UNIVERSITÄT OSNABRÜCK

Bachelorthesis

Optimierung von Straßennetzdaten mit Hilfe von Geodaten aus OpenStreetMap

eingereicht von
Phillipp Bertram - 925089

am
14. Dezember 2009

Fachbereich Informatik
Research Group Media Computer Science
Prof. Dr. Oliver Vornberger
Jun. Prof Dr.-Ing. Elke Pulvermüller

Inhaltsverzeichnis

1. Einleitung

Mobilität ist ein wichtiger Bestandteil unserer Gesellschaft. Nur durch umfassende Verkehrsbeobachtungen und der Erfassung von Straßennetzdaten ist bei wachsendem Verkehrsaufkommen optimale Fortbewegung möglich. Ein Unternehmen, welches sich mit dieser Aufgabe beschäftigt, ist *Die Gesellschaft für Verkehrsdaten* (DDG), ein führender Content-Provider für Dienstanbieter der Verkehrstelematik[1]. Sie erfasst europaweit verkehrstelematisch interessante Informationen und kann dadurch für den Verkehr bedeutende Prognosen (u.a. Staumeldungen) erstellen.

Doch auch kommerzielle Unternehmen wie *Google* haben es sich zur Aufgabe gemacht, derartige Daten zu sammeln und – für private Zwecke jedoch meist in Verbindung mit hohen Kosten – zur Verfügung zu stellen. Diese Daten sind in diesem Fall allerdings nicht wie bei der DDG auf den Verkehr zugeschnitten, sondern bilden in der Regel nur Sammlungen geographischer Objekte zur Konstruktion und Darstellung von Karten.

Seit der Gründung von *OpenStreetMap*, einem offenen Projekt zur Sammlung von frei verfügbaren Geodaten, können diese Daten nun auch kostenlos erworben und für private Zwecke genutzt werden. Dadurch besteht nun die Möglichkeit, diese frei erwerblichen Daten zur Optimierung der Straßennetzdaten der DDG-Daten derartig zu nutzen, dass durch die wesentlich höher aufgelösten Geodaten der OpenStreetMap die ungenauen Straßenverläufe der DDG verfeinert werden. Diese Datenoptimierung stellt den Kern der vorliegenden Arbeit dar.

In den folgenden Abschnitten erfolgt zunächst eine kurze Darstellung der DDG sowie des OpenStreetMap-Projekts, bevor anschließend die Zusammenführung und Nutzung der Geodaten von OpenStreetMap zur Optimierung der Straßennetzdaten der DDG im Ziel der Arbeit dargelegt wird.

1.1. DDG, Die Gesellschaft für Verkehrsdaten mbH

Gut zwei Jahre vor der Gründung der DDG im Mai 1997 verfolgten die Telekommunikationsunternehmen T-Mobile und Mannesmann Autocom unabhängig voneinander ein einheitliches Ziel: „objektive Verkehrsdatenerfassung mit technischen Systemen, um neue Mobilitätsdienste zu ermöglichen“ [10].

Innerhalb dieser zwei Jahre entstand neben dem stationären Erfassungs-System (*SES*), welches mit Sensoren im Autobahnnetz Verkehrsdaten elektronisch sammelt und somit

[1] Verkehrstelematik versteht sich als Anwendung moderner Kommunikations-, Informations- und Leittechnologien im Verkehr

etwa 70% des Autobahnnetzes und rund 90% aller Verkehrsstörungen erfasst. Auch das *Floating Car Data*-Verfahren[2] (*FCD*), das nunmehr der Vorreiter des heutigen DDG-Verfahrens ist, entstand in dieser Zeit. Hierbei sorgen mit Sensoren ausgestattete Autos für mobile, umfassende und aktuelle Erkenntnisse aus dem Geschehen im Straßenverkehr.

Derzeit generiert die DDG als einziges Unternehmen objektive Daten für das ganze Autobahnnetz und verfügt zudem über große Erfahrungen mit diesem Verfahren der Verkehrstelematik. Damit ermöglicht die DDG den Telematik-Dienstanbietern, allen Autofahrern aktuelle Verkehrsinformationen zuverlässig zur Verfügung zu stellen.

1.2. Das OpenStreetMap-Projekt

OpenStreetMap (OSM) wurde 2004 von Steve Coast in Großbritannien mit dem Ziel gegründet, eine freie Weltkarte aus *user*generiertem Inhalt zu erschaffen. Anders als bei kommerziellen Anbietern (z.B. *Google*) wird keine rechtliche Bindung an die Benutzung der Daten geknüpft, sodass durch die Zusammenarbeit der Projektmitglieder eine freie Geodatenbank[3] entwickelt wird, die weltweit allen Menschen lizenzkostenfrei und mit Möglichkeit der beliebigen Weiterverarbeitung zur Verfügung steht.

1.3. Ziel der Arbeit

Wer ein Navigationsgerät erwirbt, zahlt in der Regel einen nicht unerheblichen Preis für das beiliegende digitale Kartenmaterial. Oftmals stellt sich erst nach dem Kauf heraus, dass das Material unvollständig oder sogar veraltet ist.

Welche Auswirkungen veraltete oder unvollständige Daten haben können, ist in Abbildung 1.1 zu sehen. Das Beispiel zeigt den Verlauf einer Straße in Schwabing. Die grünen Linien wurden anhand der Daten der DDG generiert und auf ein Satellitenbild projiziert. Wie man gut erkennen kann, fehlen prägnante Knoten, die für den *wahren* Verlauf der Straße wichtig sind. Diese müssten in der Regel wieder durch Messungen per GPS oder ähnlichem erhoben und in die Datenbank aufgenommen werden.

Mithilfe von OpenStreetMap besteht nun die Möglichkeit, die Sammlung frei zugänglicher Daten zu nutzen und die Straßennetzdaten der DDG mit einem geeigneten Algorithmus zu optimieren. Durch diesen Ansatz können nicht nur Kosten gespart, sondern auch Aktualisierungen der Tabelle autonom und regelmäßig ohne jeglichen Aufwand europaweit durchgeführt werden, was ein hohes Maß an Effizienz und Flexibilität mit sich

[2] mehr Informationen unter `www.ddg.de/3_leist_6_datenerf_fcd.html`
[3] zu finden unter `download.geofabrik.de/osm`

bringt.

Auf Grundlage des im Rahmen der Promotionsarbeit von Frau Dorothee Kunze entstandenen Teilproblems wird in dieser Arbeit versucht, einen derartigen Algorithmus zu finden und zu implementieren.

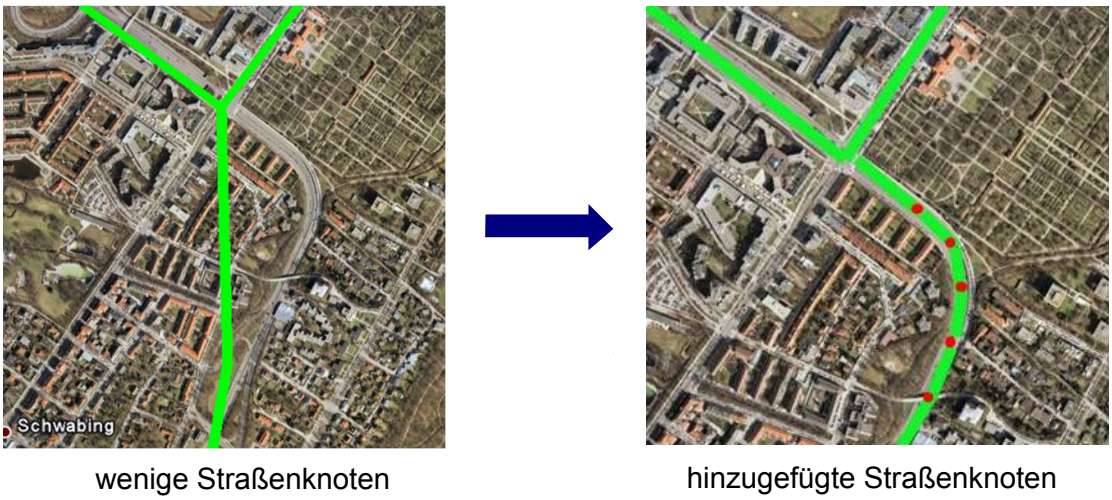

Abbildung 1.1: **Ziel des Algorithmus.** Links: Straßenverlauf, generiert anhand der Daten aus der Datenbank vor - und rechts nach der Optimierung mithilfe der Daten aus OSM.

2. Allgemeine Grundlagen

Die vorliegende Arbeit verwendet Elemente aus diversen Bereichen der Informatik. Diese erstrecken sich von der Geoinformatik (Rechnen mit und Visualisierung von geografischen Objekten) über die Integration und Aufbereitung von Daten (parsen und importieren von XML-Dateien) in eine Datenbank, der künstlichen Intelligenz (Graphenalgorithmen zur Lösung kombinatorischer Probleme), bis hin zur Computergrafik (Visualisierung der Daten mittels *GoogleEarth*/*GoogleMaps* über KML[4]).

Da in der Regel nicht jedem alle diese Techniken bekannt sind, geben die nachfolgenden Abschnitte die nötigen Hintergründe zum Verständnis der weiteren Vorgehensweise.

2.1. Extensible Markup Language

Die *Extensible Markup Language* (XML), ist eine vom *World Wide Web Consortiums* (W3C) herausgegebene Spezifikation. Sie definiert eine Meta-Markup-Sprache (Auszeichnungssprache), auf deren Grundlage konkrete Markup-Sprachen (u.a. RSS, MathML) definiert werden können, die zur Darstellung hierarchisch strukturierter Daten in Form von Textdaten dienen. Im Allgemeinen wird sie aufgrund ihrer hohen Portabilität unter anderem für den Austausch von Daten zwischen Computersystemen eingesetzt, speziell über das Internet. XML-Dokumente sind somit unabhängig von Anwendung und Plattform austauschbar und können jederzeit wiederverwendet werden.

Da die zur Verfügung stehenden Hilfsmittel zur XML-Manipulation für *Java* am weitesten entwickelt sind, werden im weiteren Verlauf der Arbeit nur Werkzeuge für die XML Bearbeitung in dieser Sprache behandelt.

2.1.1. Möglichkeiten der XML-Bearbeitung in Java

Das *Java API for XML Processing* (JAXP), ist ein standardisiertes API zum Validieren, Parsen, Generieren und Transformieren von XML-Dokumenten, dessen jeweilige Implementierung austauschbar ist.

[4] Keyhole Markup Language, ist eine Auszeichnungssprache zur Beschreibung von Geodaten, speziell für Google Earth.

Die vier grundlegenden Schnittstellen sind hier aufgeführt.

- *Document Object Model*
- *Simple API for XML*
- *Streaming API for XML*
- *XSLT*-Schnittstelle, um Transformationen an Daten und Strukturen eines XML-Dokuments zu ermöglichen.

Im Folgenden werden nur die ersten drei APIs vorgestellt, da mithilfe der XSLT-Schnittstelle nur Transformationen der inneren Struktur einer XML durchführt werden können und diese demnach nicht direkt zum Parsen geeignet ist.

Document Object Model (DOM) definiert eine vom W3C entwickelte abstrakte Programmierschnittstelle, die unabhängig von einer bestimmten Plattform oder Sprache ist, um Programmen und Skripts einen wahlfreien Zugriff auf den Inhalt und die Struktur eines XML-Dokuments zu ermöglichen. Sie parst ein ganzes XML-Dokument und erstellt eine vollständige „in memory"-Darstellung[5], die bestimmte Methoden des Zugriffs und der Veränderung bereitstellt. Zudem bietet die Schnittstelle Möglichkeiten zum Schreiben von XML-Dokumenten, ist aber im Allgemeinen für größere Dokumente eher ungeeignet.
JDOM ist eine einfache Möglichkeit, XML-Dokumente leicht und effizient mit einer Java-API zu nutzen. Im Gegensatz zu SAX und DOM, die unabhängig von einer Programmiersprache sind, wurde JDOM speziell für Java entwickelt. Während das Original-DOM keine Rücksicht auf die Java-Datenstrukturen nimmt, nutzt JDOM konsequent die Collection-API. Auch ermöglicht JDOM eine etwas bessere Performance und eine bessere Speichernutzung als beim Original-DOM [**?**], bleibt jedoch weiterhin für größere Dokumente ungeeignet.

