, S. 209-219, 2006.
- [7] A. Hamacher und C. Moll, „The Euklidian Travelling Salesman Selection Problem“, Universität Köln, Köln, 1995.
- [8] C. Ullenboom, Java ist auch eine Insel, 10. Aktualisierte Auflage, Bonn: Galileo Press GmbH, 2012, 978-3-8362-1802-3.
- [9] „About CloudMade“, CloudMade, 2008-2012. [Online]. Available: <http://cloudmade.com/about>. [Zugriff am 17.06.2012].
- [10] „Geocoding and Geosearch“, CloudMade, 2008-2012. [Online]. Available: <http://developers.cloudmade.com/projects/show/geocoding-http-api>. [Zugriff am 17.06.2012].
- [11] „Microsoft Bing Maps Platform API's Terms Of Use“, Microsoft, 09 2012. [Online]. Available: <http://www.microsoft.com/maps/product/terms.html>. [Zugriff am 28.09.2012].
- [12] „Google Maps/Google Earth APIs Terms of Service“, Google, 20.04.2012. [Online]. Available: https://developers.google.com/maps/terms#section_10_12. [Zugriff am 28.09.2012].

- [13] B. Lahres und G. Rayman, Objektorientierte Programmierung, 2. aktualisierte und erweiterte Auflage, Galileo Computing, 2009, 978-3-8362-1401-8, S. 656.

Anlagenverzeichnis

A. CD

Im Wurzelverzeichnis der beiliegenden CD gibt es fünf Ordner mit den folgenden Ressourcen:

- Inhaltsübersicht.pdf
- Abbildungen (Abbildungen der Bachelorarbeit)
- Bachelorarbeit
 - Integration_von_Geodaten_in_ein_Planungssystem.pdf
- Dokumente des Entwicklungsprozesses
 - Risikoanalyse.pdf
 - Vorgehensbeschreibung.pdf
 - Glossar.pdf
 - Recherchebericht.pdf
 - Lastenheft.pdf
 - Pflichtenheft.pdf
 - Dokumentationskonzept.pdf
- Literatur (Online-Quellen)
 - Das Travelling-Salesman-Problem - Anwendungen und heuristische Nutzung von Voronoi- bzw. Delauney-Strukturenpdf [5]
 - The multiple traveling salesman problem - an overview of formulations and solution procedures.pdf [6]
 - The Euclidian Travelling Salesman Selection Problem.pdf [7]
 - About CloudMade.htm [9]
 - Geocoding and Geosearch.htm [10]
 - MICROSOFT® BING™ MAPS PLATFORM API's TERMS OF USE.htm [11]
 - Google Maps-Google Earth APIs Terms of Service - Google Maps API — Google Developers.htm [12]
- Prototyp
 - Dinner_Hopping_Planner_v.1.0.jar (Ausführbarer Prototyp)
 - doc (JavaDoc)
 - src (Quellcode)
 - ...
 - Benutzer_Leitfaden_v.1.0.pdf
 - addr.csv (Beispieldatensatz)
 - sn (Osm2po-Graph-Daten)
 - readme.txt

Erklärung

"Ich versichere, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zu widerhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann."

Leipzig, 9. November 2012

Björn Buchwald