Simple API for XML (SAX) ist von David Megginson zum schnellen Verarbeiten der Daten entworfen worden und ermöglicht es, ein XML-Dokument sequenziell abzuarbeiten. Sie basiert auf einem Ereignismodell, d.h. immer wenn dabei ein bestimmtes Ereignis abtritt, etwa das Auftauchen eines Start-Tags, werden bestimmte Methoden aufgerufen, die der Anwendung Informationen bereitstellen, auf die sie dann reagieren kann. Im Unterschied zu DOM wird kein Objektbaum im Speicher aufgebaut, sodass selbst bei großen XML-Dokumenten nur wenige Speicheressourcen benötigt werden. Dies ist aber mit dem Nachteil verbunden, dass wahlfreier Zugriff auf ein einzelnes Element nicht ohne

[5] Das gesamte XML-Dokument wird für die weitere Verwendung in Form eines Objektbaums in den Arbeitsspeicher abgelegt.

Zwischenspeicherung möglich ist.

Streaming API for XML (StAX) fordert im Gegensatz zu StAX aktiv den nächsten Teil eines XML-Dokuments an (*Pull*-API). Das Prinzip entspricht dem *Iterator Design Pattern*, das auch von der Collection-API bekannt ist. Es werden die beiden grundsätzlichen Verarbeitungsmodelle »*Iterator*« und »*Cursor*« unterschieden. Die Verarbeitung mit dem Iterator ist flexibler, aber auch ein bisschen aufwändiger. Die Cursor-Verarbeitung ist einfacher und schneller, aber nicht so flexibel [**?**].

2.2. Datenbankanbindung

Zurzeit stellen ODBC (*Open Database Connectivity*) und JDBC (*Java Database Connectivity*) die einzige Möglichkeit dar, von einer Client-Seite komfortabel auf Datenbanken zuzugreifen. Sie sind APIs von Microsoft bzw. JavaSoft, die Schnittstellen zur Verfügung stellen, um von höheren Programmiersprachen aus Datenbanken mittels SQL[6] zu verwalten und abzufragen. SQL alleine kann hier dagegen nicht überzeugen, da es noch zu stark vom Datenbanksystem abhängig ist.

ODBC ist ein API von Microsoft für den Zugriff auf Daten in relationalen und nicht relationalen Datenbank-Management-Systemen. Mit Hilfe der ODBC-API können Applikationen auf Daten zugreifen, die in Databasemanagement Systemen (DBMS) auf PCs oder Servern abgelegt sind, selbst wenn diese DBMSe andere Datenformate und andere Programmierschnittstellen verwenden.

Die Unterstützung von ODBC ist nicht auf Microsoft beschränkt. Fast alle DBMS-Hersteller unterstützen ODBC, unter anderem Oracle, Informix, Novell, Borland und IBM. ODBC ist eine plattformübergreifende Lösung. Es existieren Portierungen für Apple Macintosh, zahlreiche Unix-Plattformen wie Sun Solaris oder IBM AIX und IBM OS/2. Diese Portierungen werden nicht von Microsoft erstellt, sondern von den Betriebssystemherstellern selbst übernommen.

JDBC ist ein API von JavaSoft zum Absenden von SQL-Befehlen aus der Sprache Java heraus. Auch JDBC ist kein echtes Akronym für Java Database Connectivity. Es kann jede Datenbank angesprochen werden, für die ein JDBC-Driver existiert. Über die standardmäßig mitgelieferte JDBC-ODBC-Bridge kann darüber hinaus auch jede

[6] Structured Query Language ist eine Datenbanksprache zur Definition, Abfrage und Manipulation von Daten in relationalen Datenbanken

ODBC-Datenbank benutzt werden. Da JDBC auf Java basiert, erbt es automatisch alle Vor- und Nachteile von Java. So zum Beispiel die hohe Portabilität, die Benutzung als Applet aus einem Browser heraus, aber auch die geringere Geschwindigkeit als ein C-Programm mit ODBC.

Der große Vorteil von JDBC liegt in der Sprache Java. Es lassen sich so leicht Applets erstellen, und man kann auf eine Datenbank von außen über eine *www*-Seite zugreifen. Das ist gerade im ständig wachsenden Markt für E-Commerce ein bedeutender Faktor.

ODBC und JDBC im Vergleich

Der Vorteil von ODBC liegt in der guten MS-Windows-Einbindung. So lassen sich die Treiber sehr elegant über Dialogboxen installieren und konfigurieren. Daraus ergibt sich allerdings auch der Nachteil, dass ODBC praktisch nur für MS-Windows verfügbar ist, abgesehen von einigen Unix-Systemen wie Solaris. Außerdem gibt es für jede Datenbank einen nativen Treiber. Dieser läuft aber in den meisten Fällen nur unter MS-Windows. Ein weiterer Vorteil liegt dagegen in der Skriptfähigkeit über Visual-Basic und damit auch dr Kommunikation mit den üblichen Office-Produkte wie MS-Word oder MS-Excel.

JDBC verfolgt hier einen anderen Ansatz. Die Programmierung ist zwar der von ODBC ähnlich, das Treiberkonzept ist allerdings völlig anders. Da die Treiber im Idealfall in Java programmiert sind, kann ein JDBC-Programm auf jeder beliebigen Plattform laufen, die Java unterstützt. Da JDBC nur mit Java zusammenarbeitet, ist die Einbindung in diese Sprache wesentlich harmonischer als ODBC in C++.

2.3. Graphentheorie

Der Graphentheorie wird eine große Bedeutung im Feld der Informatik - und hier insbesondere innerhalb der Komplexitätstheorie - zugeschrieben. Dies beruht darauf, dass nicht nur viele algorithmische Probleme auf Graphen zurückgeführt werden können, sondern zudem die Lösung graphentheoretischer Probleme oft auch auf Algorithmen basiert.

In der Graphentheorie besteht ein Graph anschaulich aus einer Menge von Punkten, die durch Linien verbunden sind. Während diese Punkte als „Knoten“ (*Node*) oder auch „Ecken“ (Vertex) bezeichnet werden, nennt man die Linien meist „Kanten„ (*Edge*) oder auch Bögen.

Formal wird zur Lösung eines Problems in der Regel wie folgt vorgegangen:

1. Modelliere das Problem als Graph.
2. Formuliere die Zielfunktion als Eigenschaft dieses Graphen.

3. Löse jetzt mit Hilfe eines Graphenalgorithmus.

Das Problem so zu modulieren, dass ein Graphenalgorithmus darauf angewandt werden kann, stellt in der Regel den größten Aufwand dar und ist in den meisten Fällen auch nur schwer zu erreichen. Ist dies aber geschafft, kann mithilfe der Zielfunktion oder eines Zielknotens problemlos ein *geeigneter* Algorithmus angewendet werden. Einen Algorithmus als geeignet einzustufen wird in der Regel von Faktoren wie *Vollständigkeit*, *Optimalität*, *Speicherverbrauch* und *Laufzeit* abhängig gemacht, die, wie in den nächsten Abschnitten beschrieben ist, stark variieren können.

Im Folgenden werden diverse Graphenalgorithmen zur Lösung derartiger Probleme vorgestellt, die sich im Wesentlichen in zwei Kategorien – den *Uninformed Search-Strategies* und den *Informed Search-Strategies* – einteilen lassen.

2.3.1. Uniformed Search-Strategies

Uninformierte Suchstrategien (auch *blind search* genannt) besitzen keine zusätzlichen Informationen über die Zustände außer den in der Problemdefinition vorgegebenen. Alles, was sie tun können, ist, Nachfolger zu erzeugen und einen Ziel-Zustand von einem Nicht-Ziel-Zustand zu unterscheiden. In diesem Abschnitt werden die wesentlichen Suchstrategien erläutert.

Breadth-First-Search (BFS) Die *Breitensuche* ist eine einfache Strategie, wobei der Wurzelknoten als Erster expandiert wird, dann alle Nachfolger des Wurzelknotens und dann deren Nachfolger usw. Im Allgemeinen werden zuerst alle Knoten einer bestimmten Tiefe im Suchbaum expandiert, bevor Knoten in der nächsten Ebene expandiert werden (siehe Abbildung 2.1).

Formal arbeitet der Algorithmus wie folgt:

1. Bestimme den Knoten, an dem die Suche beginnen soll, und speichere ihn in einer Warteschlange ab.
2. Entnimm einen Knoten vom Beginn der Warteschlange und markiere ihn.
3. Falls das gesuchte Element gefunden wurde, brich die Suche ab und liefere „gefunden“ zurück. Anderenfalls hänge alle bisher unmarkierten Nachfolger dieses Knotens, die sich noch nicht in der Warteschlange befinden, ans Ende der Warteschlange an. Wiederhole Schritt 2.
4. Wenn die Warteschlange leer ist, dann wurde jeder Knoten bereits untersucht. Beende die Suche und liefere „ nicht gefunden“ zurück.

Die Suche ist *vollständig*, wenn sich der Zielknoten auf einer endlichen Tiefe d befindet. Der *flachste* Zielknoten ist aber nicht zwingend der *optimalste*. Allerdings beträgt der Speicherplatzverbrauch $O(|V| + |E|)$ (V für die Anzahl der Knoten (*Vertex*), E für die Anzahl der Kanten (*Edge*) im Graphen) und ist daher für größere Probleme ungeeignet.

Da im schlimmsten Fall alle möglichen Pfade zu allen möglichen Knoten betrachtet werden müssen, beträgt die Laufzeit der Breitensuche ebenfalls $O(|V| + |E|)$.

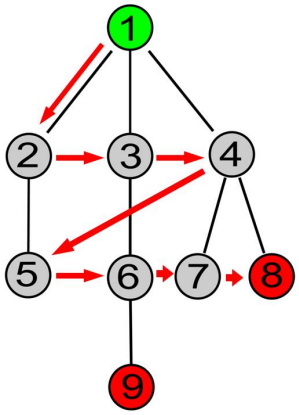

Abbildung 2.1: Breitensuchverlauf für einen einfachen Baum

Depth-First Search (DFS) Die *Tiefensuche* expandiert immer die tiefsten Knoten in dem aktuellen Randbereich des Suchbaums. Sie geht unmittelbar auf die tiefste Ebene des Suchbaums, wo die Knoten keinen Nachfolger haben. Wenn diese Knoten expandiert werden, werden sie aus dem Randbereich entfernt, sodass die Suche dann zum nächstflacheren Knoten weitergeht, der noch nicht erkundete Nachfolger aufweist (siehe Abbildung 2.2).

Formal kann der Algorithmus wie folgt beschrieben werden:

1. Bestimme den Knoten an dem die Suche beginnen soll.
2. Expandiere den Knoten und speichere alle Nachfolger in einem Stack.
3. Rufe rekursiv für jeden der Knoten in dem Stack DFS auf.
    - Wenn die Warteschlange leer ist, dann wurde jeder Knoten bereits untersucht. Beende die Suche und liefere „nicht gefunden“ zurück.
    - Falls der *Stack* leer sein sollte, tue nichts.
    - Falls das gesuchte Element gefunden worden sein sollte, brich die Suche ab

und liefere ein Ergebnis.

Falls ein Graph unendlich groß ist oder kein Test auf Zyklen durchgeführt wird, so ist die Tiefensuche nicht *vollständig*. Es kann also sein, dass ein Ergebnis - obwohl es existiert - nicht gefunden wird. Zudem ist sie insbesondere bei monoton steigenden Pfadkosten nicht *optimal*, da eventuell ein Ergebnis gefunden wird, zu welchem ein sehr viel längerer Pfad führt als zu einem alternativen Ergebnis. Dafür wird ein solches Ergebnis i.A. deutlich schneller gefunden als bei der (in diesem Fall optimalen, aber sehr viel speicheraufwendigeren) Breitensuche.

Die Speicheranforderungen sind sehr gemäßigt, da sie nur einen einzigen Pfad von der Wurzel zu einem Blattknoten, zusammen mit den verbleibenden nicht expandierten Geschwisterknoten für jeden Knoten auf dem Pfad, speichern muss.

Da im schlimmsten Fall alle möglichen Pfade zu allen möglichen Knoten betrachtet werden müssen, beträgt die Laufzeit $O(|V| + |E|)$.

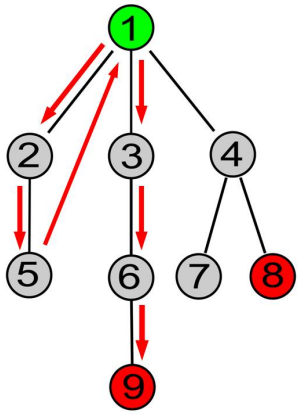

Abbildung 2.2: Tiefensuchverlauf für einen einfachen Baum

Eine Variante der Tiefensuche ist die so genannte **Backtracking-Suche**, die noch weniger Speicher benötigt, indem nur jeweils ein einziger Nachfolger erzeugt wird und nicht alle Nachfolger auf einmal; jeder partiell expandierte Knoten merkt sich, welcher Nachfolger als nächstes erzeugt werden soll.

Depth-Limited Search (DLS) Das Problem unbegrenzter Bäume kann abgeschwächt werden, indem man eine Tiefensuche mit vorgegebener Tiefenbegrenzung l bereitstellt. Das bedeutet, die Knoten der Tiefe l werden behandelt, als hätten sie keinen Nachfolger. Dieser Ansatz wird auch als *tiefenbeschränkte Suche* bezeichnet.

Formal arbeitet der Algorithmus wie folgt:

1. Bestimme den Knoten, an dem die Suche beginnen soll, und bestimme die maximale Suchtiefe.
2. Prüfe, ob der aktuelle Knoten innerhalb der maximalen Suchtiefe liegt.
   - Falls nein: Tue nichts.
   - Falls ja:
     a) Expandiere den Knoten und speichere alle Nachfolger in einem Stack.
     b) Rufe rekursiv für jeden der Knoten des Stacks DLS auf und gehe zu Schritt 2.

Sie führt dadurch auch eine zusätzliche Quelle der Unvollständigkeit ein, wenn $l < d$ gewählt wird; für $l > d$ ist sie zudem nicht *optimal*.

Iterative Deepening Depth-First Search (IDS) Die *iterativ vertiefte Tiefensuche* ist eine allgemeine Strategie, die häufig in Kombination mit der Tiefensuche verwendet wird, die die beste Tiefenbegrenzung ermittelt. Dazu erhöht sie schrittweise die Begrenzung beginnend ab 0, bis ein Ziel gefunden ist. Das passiert, wenn die Tiefenbegrenzung d erreicht, die Tiefe des flachsten Zielknotens.

Die iterative Tiefensuche kombiniert die Vorteile der Tiefen- und der Breitensuche und hat folgenden Ablauf:

1. Bestimme den Knoten, an dem die Suche beginnen soll.
2. Rufe beschränkte Tiefensuche mit der aktuellen Suchtiefe auf.
3. Erhöhe die Suchtiefe um 1 und gehe zu Schritt 2.

Bidirectional Search Die Idee bei der *bidirektionalen Suche* ist, zwei gleichzeitige Suchvorgänge auszuführen. Einen vom Ausgangszustand, den anderen vom Ziel aus. Beide werden angehalten, sobald sich die beiden Suchprozesse in der Mitte treffen (Abbildung 2.3).

Die Motivation dabei ist, dass die Laufzeit und der Speicherverbrauch wesentlich geringer sind, oder wie in Abbildung 2.3 bildlich dargestellt, dass die Fläche der beiden kleinen Kreise kleiner als die Fläche eines großen Kreises ist, dessen Mittelpunkt am Start liegt und dessen Radius bis zum Ziel verläuft. Sie wird implementiert, indem man veranlasst, dass einer oder beide Suchläufe jeden Knoten überprüfen, bevor dieser expandiert wird, um festzustellen, ob er in der Verzweigung des anderen Suchbaums liegt; in diesem Fall wurde eine Lösung gefunden.

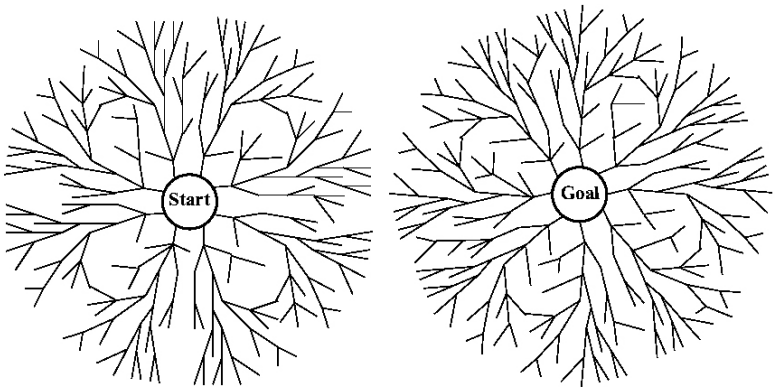

Abbildung 2.3: Schematische Ansicht einer bidirektionalen Suche, die erfolgreich ist, wenn eine Verzweigung vom Startknoten auf eine Verzweigung vom Zielknoten trifft

2.3.2. Informed (Heuristic) Search-Strategies

Während uninformierte Suchstrategien Lösungen für Probleme suchen, indem sie systematisch neue Zustände erzeugen und sie mit dem Ziel vergleichen, berücksichtigt die informierte Suche problemspezifisches Wissen und kann so effizienter Lösungen finden. Hier werden die zu expandieren Knoten von einer *Evaluierungsfunktion* $f(n)$ abhängig gemacht. Eine Schlüsselkomponente dieser Algorithmen ist eine *Heuristikfunktion* $h(n)$, die zusätzliches Wissen über das Problem mitteilt. Bekannte Beispiele derartiger Strategien sind u.a. *Greedy Best-First Search* (Bestensuche) und A^*, oder *Hillclimbing* und *Simulated-Annealing*, wenn es sich um einen *lokalen* Suchalgorithmus handelt.

Für die vorliegende Arbeit sind diese Suchstrategien allerdings weniger von Bedeutung, weshalb an dieser Stelle nicht näher auf sie eingegangen wird.

3. Aufbereitung & Integration der Daten

Ausgehend von den zuvor dargelegten Grundlagen gilt es nun, einen Algorithmus zu entwickeln, der letztendlich eben diese Optimierung der EGT durchführt. Dazu müssen zunächst die Daten der DDG und der von OSM, die je in einer XML-Datei vorliegen, aufgrund ihrer Größenordnung in eine gemeinsame Datenbank überführt werden. Das Problem hierbei sind die von Grund auf unterschiedlichen Konzepte und somit auch Strukturen der Datenmodelle, die in den Abschnitten 3.1 und 3.2 beschrieben werden. Darauf stützend kann anschließend eine geeignete Datenbank modelliert (Abschnitt 3.3), die Daten importiert (Abschnitt 3.4ff) und die EGT anschließend optimiert werden. Ein grobes Schema der gesamten Applikation ist in Abbildung 3.1 dargestellt.

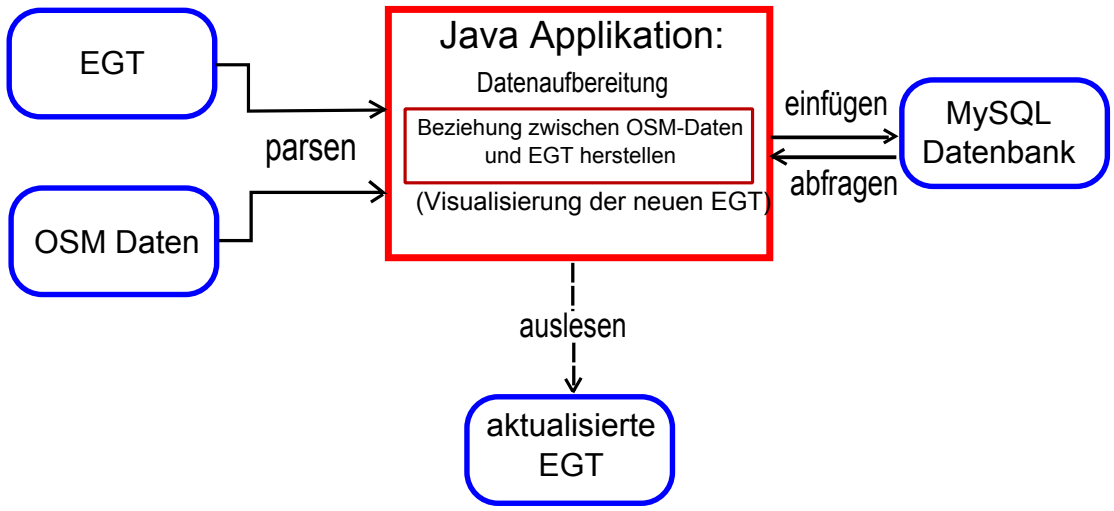

Abbildung 3.1: Schema

Dieser Abschnitt befasst sich zunächst nur mit der reinen Aufbereitung und Integration der Daten der jeweiligen Partei. Hierzu werden vorab die jeweiligen Datenmodelle mit den jeweiligen XML-Strukturen erläutert, die die Grundlage für die Implementierung der nachfolgenden XML-Parser darstellen.

3.1. DDG-Datenmodell

Die von der DDG erfassten Daten werden für den *offline*-Betrieb in einer so genannten Endgerätetabelle (EGT) abgespeichert, die zwei mal im Jahr aktualisiert wird. Die Tabelle enthält alle verkehrstelematisch relevanten Informationen und kann in verschiedenen Telematik-Endgeräten eingesetzt werden; sie stellt somit eine Schnittstelle zu den Kunden dar.

Wichtige verkehrstelematische Objekte stellen dabei unter anderem Kreuzungspunkte von Straßen (Bundesautobahnen, Bundesstraßen) und sogenannte *Points of Interest*[7] (POI) dar. Diese Objekte werden geocodiert, indem aus den geographischen Koordinaten eines zentralen Punktes des Objektes der Geocode[8] berechnet wird. Die Vergabe von Geocodes stellt also eine Möglichkeit dar, Verkehrsinformationen zu referenzieren, die für Europa einen einheitlichen Ortsbezug herstellen. Sie sind notwendig, damit geocodierte eindeutige Ortsbeschreibungen von Verkehrsmeldungen in den Telematik-Endgeräten von den Kunden entschlüsselt und genutzt werden können. Geocodes selbst sind jedoch nicht zwangsläufig eindeutig: durch Rundungsfehler bei der Berechnung oder mehrfach-Nutzung eines Knotens für verschiedene Objekte können diverse Einträge so den selben Geocode erhalten.

Jedes verkehrstelematisch interessante Objekt erhält demnach anhand seiner Lage einen Geocode. Dieser wird zusammen mit anderen Informationen in die EGT aufgenommen. Darunter befindet sich unter anderem eine ID zur eindeutigen Identifizierung, ein Typ, eine Straße, ein Name, ein Längen- und Breitengrad, sowie der Geocode des Vorgängers und Nachfolgers, wenn es sich z.B. um eine Straße handelt. Eine Auflistung und Beschreibung aller Spalteneinträge ist im Anhang C auf Seite 77 zu finden.

Die gesamte Menge aller Daten lassen sich in drei logischen Gruppen einteilen:

- *Punktobjekte:* Hierunter fallen Straßenknoten (Geocodetypen 1, 2, 3, 5, 6, 7, 8, 30, 31, 32, 33, 34, 35), Points of Interest (Geocodetyp 21) sowie Straßen-POI (Geocodetyp 22) und Richtungsorte (Geocodetyp 23).
- *Flächenobjekte:* Hierunter fallen die Verwaltungsgebietseinheiten (Geocodetypen 11, 12, 13, 14, 17, 18, 19) sowie die sonstigen Gebiete (Geocodetypen 16, 20).
- *Linienobjekte:* Hierunter fallen die Straßenabschnitte (Geocodetyp 92), die Umleitungsstrecken (Geocodetyp 94), die Bundesstraßen (Geocodetyp 98) und die Autobahnen (Geocodetyp 99).

[7] Punkte von besonderem Interesse
[8] die genaue Berechnung des Geocodes ist im Anhang D auf Seite 79 zu finden

Da die vorliegende Arbeit ausschließlich Straßenverläufe behandelt, sind an dieser Stelle nur die Knotenpunkte der Straßen - zugehörig zur Gruppe der Punktobjekte - von Interesse.

Straßenverläufe können anhand der Geocodes von Vorgänger-Nachfolger-Relationen nachvollzogen werden. Da diese in der EGT für einen Straßenknoten nicht eindeutig sind, müssen die Geocodes mit der Information *Strasse* konkateniert werden. Dadurch entsteht ein eindeutiger *Key*, mithilfe dessen die Vorgänger- bzw. Nachfolger-Knoten gefunden werden können.

Besteht eine Straße beispielsweise aus zwei Knoten, so hat der erste Knoten keinen Vorgänger und als Nachfolger den Geocode des zweiten Knotens. Andererseits ist für den zweiten Knoten kein Nachfolger eingetragen, dafür aber der Geocode des ersten Knoten in der Spalte des Vorgängers. Will man den Eintrag finden, der zu dieser Straße gehört, muss der Geocode verknüpft mit der Information aus der 'Strassen'-Spalte des Nachfolgers bzw. Vorgängers gesucht werden.

Ein Sonderfall bei der Festlegung von Vorgänger-Nachfolger-Relationen sind Ringstraßen. Hierbei sind zwei Fälle zu unterscheiden:

1. Straßen, die auf einem Teilstück einen getrennten Verlauf haben, d.h. sich an einem Knoten in zwei gleichwertige Verzweigungen teilen und sich später wieder zu einem gemeinsamen Verlauf vereinigen (wie z.B. Ringstraßen um Stadtzentren);
2. Straßen, die durch ihren gesamten Verlauf einen in sich geschlossenen Ring bilden.

Bei Zwischenknoten werden als Vorgänger- und Nachfolger-Knoten die Geocodes der benachbarten regulären Knoten genannt. Bei den benachbarten Knoten wird der Geocode des Zwischenknotens als Vorgänger bzw. Nachfolger genannt.

3.1.1. Aufbau der EGT-XML-Datei

Das Verständnis über den Aufbau der Dateien ist für das Parsen und den Import der Daten in die MySQL-Datenbank enorm wichtig. In Abschnitt 3.1 wurde bereits das Konzept der Datenerfassung der DDG erläutert.

Der Nachfolgende XML-Code zeigt einen Ausschnitt der ersten beiden Einträge aus der EGT. Der erste Eintrag besteht nur aus den Tabellenspaltennamen, die ebenfalls in Abschnitt 3.1 aufgeführt sind. Auf der rechten Seite ist ein kompletter Knoten aufgelistet, wie er in der Tabelle zu finden ist.

```
1 <Workbook>
2     <Row>                                   <Row>
```

```
3       <d>EGT_ID</d>                          <d>1</d>
4       <d>EGT_GEOCODE</d>                     <d>4170057181</d>
5       <d>EGT_LFDNR</d>                       <d>14</d>
6       <d>EGT_TYP</d>                         <d>1</d>
7       <d>EGT_STRASSE</d>                     <d>A1</d>
8       <d>EGT_NAME</d>                        <d>Hamberge</d>
9       <d>EGT_REFGEOCODE</d>                  <d>4169336270</d>
10      <d>EGT_NUMMERRICHTUNGNACHORT</d>       <d>24</d>
11      <d>EGT_NUMMERRICHTUNGVONORT</d>        <d>24</d>
12      <d>EGT_VONORT</d>                      <d>Lübeck</d>
13      <d>EGT_NACHORT</d>                     <d>Hamburg</d>
14      <d>EGT_LAENGE1</d>                     <d>10.5945699999</d>
15      <d>EGT_BREITE1</d>                     <d>53.84975</d>
16      <d>EGT_LAENGE2</d>                     <d>10.5945699999</d>
17      <d>EGT_BREITE2</d>                     <d>53.84975</d>
18      <d>EGT_STRASSEINT</d>                  <d>E22</d>
19      <d>EGT_NAMEKURZ</d>                    <d>Hamberge</d>
20      <d>EGT_VORGAENGER</d>                  <d>4171367916</d>
21      <d>EGT_NACHFOLGER</d>                  <d>4166256060</d>
22    </Row>                                 </Row>
23  </Workbook>
```

Der Aufbau hier ist recht simpel. Nach der konventionellen XML-Deklaration, die an erster Stelle des Dokuments aufgeführt ist, vermerkt das Schlüsselwort *Workbook* in dem ersten Knoten, dass hier die Tabelle beginnt. Die Zeilen zwischen <Row> und </Row> enthalten die Informationen eines einzelnen *Node*-Eintrags.

3.2. OSM-Datenmodell

Im Gegensatz zu dem Konzept der DDG, in der ein einzelner Eintrag alle nötigen Informationen enthält, gruppiert OpenStreetMap die Daten in konkrete Untergruppen. Sie unterscheiden im Groben zwischen drei Grundelementen (Data-Primitives), den *Nodes*, *Ways* und *Relations*. Ein weiteres, aber nicht als eigenes Grundelement angesehenes Primitiv, sind die *Areas*. Sie sind nur Spezialfälle der *Ways*. Diesen Elementen können bestimmte Merkmale (*tags*) zugewiesen werden. Diese Merkmale bestehen immer aus einem Schlüssel (*key*) und einem Wert (*value*).

Ein Daten-Primitiv ist eine Objektklasse, die mithilfe einer API auf einen Server gespei-

chert werden kann und stellt die oberste Ebene der Objekt-Hierarchie dar. Diese können durch adäquate *MapFeatures*[9] z.B. in Straßen, Haltestellen, Point of Interest oder Gebäude spezifiziert werden. Jedes Objekt kann dabei folgende drei Attribute aufnehmen, die im eigentlichen Sinn keine Beschreibung des Objekts liefern.

- **user, timestamp**: Author und Zeitpunkt der letzten Modifizierung [13].
- **visible**: Unabhängig davon, ob das Objekt aus der Datenbank gelöscht wurde oder nicht: sobald das Attribut auf *false* gesetzt ist, kann es nur über *History Calls* zurückgeholt werden [13].

Alle anderen Attribute sind objektspezifisch und können aus den oben genannten MapFeatures entnommen werden.

Im Folgenden werden die einzelnen Objekt-Primitive beschrieben und anschließend auf die wichtigsten Attribute eingegangen.

Nodes sind die Grundelemente von OpenStreetMap. Ein *Node* ist ein geometrischer Punkt, dessen Position durch Breiten- und Längengrad bestimmt ist (engl.: latitude und longitude).

Sie können

- einen einzelnen geografischen Punkt oder Ort beschreiben, beispielsweise Sehenswürdigkeiten, Ortschaften, Städte, POI).
  Dazu werden den Knoten Eigenschaften zugewiesen. (siehe MapFeatures)
- den Start- und den Endknoten und alle Zwischenknoten einer Linie beschreiben.
- alle Knoten des Umrisses einer Fläche beschreiben.
- die Höhe über oder unter Meeresspiegel angeben.

Alle weiteren Eigenschaften werden mit Merkmalen spezifiziert die hier aufgeführt sind:

- **id** Eine natürliche Zahl, die größer gleich eins und innerhalb der *Node*-Einträge eindeutig sein muss. Ein *Way* kann also die gleiche ID wie ein *Node* besitzen.
- **lat** Breitengrad. Eine Zahl zwischen -90 und 90.
- **lon** Längengrad. Eine Zahl zwischen -180 and 180.
- **tags** Eine Menge von Key/Value Paaren, in der kein Key doppelt vorkommen darf.

In den Anwendungen *Ways* und *Areas* werden den einzelnen Punkten allerdings keine Eigenschaften in Form von Schlüsseln und Werten zugewiesen, um eine unnötige Vergrößerung der Datenbank zu vermeiden.

[9] zu finden unter `http://wiki.openstreetmap.org/wiki/Map_Features`

Stehen (Nodes) in keinem weiteren Kontext, stellen sie im eigentlichen Sinne nur Punktwolken im Raum dar (siehe Abbildung 3.2). Nur durch Referenzierung über andere Primitive kann ihnen eine größere Bedeutung zugeschrieben werden.

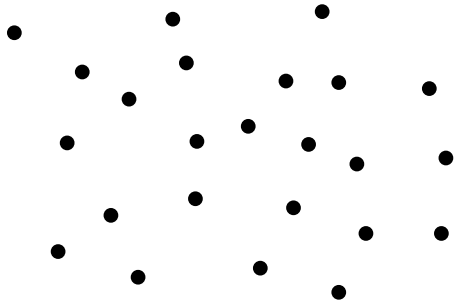

Abbildung 3.2: **Punktwolke:** Ohne weitere Referenzierung stellen *Nodes* im Grunde genommen nur Punkte im Raum dar.

Ways sind ebenfalls Grundelemente. Sie bestehen aus mindestens zwei Punkten (*Nodes*) und deren gerader Verbindung, oder aus einem Linienzug mit mehreren Punkten und den geraden Verbindungen vom einen Punkt zum nächsten. Sie dienen zur geometrischen Darstellung des geografischen Verlaufes von Linien im Gelände – also z.B. Straßen, Wege, Eisenbahnschienen, kleine Flüsse, Zäune, Überlandleitungen, Flächenumrisse, Gebäudeumrisse (u.v.m) und haben eine Richtung, die je nach zugewiesener Eigenschaft beispielsweise die Richtung einer Straße oder die Fließrichtung eines Gewässers beschreibt. Zudem bringen sie einen Teil der *Nodes* in einen Kontext, sodass aus Abbildung 3.2 nun beispielsweise Beziehungen wie in Abbildung 3.3 abgebildet, hergestellt werden können.

Ein *Way* kennzeichnet sich durch *gleichbleibende* Eigenschaften für alle Teilstrecken, z.B. eine Autobahn, eine Bundesstraße, eine Landstraße, gleiche Oberflächenbeschaffenheit, gleiche zulässige Höchstgeschwindigkeit aus.

Hierbei müssen nicht nur die Pfeile aller Segmente in die gleiche Richtung weisen (= richtige Reihenfolge der *Nodes*), sondern auch die Reihenfolge der Segmente in einem *Way* muss zusätzlich geordnet sein. Beim Hinzufügen der Segmente zu einem *Way* muss man darum genau die Reihenfolge einhalten.

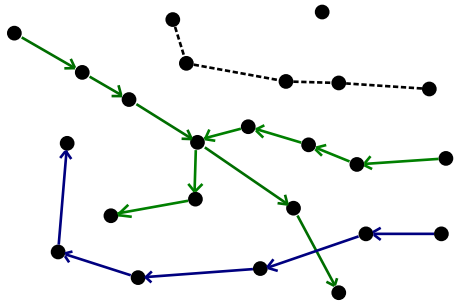

Abbildung 3.3: **OSM-Ways:** Sie enthalten Listen mit Verweisen auf *Nodes*, die den Verlauf des Segments wiedergeben. Die grünen Linien könnten z.B. Straßen, die blaue z.B. ein Fluss und die schwarze eine Grenze darstellen.

Einmal genutzte Knoten können auch durch andere Wege nochmals genutzt werden. Wege müssen in jedem Fall einen Schlüssel und einen zugehörigen Wert besitzen.

Die Grundlegenden Attribute sind hier aufgeführt:

- **id** wie bei *Nodes*
- **tags** wie bei *Nodes*
- **nodes** Eine Liste mit *Nodes*.

Relation, Area Diese Primitive tragen nicht zum weiteren Verständnis dieser Arbeit bei und werden aus diesem Grund nicht näher erläutert

3.2.1. Aufbau der OSM-XML-Datei

Ein Dokument der OSM (**.osm*) enthält die Einträge aller Daten-Primitive innerhalb eines Bereichs, wobei diese zu rund 80% aus einzelnen *Node*-, und zu 15% aus einzelnen *Way*-Einträgen bestehen. Die restlichen 5% machen in etwa die *Relations* und *Areas* aus, die hier aber nicht benötigt werden.

Abgesehen von der konventionellen XML-Deklaration beginnt jede OSM-XML-Datei mit den folgenden beiden Einträgen:

```
1 <osm version="0.5" generator="Osmosis 0.29">
```

```
2   <bound box="51.28559,6.45099,54.17288,11.64342"
3          origin="http://www.openstreetmap.org/api/0.5"/>
```

Das Wurzelelement *osm* in Zeile 1 beinhaltet ein optionales Bounds-Element, sowie eine Liste von allen *Nodes*, *Ways* und *Relations*. Das Attribut *version* steht für die API-Version. *Generator* bezeichnet das Programm, welches die XML-Datei erzeugt hat. Dieses Attribut wird beim Transfer zum Server gespeichert und ist nicht über die API zugänglich. Sie dient dazu, fehlerhafte Eingaben von defekten Editoren besser identifizieren zu können.

Obwohl das *Bounds*-Element, welches sich über die Zeilen 2 und 3 erstreckt, nicht in der DTD[10] erwähnt wird[11] , schreibt der Server ein solches Element in die XML-Datei. Es gibt den Bereich der Geodaten innerhalb der OSM-XML-Datei an.

Die erste Menge an Einträgen bilden die *Nodes*, gefolgt von den *Ways*, *Relations* und *Areas* in eben dieser Reihenfolge. Einen Auszug aus der *germany.osm* für einen *Node*-Eintrag ist hier zu sehen:

```
1 <node id="122440" lat="52.2615792" lon="8.0558196" user="Joe"
2     visible="true" timestamp="2009-10-28T18:45:22+00:00">
3   <tag k="created_by" v="JOSM"/>
4 </node>
```

Dies ist beispielsweise ein *Node* mit der ID 122440 und den geographischen Koordinaten 52.2615792, 8.0558196, das am 28.10.2009 von Joe erhoben wurde. Der Eintrag ist wegen *visible=true* zudem noch sichtbar.

Ein *Way* kann, wie dieses Beispiel zeigt, in etwa folgende Darstellung haben:

```
1 <way id="40746692">
2  <nid="56328536">
3  <nid="56359012">
4  <tag k="name" v="Iburger Straße"/>
5  <tag k="highway" v="primary"/>
6  <tag k="ref" v="B 68"/>
7 </way>
```

Hierbei handelt es sich um die Iburger Straße mit der ID 40746692. Sie besteht aus den beiden *Nodes* mit den IDs 56328536 und 56359012. Ein wichtiger Unterschied zur

[10] Document-Type-Definition
[11] siehe Anhang A auf Seite 71

DDG ist die Notation der Referenznummer. Während bei der DDG die Referenz ohne Leerzeichen notiert wird, ist dies in OSM Vorschrift, wie in Zeile 6 zu sehen ist.

3.3. Entwurf der Datenbank

Anhand der bisher beschriebenen Datenmodelle, kann an dieser Stelle schließlich eine geeignete Datenbank für die Datenintegration entworfen werden. Angestrebt ist zunächst ein lokales Datenbanksystem (DBS) mit zentraler Datenhaltung und relationalem Datenmodell. Aufgrund der hohen Geschwindigkeit, des geringen Speicherverbrauchs sowie der hohen Popularität von MySQL 5.1 wird die Datenbank an dieser Stelle verwendet und zu diesem Zweck lokal auf dem Rechner eingerichtet.

Die eigentliche Schwierigkeit liegt hier jedoch in dem Entwurf der Tabellen, in denen die Daten der DDG und der OSM abgelegt werden sollen. Hier können schon im Vorfeld evtl. später auftretende Probleme vereinfacht oder im Idealfall vermieden werden. Da

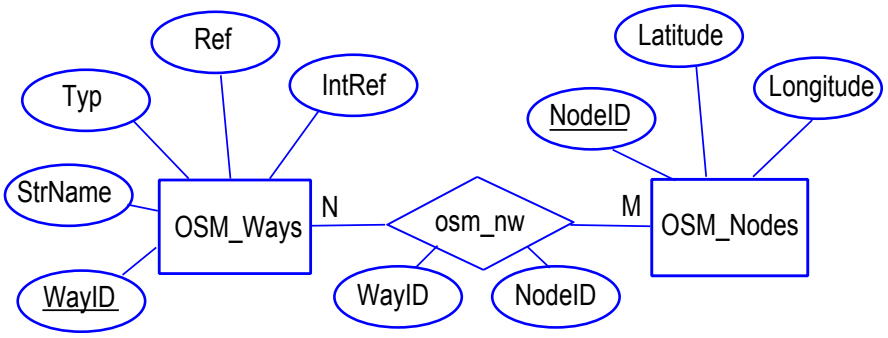

Abbildung 3.4: **OSM ER-Diagramm:** Tabellenentwurf für die Daten aus OpenStreetMap

die EGT im Gegensatz zu den Daten von OSM in einem nicht sehr intuitiven Format vorliegen, werden die Tabellen an das Datenmodell der OSM angepasst. Für die *Nodes* und *Ways* wird also je eine separate Tabelle angelegt. Deren N zu M Beziehung untereinander, die sich aus der Tatsache ergibt, dass ein *Node* in mehreren *Ways* enthalten sein kann und ein *Way* in der Regel aus mehreren *Nodes* besteht, wird durch je eine zusätzliche Tabelle repräsentiert. Im Großen und Ganzen ergeben sich die in Abbildung 3.5 und 3.4 zu sehenden ER[12]-Diagramme.

Leider können die in Abschnitt 3.2 angedeuteten *Relations* nicht verwendet werden, da diese nicht für alle Autobahnen und Bundesstraßen vollständig sind. Diese enthalten

[12] Entity-Relationship

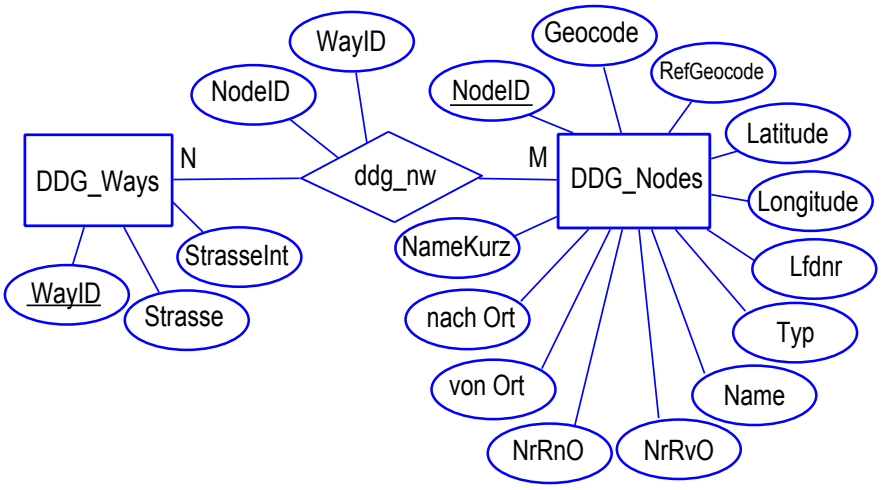

Abbildung 3.5: **DDG ER-Diagramm:** Tabellenentwurf für die EGT

Listen aller *WayIDs* zu einem größeren Straßen-Abschnitt, mithilfe dessen die *Nodes* hätten relativ intuitiv ermittelt werden können.

3.4. XML-Parsing & Daten-Integration

Nach dem Entwurf der Datenbank müssen nun die Daten der DDG und OSM aus den XML-Dateien in die MySQL-Datenbank importiert werden. Aus Abschnitt 2.1.1 ging bereits hervor, dass Java für die Arbeit mit XML-Dokumenten prädestiniert ist; daher werden auch die dort erläuterten Methoden für die weitere XML-Bearbeitung verwendet. Diese durchlaufen die Dateien und senden deren Datensätze über eine Schnittstelle an die Datenbank.

Für die Implementierung der XML-Importer ist je eine eigene Klasse vorgesehen, die je nach Anforderung der jeweiligen Daten auf die EGT (Klasse `EGT2MySQL`) bzw. OSM (Klasse `OSM2MySQL`) zugeschnitten ist und einen entsprechenden Parser enthält. Diese erben von der abstrakten Oberklasse `XML2MySQL`, die alle Basiskonstrukte zur Verfügung stellt. Als Schnittstelle zur Datenbank wird der `MySQLHandler` verwendet, die sich über JDBC mit der MySQL Datenbank verbindet und Methoden zum Senden von SQL Statements bereitstellt.

Die Applikation soll weitestgehend autonom sein; daher erstellt sie alle nötigen Tabellen von alleine. Diese Klassen mit deren grundlegenden Funktionen werden in den nachfol-

genden Abschnitten erläutert.

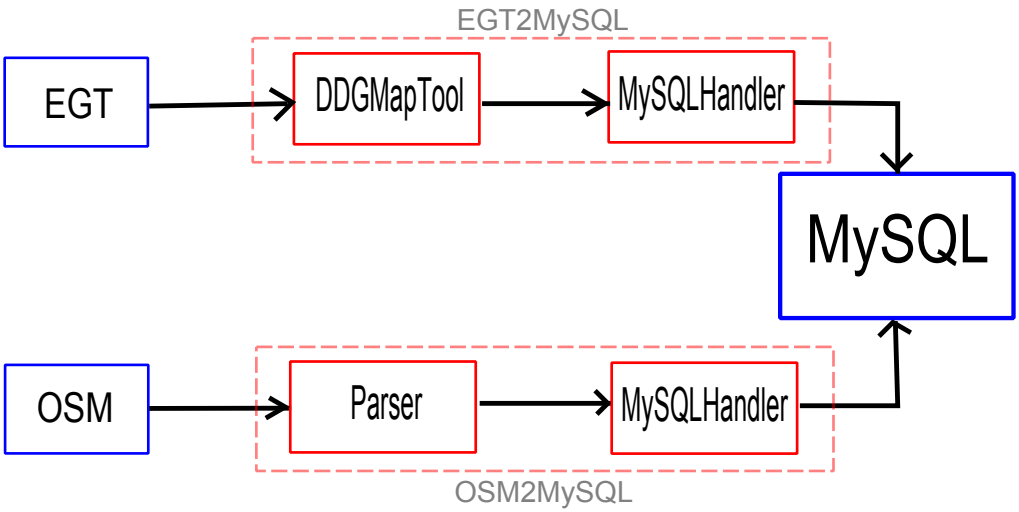

Abbildung 3.6: Schema der `EGT2MySQL` Klasse

Die Klasse `MySQLHandler`

`MySQLHandler` repräsentiert eine Verbindung zur MySQL Datenbank über JDBC und enthält neben ein paar wenigen Objektattributen, die für die Datenbankverbindung relevant sind, lediglich eine Handvoll Methoden. Die beiden wichtigsten sind in Code 1 und Code 2 aufgeführt und dienen zum Senden der SQL-Statements an die Datenbank. JDBC wird in diesem Fall aus den in Abschnitt 2.2 aufgeführten Gründen als Datenbankanbindung verwendet.

```
1 public ResultSet executeQuery(String query) throws
     SQLException {
2    ResultSet rs = stmt.executeQuery(query);
3    return rs;
4 }
```

Java-Code 1: `MySQLHandler.executeQuery`

Die Methode `executeQuery(query)` schickt SQL-Queries zum *Anfragen* (*Select*) an die Datenbank. Es wird also ein Ergebnis in Tabellenform erwartet, welches sich in dem `ResultSet` befindet.

```
1 public void executeUpdate(String query) throws SQLException{
2    stmt.executeUpdate(query);
3 }
```

Java-Code 2: `MySQLHandler.executeUpdate`

`executeUpdate(query)` schickt hingegen SQL-Queries an die Datenbank, die zu einem *Update* der Datenbank führen (`Insert, Drop, Update`, etc.); daher wird in diesem Fall kein Ergebnis erwartet.

Über den Konstruktor werden die notwendigen Attribute zur Datenbankverbindung initialisiert – der Default-Konstruktor verwendet Standardwerte.

Die Klasse `XML2MySQL`

Die abstrakte Basisklasse `XML2MySQL` definiert einen allgemeinen MySQL-Importer mit relativ kleinem Funktionsumfang. Sie enthält den Pfad zu der importierenden XML-Datei, den`MySQLHandler` zur Datenbankanbindung, sowie drei abstrakte Methoden zum Initialisieren des Parsers (`initParser`), Importieren der Daten (`importData`) und Löschen einer Tabelle in der Datenbank (`dropTable`).

Der Sinn dieser Klasse besteht darin, dass die zu implementierenden Parser im Wesentlichen die gleichen Funktionalitäten verwenden. Die folgenden beiden Unterabschnitte zeigen, wie die EGT und die Daten der OSM-Datei über Java in die Datenbank importiert werden.

3.4.1. EGT parsen und importieren

Der Umfang der EGT beschränkt sich auf rund 55.000 Einträge. Daher kann hier das DOM zum Parsen verwendet werden. Dazu sollte allerdings der *Java-Heap-Space* etwas angehoben werden[13], da es sonst schnell zu einem *Stack-Overflow* kommen kann. Vorteil dieser API ist, wie in Abschnitt 2.1.1 beschrieben, dass die Daten in einem Objektraum gespeichert werden, was den Komfort der Verarbeitung enorm steigert.

Die Klasse `EGT2MySQL` dient diesem Zweck, implementiert den DOM-Parser und stellt Funktionalitäten zur Aufbereitung und Senden der Daten an die Datenbank bereit. Abbildung 3.7 zeigt die grobe Arbeitsweise dieser Klasse.

[13] durch den Parameter -Xmx256m

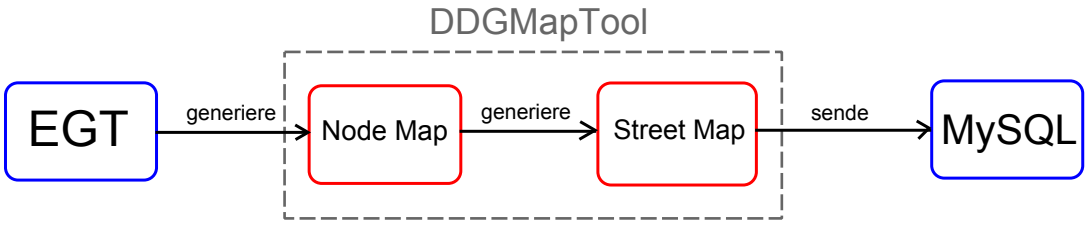

Abbildung 3.7: Schema der `EGT2MySQL` Klasse

Die Klasse `EGT2MySQL` und *DDGMapTool*

Als Unterklasse von `XML2MySQL` implementiert `EGTMySQL` die oben genannten abstrakten Methoden. Der eingebaute Parser stützt sich im Grunde genommen auf die bereits von Frau Kunze implementierte Klasse `EGTMapTool` (hier `DDGMapTool`) und nutzt dessen Funktionalität. Sie benötigt lediglich den Pfad zu der EGT, der über den Konstruktor weitergereicht wird, um die in Abbildung 3.7 zu sehende *NodeMap*, sowie daraus entstehend *StreetMap* zu generieren, dessen Inhalt an die Datenbank gesendet wird.

Dazu stellt die Klasse `DDGMapTool` die Methode *generate* breit, die, wie in Code 3 zusehen, je eine Methode zur Generierung dieser Maps aufruft.

```
1 public void generate() {
2     this.map = generateMap(); //<String, DDGNode>;
3     this.simpleStreets = generateStreets(map); //<DDGNodeList
          >
4 }
```

Java-Code 3: `DDGMapTool.generate`: Startet das Parsen der EGT

Zunächst wird also eine Map mit allen EGT-Einträgen mittels `generateMap()` erstellt. Dazu wird das gesamte XML-Dokument (EGT) über den *DOMParser*, in einem Objektbaum gespeichert.

```
1 DOMParser parser = new DOMParser();
2 parser.parse(mapfile);                    // DOM Tree
3 Document doc = parser.getDocument();   // XML Dokument
4 NodeList l = doc.getElementsByTagName("Row");
```

Java-Code 4: DDGMapTool.generateMap

Die EGT befindet sich nun in dem Objekt `doc`, aus welchem über `getElementsByTagName` bestimmte Mengen an Knoten herausgefiltert und in einer Liste abgespeichert werden können. In diesem Fall wird eine Liste sämtlicher Knoten mit dem Namen „Row“ angelegt. Aus dieser werden im weiteren Verlauf alle Informationen[14] zu jedem Row (gleichbedeutend mit einem EGT-Eintrag) geholt und nach dem *Typ*[15] gefiltert in die oben genannte Map gespeichert. Als Key für diese Map wird der in Abschnitt 3.1 erwähnte Key zur eindeutigen Identifizierung verwendet, der die Form *'Geocode+Strasse'* hat.

Auf Grundlage dieser Map muss eine weitere Map generiert werden, die aus den *Nodes* die Straßenzugehörigkeiten konstruiert, da letztendlich genau diese Zusammengehörigkeit gebraucht wird, um die in Abschnitt 3.3 aufgezeigten Tabellen der DDG mit Inhalt zu füllen.

```
1  HashMap<String, List<DDGNode>> streets;
2  streets = new HashMap<String, List<DDGNode>>();
3
4  for (DDGNode mynode : map.values()) {
5     key = mynode.getStrasse(); // Strasse besorgen
6
7     if (streets.get(key) == null) {
8        streets.put(key, new ArrayList<DDGNode>());
9     }
10
11    streets.get(key).add(mynode);
12 }
```

Java-Code 5: `DDGMapTool.generateStreets`

Um dies zu ermöglichen wird zunächst von jedem *Node* die *Strasse* in eine Map geschrieben, zu der jeweils eine Liste angelegt wird, die sämtliche *Nodes* dieser Straße enthalten.

Die Map wird also komplett durchlaufen und der aktuelle *Node* betrachtet. Enthält die *streetMap* keinen entsprechenden Eintrag mit dem *Key Strasse*, wird eine neue Liste angelegt und in jedem Fall lediglich der aktuelle *Node* eingefügt.

Im nächsten Schritt wird nun zu jedem *Node* der erstellen *streetMap* über die Vorgänger-/Nachfolger- Beziehung diejenigen *Nodes* aus der *NodeMap* geholt, die zu der jeweiligen *Strasse* gehören (siehe Code 6).

[14] siehe Abschnitt 3.1
[15] gebraucht werden nur Geocodes vom Typ der Straßenknoten (siehe Abschnitt **??**)

```
13 List<DDGNodeList> all = new ArrayList<DDGNodeList>();
14 while ((next = findNachfolger(now)) != null && !kreis) {
15    if (!street.contains(next))
16       kreis = true;
17    street.remove(next);
18    list.addLast(next);
19    now = next;
20 }
21 kreis = false;
22 now = first;
23
24 while ((next = findVorgaenger(now)) != null && !kreis) {
25    if (!street.contains(next))
26       kreis = true;
27    street.remove(next);
28    list.addFirst(next);
29    now = next;
30 }
31 kreis = false;
32 all.add(list);
```

Java-Code 6: `DDGMapTool.generateStreets`

Die beiden *while*-Schleifen holen über die Methoden `findNachfolger` und `findVorgaenger` alle *Nodes* aus der *NodeMap* und fügen diese einer neuen Liste hinzu. Anhand dieser können anschließend die *Nodes* und *Ways* zueinander in die MySQL Datenbank eingefügt werden. Hierbei darf nicht vergessen werden, zu jedem Weg auch die *Way/Node* Relation in einer Tabelle zu vermerken, da die *Nodes* ansonsten anschließend nicht mehr den *Ways* zugeordnet werden können oder umgekehrt.

Es hat sich als äußerst ineffizient erwiesen, für jeden *Node* eine *Insert-Query* an die Datenbank zu schicken, da jede Anfrage Zeit benötigt und folglich bei mehr als 55.0000 Queries die Dauer des Importierens unnötig lang sein kann. Daher werden die Informationen in `StringBuffer` gesammelt und erst dann abgeschickt, wenn diese eine vorher festgelegte Maximallänge überschreiten.

3.4.2. OSM parsen und importieren

StAX als API zum Parsen der OSM-Datei zu verwenden ist für den Import der OSM-Daten optimal, da hier sehr große Datenmengen[16] verarbeitet und in die MySQL-Datenbank importiert werden müssen. Die Klasse `OSM2MySQL` implementiert diesen Parser und die Funktionalitäten zum Einfügen der Daten in die Datenbank nach dem in Abbildung 3.8 dargestellten Schema.

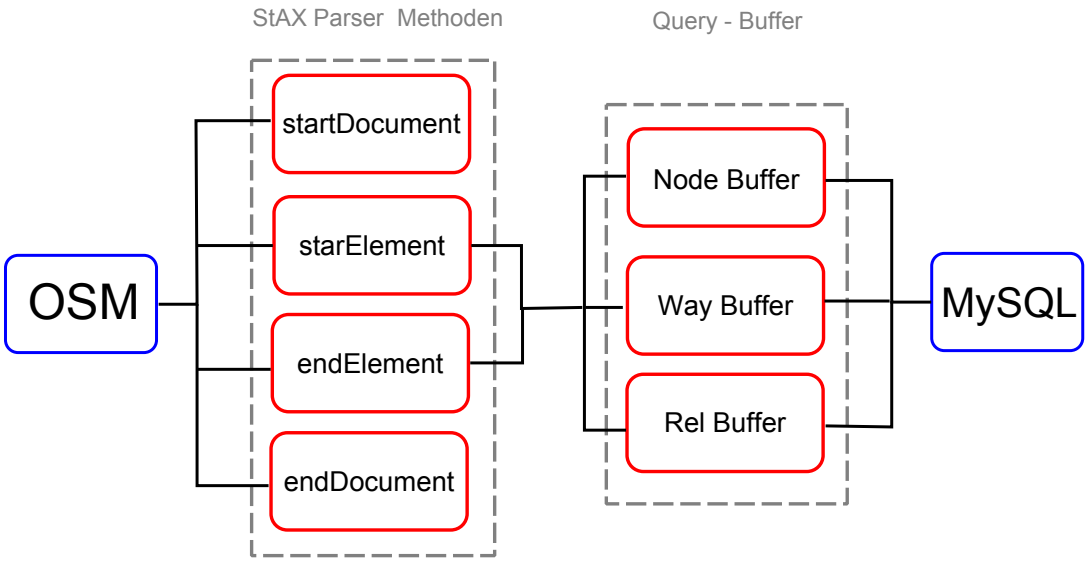

Abbildung 3.8: Schema der `OSM2MySQL` Klasse

Die Klasse `OSM2MySQL`

Auch diese Klasse erbt von `XML2MySQL` und implementiert die drei abstrakten Methoden. Über `initParser` wird, wie in Code 7 dargestellt, der StAX-Parser (hier `reader`) erzeugt, mithilfe dessen über `importData` die OSM-Daten geparst und in die MySQL-Datenbank importiert werden.

```
1 protected void initParser() {
2    FileInputStream fis = new FileInputStream(file);
3    XMLInputFactory factory = XMLInputFactory.newInstance();
```

[16] die germany.osm enthält rund 30 Mio. Einträge

```
4     factory.setProperty("javax.xml.stream.isValidating",
          Boolean.FALSE);
5     this.reader = (XMLStreamReader) factory.
          createXMLStreamReader(fis);
6 }
```

Java-Code 7: `OSM2MySQL.initParser`

Es kann davon ausgegangen werden, dass die XML-Files wohlgeformt sind, da diese in der Regel direkt von dem OSM-Server generiert werden, sodass, wie in Zeile 4 zu sehen, die Validierung des Parsers ausgeschaltet und zugleich etwas Performance gewonnen werden kann. Die Datei wird sequentiell von dem StAX-Parser durchlaufen. Dieser wirft bestimmte Stream-Ereignisse, auf die entsprechend reagiert werden kann, wie in Code 8 und 8 dargestellt ist.

```
1 createNodeTable();
2 createWayTable();
3 createNWRelTable();
```

Java-Code 8: `OSM2MySQL.importData`

Auch diese Klasse erstellt die nötigen Tabellen autonom, wie in den Zeilen 1 bis 3 dargestellt ist. Der Parser `reader` durchläuft in einer `while`-Schleife über `next` die XML und gibt je nach Ereignis gewisse Konstanten zurück. Die beiden Wichtigsten sind `StartElement` und `EndElement`, die immer dann zurückgegeben werden, wenn sich der Parser am Anfang bzw. am Ende eines Elements befindet. In den dadurch aufgerufenen Methoden `startElement` und `endElement` werden dann schließlich die *Nodes*, *Ways* und deren Beziehungen untereinander in den `StringBuffern` gesammelt und schließlich „paketweise“ an die Datenbank geschickt.

```
4  while (reader.hasNext()) {
5     switch (reader.next()) {
6        case XMLStreamConstants.START_DOCUMENT
7           startDocument();
8           break;
9        case XMLStreamConstants.START_ELEMENT:
10          startElement(reader)
11          break;
12       case XMLStreamConstants.END_ELEMENT:
13          endElement(reader);
```

```
14              break;
15          case XMLStreamConstants.END_DOCUMENT:
16              endDocument();
17              break;
18      }
19  }
```

Java-Code 9: `OSM2MySQL.importData`

Die *Nodes* werden, da ihre Informationen ausschließlich als Attribute vorkommen, direkt in der `startElement` Methode verarbeitet und in den Buffer eingefügt.

```
1  if (localName.equals("node")) {
2      for (int i = 0; i < attributeCount; i++) {
3          attributeLocalName = reader.getAttributeLocalName(i);
4          if (attributeLocalName.equals("id"))
5            node.setID(reader.getAttributeValue(i));
6
7          if (attributeLocalName.equals("lat"))
8            node.setLat(reader.getAttributeValue(i));
9
10         if (attributeLocalName.equals("lon"))
11           node.setLon(reader.getAttributeValue(i));
12     }
13 }
```

Java-Code 10: `OSM2MySQL.startElement`

Damit nicht alle rund 30 Mio. *Nodes* importiert werden, sollte bereits beim Parsen ein Filter dafür sorgen, dass alle unnötigen Einträge herausgefiltert und somit Performance für die spätere Optimierung gewonnen werden kann. Allerdings besitzen die *Nodes* zu wenige Informationen, um entscheiden zu können, welche später auch wirklich benötigt werden. Daher wird eine temporär angelegte Spalte in der Tabelle der *OSM-Nodes* verwendet, die später die brauchbaren *Nodes* mit einer 1 markiert.

```
14 if (sb_node.length() != NODE_QUERY.length())
15     sb_node.append(",");
16
17 sb_node.append("(" + node.getSQLString() + ","
18         + OSM_NODE_USED_DEFAULT + ")");
```

```
19
20 if (sb_node.length() > MAX_QUERY_LENGTH)
21    sendQuery(sb_node.toString(), 1);
```

Java-Code 11: `OSM2MySQL.startElement`

Folglich müssen dennoch vorerst *alle Nodes* importiert werden. Erst wenn der Parser im Dokument bei den *Ways* angelangt ist, können anhand derer *Node*-Listen diejenigen *Nodes* markiert werden, die in diesen aufgelistet sind. Ein Filter *filtert* bereits hier alle Straßen heraus, die nicht zu einer Autobahn oder Bundesstraße gehören, denn nur solche befinden sich in der EGT. Hierzu werden nur alle *Ways* vom Typ *motorway*, *motorway_link* und *primary*, *primary_link* an den Buffer weitergereicht. Die *_link* Angabe deutet hier lediglich auf eine Anschlussstelle hin.

Da sich nicht alle wichtigen Informationen der *Ways* in den Attributen befinden, sondern in weiteren Knoten, müssen diese zunächst vollständig gesammelt und erst am Ende eines Elements, sprich in der Methode *endElement*, in den Buffer eingefügt werden, sofern dieser dem gewünschten Typ entspricht. Sobald der Parser also auf ein Knoten namens „Way“ zeigt, muss in einer Variablen (hier *endOfWay*) vermerkt werden, ob man sich noch innerhalb eines *Ways* befindet, damit anschließend darüber die zu diesem *Way* benötigten Informationen selektiert werden können.

```
22 if (localName.equals("way")) {
23    endOfWay = false;
24    for (int i = 0; i < attributeCount; i++) {
25       attributeLocalName = reader.getAttributeLocalName(i);
26       if (attributeLocalName.equals("id")) {
27          way.setID(reader.getAttributeValue(i));
28      break;
29       }
30    }
31 }
```

Java-Code 12: `OSM2MySQL.startElement`

Falls sich der Parser innerhalb eines *Way*-Eintrags befindet, also zu diesem bisher kein schließendes Tag-Element verarbeitet wurde und demnach die Variable `endOfWay` immer noch `false` ist, kann geschaut werden, ob das Aktuelle Element, auf dem sich der Parser derzeit befindet, vom Typ `tag` ist. Ist dies der Fall, können nun die zu dem Way wichtigen Informationen zwischengespeichert werden.

```
32 if (attributeLocalName.equals("v")) {
33       if (lastTagName.equals("name"))
34             way.setName(reader.getAttributeValue(i));
35
36          if (lastTagName.equals("highway"))
37             way.setType(reader.getAttributeValue(i));
38
39          if (lastTagName.equals("ref"))
40            way.setRef(reader.getAttributeValue(i));
41
42          if (lastTagName.equals("int_ref"))
43            way.setIntRef(reader.getAttributeValue(i));
44 }
```

Java-Code 13: `OSM2MySQL.startElement`

Der nachfolgende Code demonstriert, wie die Attribut-Informationen aus der XML und in dem *Way*-Objekt zwischengespeichert werden.

```
45 for (int i = 0; i < attributeCount; i++) {
46    attributeLocalName = reader.getAttributeLocalName(i);
47    if (attributeLocalName.equals("k")) {
48    lastTagName = reader.getAttributeValue(i);
49 }
```

Java-Code 14: `OSM2MySQL.startElement`

Ist das Ende des Dokuments erreicht, sorgt die Methode `endDocument()` dafür, dass die in den `Buffern` verbliebenen Daten ebenfalls an die Datenbank geschickt werden. Zudem werden über die SQL-Query

```
1 DELETE FROM osm_nodes WHERE used!=1
```

alle nicht genutzten *Nodes* gelöscht.

4. Umsetzung und Visualisierung des Optimierungsalgorithmus

In Abschnitt 1.3 wurde bereits das Ziel der vorliegenden Arbeit näherungsweise erläutert. Nach dem jetzigen Wissensstand kann dieses detaillierter beschrieben werden. Die Daten der EGT werden von Frau Kunze im Rahmen ihrer Promotion u.a. zur Visualisierung der Verkehrsdichte verwendet. In Abbildung 4.1 ist hierzu ein Ausschnitt ihrer Visualisierung von München dargestellt.

Abbildung 4.1: Visualisierung der EGT in Verbindung von Verkehrsdichtedaten am Beispiel von München

Anhand von Verkehrsdichte-Daten, die den Geocodes in der EGT zugewiesen werden, können in Teilabschnitte unterteilte Straßen entsprechend eingefärbt und über einen gewissen Zeitraum simuliert werden[17]. Allerdings sind die Straßenverläufe, wie hier gut zu erkennen ist, aufgrund fehlender *Nodes* sehr ungenau. Bereits in Abbildung 1.1 wurde diese Problematik dargestellt, die in Abbildung 4.2 anhand eines weiteren Beispiels erneut aufgenommen wird.

Das Beispiel zeigt einen Teilabschnitt der Autobahn 'A31'. Hier ist sehr gut zu erkennen,

[17] `http://project.informatik.uni-osnabrueck.de/dlangfel/deuanders/`

Abbildung 4.2: Grobe Darstellung des anhand der EGT generierten Straßenverlaufs der 'A31'.

dass der *wahre* Straßenverlauf (hier gelbe Linie) stark von dem aus den Daten der DDG generierten abweicht. Gewünscht ist leztendlich der in Abbildung 4.3 gezeigte Verlauf, der anhand der Daten von OpenStreetMap generiert werden soll.

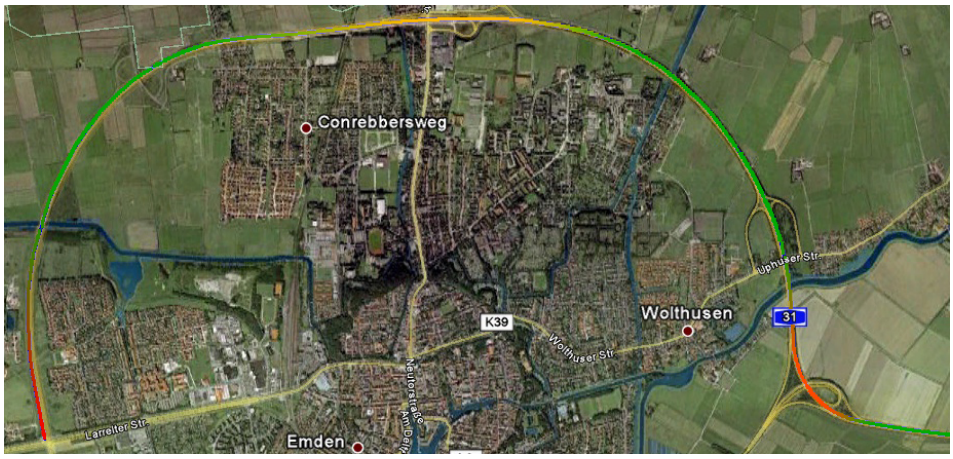

Abbildung 4.3: Genauere Darstellung der 'A31' durch eine Optimierung anhand der OSM-Daten

4.1. Ziel der Optimierung

Die Grundlagen zur Optimierung sind in Abschnitt 3 geschaffen worden. Sämtliche Daten der DDG sowie von OpenStreetMap liegen also nun für die weitere Verarbeitung in der in den Abbildungen 3.5 und 3.4 gezeigten Datenbankstruktur vor. Das Ziel des Algorithmus ist letztendlich die in Abbildung 4.4 gezeigte Relation zwischen diesen Daten herzustellen.

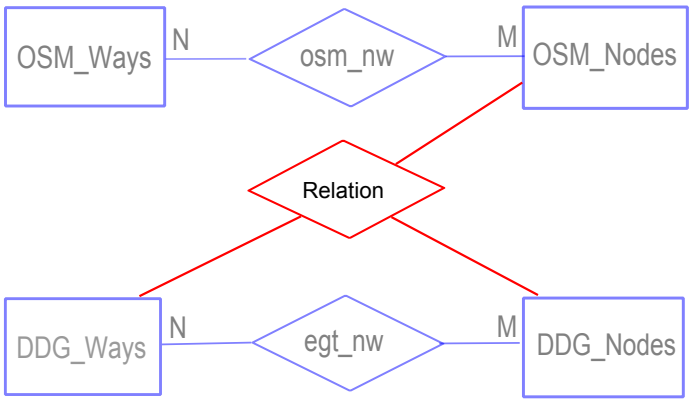

Abbildung 4.4: **Ziel der Arbeit:** Ziel ist es, eine Relation zwischen der DDG und OSM herzustellen.

4.2. Der Algorithmus

Der Algorithmus muss also anhand der Datenbestände die in Abbildung 4.4 gezeigte Relation herstellen. Dies wird durch zwei Tabellen in der Datenbank erreicht, die zum einen eine Beziehung zwischen den DDG-*Nodes* und OSM-*Nodes* und zum anderen eine Beziehung zwischen den DDG-*Ways* und OSM-*Ways* herstellen. Dadurch ist es möglich, anhand der Angabe eines EGT-*Ways* alle OSM-*Nodes* zu erhalten, die den besseren Verlauf der Straßen wiederspiegeln. Anhand der *Node* zu *Node* Relation können diesen OSM-*Nodes* dann wiederum die späteren Verkehrsinformationen der EGT zugeordnet und visualisiert werden. Es ergibt sich also im Wesentlichen das in Abbildung 4.5 gezeigte ER-Diagramm.

Um dies zu erreichen, muss der Algorithmus folglich in der Lage sein, die im Folgenden zusammengefassten Schritte, anhand der sich in der Datenbank befindlichen Informationen, durchzuführen:

1. Sammle alle OSM-Nodes der Autobahnen und Bundesstraßen, die in der *ddg_ways* Tabelle eingetragen sind.
2. Stelle eine Beziehung zwischen den einzelnen EGT-*Nodes* und den OSM-*Nodes* her.
3. Stelle eine Beziehung zwischen den EGT-*Ways* und den OSM-*Nodes* her.

Ist der Algorithmus durchgelaufen sollten die in Abbildung 4.5 rot dargestellten Tabellen mit Daten gefüllt sein.

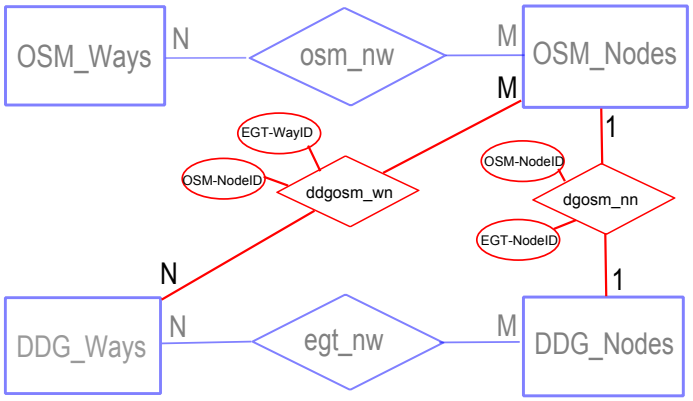

Abbildung 4.5: Das Ziel der Arbeit ist es, die beiden in rot dargestellten Tabellen zu erzeugen.

4.2.1. Problem als Graphen formulieren

Die Implementation des zweiten Schritts erfordert die Kenntnisse aus der in Abschnitt 2.3 beschriebenen Vorgehensweise der Graphentheorie. Hier liegt zunächst die Aufgabe darin, das vorhandene Teilproblem als Graphen zu formulieren. Hierzu müssen lediglich die *Vertices* und *Edges* definiert werden. Da dies nicht ganz trivial ist, bemüht sich dieser Abschnitt um ein besseres Verständnis zur Lösung der Problematik.

Im Verlauf der Implementation erwies es sich als nützlich die Anfangs- und End- *Nodes* eines *Ways* als *Vertex*[18] zu definieren. Den *Edges* wird keine größere Bedeutung zugeordnet, da diese in der Regel nur benötigt werden, um eventuelle Evaluierungsfunktionen (siehe Abschnitt 2.3.2) auf ihnen anwenden zu können, die hier nicht verwendet werden. Abbildung 4.7 soll in etwa verdeutlichen, wie anhand dieser Definition die *Nodes* der in Abbildung 4.6 gezeigten Straße aus den in Abbildung 4.8 aufgeführten Straßen über

[18] zu diesem Zweck ist die Klasse `Vertex` geschrieben worden.

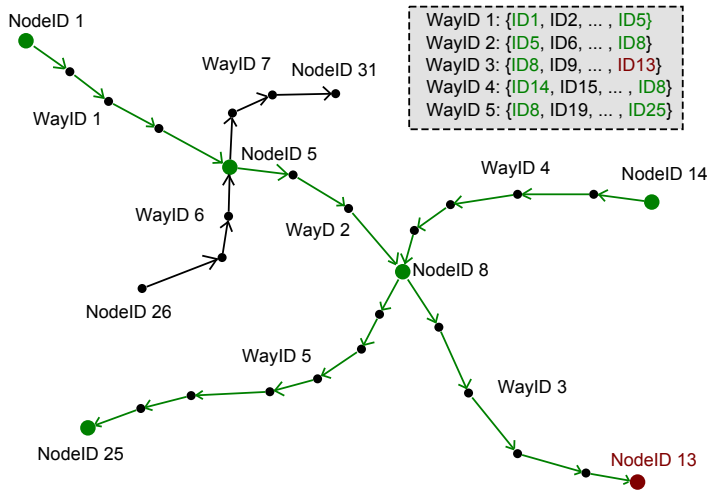

Abbildung 4.6: Beispielsszenario zur Veranschaulichung des Tiefenalgorithmus

einen Graphenalgorithmus abgelaufen werden können. Gesucht sind in diesem Beispiel die in den *Ways* 1 bis 3 enthaltenen *Nodes*, wobei die Suche bei *Node* 1 beginnt und in *Node* 13 endet.

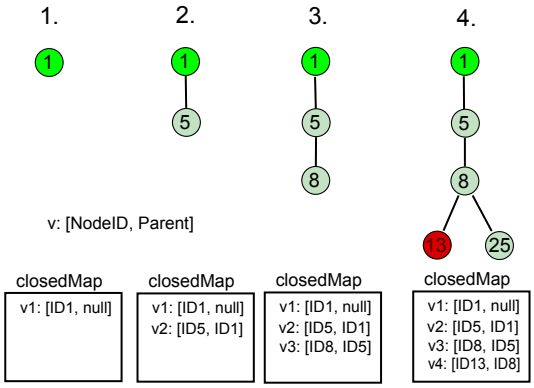

Abbildung 4.7: Vorgehensweise des Tiefenalgorithmus anhand des in Abbildung 4.6 dargestellten Beispiels

Es wird der Tiefensuchalgorithmus (DFS) verwendet. Dieser expandiert als erstes *Node* 1. Lediglich *Way* 1 enthält diesen Knoten; daher wird nur *Node* 5 als Nachfolger zurückgegeben, da sich dieser in Suchrichtung (hier Pfadrichtung) – also am Ende der *Node*-Liste dieses *Ways* – befindet. In diesem muss zudem *Node* 1 als Vorgänger (*Parent*) vermerkt

werden, damit am Ende der Suche der Pfad rekonstruiert werden kann. Die anderen *Nodes* erhalten wiederum den *Node* als Vorgänger, der sie expandiert hat. Nun wird *Node* 5 expandiert, woraus sich der Nachfolger *Node* 8 ergibt. Zwar ist dieser ebenfalls in den *Ways* 6 und 7 enthalten, diese sind allerdings von einem anderen Typ. Schließlich wird *Node* 8 expandiert, welches neben *Node* 25 – den Ziel-*Node* 13 als Nachfolger enthält. *Node* 14 wird nicht expandiert, da sich dieser ebenfalls nicht in Richtung der Suche befindet. Alle expandierten *Nodes* werden während dieser Suche in einer `closedMap` gespeichert, wie in Abbildung 4.7 dargestellt ist.

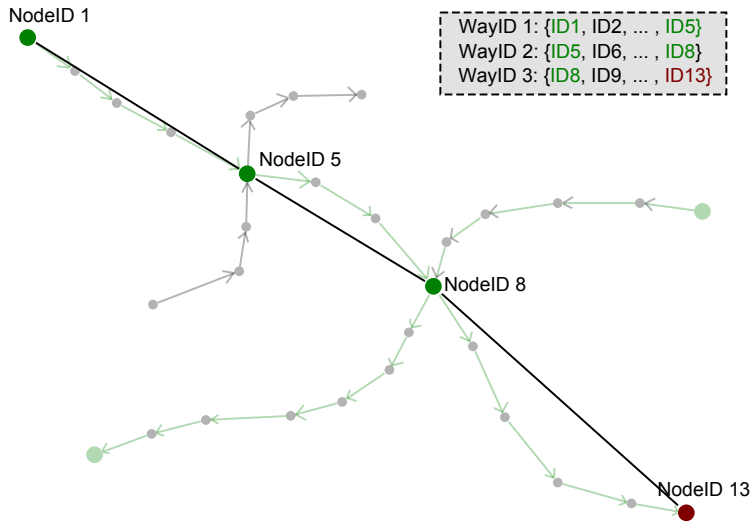

Abbildung 4.8: Ergebnis der Tiefensuche

Letztendlich ergibt sich aus den Vorgänger-Beziehungen und den in der `closedMap` gesammelten *Nodes* der in Abbildung 4.8 hervorgehobene Straßenverlauf. Die noch fehlenden Zwischen-*Nodes* können anschließend in einem weiteren Verfahren hinzugefügt werden.

4.3. Die Klasse `DDGOptimizer`

Für die Optimierung ist letztendlich die Klasse `DDGOptimizer` geschrieben worden, die über die `public` deklarierte Methode `optimize()` den Algorithmus zur Optimierung startet. Auch sie enthält den bereits in Abschnitt 3.4 dargestellten `MySQLHandler`, da Anfragen an die Datenbank gestellt werden müssen. In dem nachfolgenden Code wird diese Methode dargestellt.

```
1 public void optimize() {
2    this.ddgStreets = getListofDDGStreets();
3    this.osmNodeLightMap = generateLightMap();
4    generateRelations();
5 }
```

Java-Code 15: DDGOptimizer.optimize

Es ist zu erkennen, dass hier die zuvor erwähnten Schritte zur Optimierung durchführt werden. Allerdings stellt erst der dritte Aufruf in Zeile 4 die gewünschte Relationen mit den Tabellen in der Datenbank her. Beide vorab aufgerufenen Methoden bereiten dies entsprechend vor.

Zunächst ist eine vorausgewählte Liste von Referenznummern für je eine Autobahn bzw. Bundesstraße der DDG zu ermitteln. Hierzu werdeb der bereits in der Datenbank angelegten Tabelle *egt_ways* mithilfe einer SQL-Query sämtliche Straßennamen entnommen. Diese Query sieht wie folgt aus:

```
1 SELECT DISTINCT(w.Strasse)
2 FROM egt_ways w
3 ORDER BY Strasse ASC
```

Für diesen Zweck ist eine eigene Methode `getListOfDDGStreets()` implementiert, die eine nach dem Alphabet sortierte Liste mit den Kennungen der Autobahnen bzw. Bundesstraßen (Referenznummern) liefert.

4.3.1. Generieren der OSMNodeLightMap

Das in Abschnitt 4.2.1 beschriebene Verfahren wird noch vor dem zweiten Schritt der Optimierung durchgeführt. Hierzu benötigt die Methode `generateLightMap()` – die im ersten Schritt generierte Liste der DDG-Streets. Der Begriff „`OSMNodeLightMap`“ wird hier verwendet, da es sich um eine zunächst „leichte“ bzw. „unvollständige“ Map mit *Nodes* der OpenStreetMap handelt. Die grundlegende Funktionsweise zur Erstellung dieser ist in Abbildung 4.9 dargestellt.

Die Methode durchläuft im Grunde genommen folgende sechs Schritte für jeden Eintrag der DDG-Streets Liste.

1. Hole die Liste aller DDG-*Nodes* zu dieser Referenznummer, sortiert nach *Lfdnr* und betrachte den ersten.

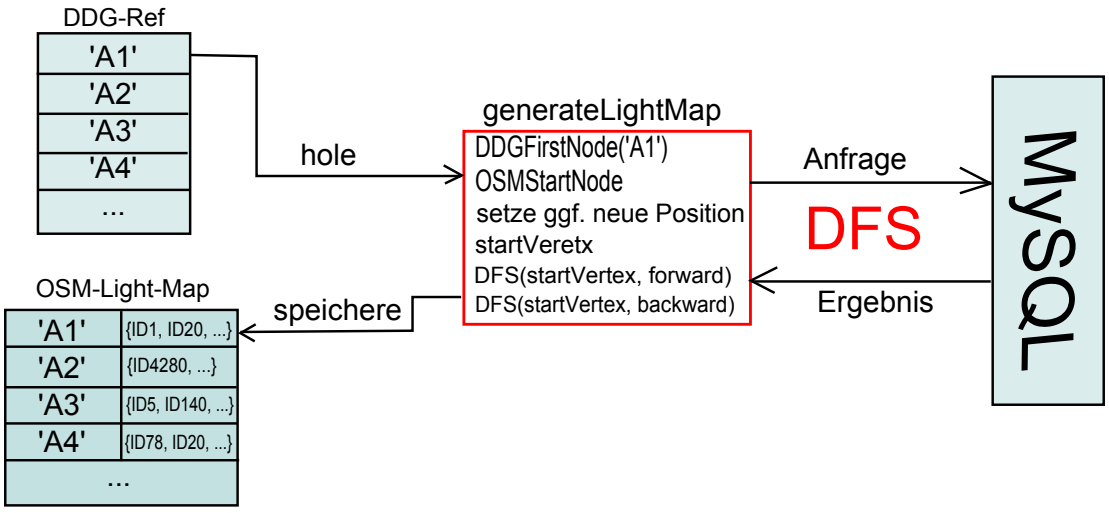

Abbildung 4.9: Schema zur Methode `generateLightMap()`

2. Suche einen dazu passenden OSM-*Node* anhand des geringsten Abstands und der Referenznummer.
3. Falls sich dieser Node nicht am Ende der zu diesem *Way* gehörenden *Node*-Liste befindet, hole und betrachte im weiteren Verlauf nur den letzten *Node*.
4. Erstelle aus diesem den Start-*Vertex*.
5. Führe DFS vorwärts durch und speichere *Nodes* in der `OSMNodeLightMap`.
6. Führe DFS rückwärts durch und speichere *Nodes* in der `OSMNodeLightMap`.

Zu Beginn werden alle notwendigen Listen und Variablen erstellt, die im weiteren Verlauf benötigt werden. In Zeile 1–2 wird eine neue `Map` instanziiert, die letztendlich die `OSMNodeLightMap` darstellt. Die in Zeile 4 deklarierte `closedMap` wurde bereits in Abschnitt 4.2.1 erwähnt und dient zur Sammlung der aktuell abgelaufenen *Nodes*. Über diese können zudem Zyklen vermieden werden, indem nur *Nodes* expandiert werden, die sich nicht in dieser *Map* befinden. In den Zeilen 6 und 7 werden `Strings` deklariert, die sich die Referenznummer aus der DDG-Streets Liste merken sollen. Hierbei ist es notwendig, die Referenznummer der DDG und der OSM zu unterscheiden, da die Konvention zur Darstellung jeweils anders ist. Während beispielsweise die Referenznummer der Autobahn 'A1' in der DDG eben genau dieser Schreibweise entspricht, ist es in OpenStreetMap vorgeschrieben, ein Leerzeichen zwischen der Kennung 'A' und der Zahl '1' zu setzten. Die in Zeile 9 und 10 deklarierten Objekte werden für die spätere Zuweisung des OSM-Start-*Nodes*, sowie des Start-*Vertex* benötigt, an der die folgende Tiefensuche

startet.

```
1  Map<String, LinkedList<OSMNode>> nodeMap;
2  nodeMap = new HashMap<String, LinkedList<OSMNode>>();
3
4  Map<Integer, Vertex> closedMap;
5
6  String ddgRef, osmRef;
7  String osmIntRef = "-1";
8
9  OSMNode startNode;
10 Vertex startVertex;
```

Java-Code 16: DDGOptimizer.generateLightMap: Deklaration der Variablen und Listen

Anschließend werden in einer Schleife, die alle Elemente der DDG-Streets Liste durchläuft, zunächst die jeweiligen Referenznummern ermittelt und über die in Zeile 13 zu sehende Methode `getOSMStartNode(ddgRef)` ein passender *Node* aus der OSM geholt.

```
11 ddgRef = ddgStreets.get(idx);
12 osmRef = makeOSMRef(ddgRef);
13 startNode = getOSMStartNode(ddgRef);
```

Java-Code 17: DDGOptimizer.generateLightMap: OSM-`startNode` besorgen.

Diese besorgt sich, wie in Schritt 1 und 2 beschrieben, den ersten *Node* aus der DDG-*Node* Liste zu der momentanen Referenznummer. Anhand dessen wird ein OSM-*Node* aus der Datenbank ermittelt, der ebenfalls auf diese Referenz passt und den geringsten Abstand zu diesem aufweist. Hierzu dient die Methode `getMinDistOSMNode(ref,lat,lon)`, der eine Referenznummer, sowie die geografischen Koordinaten des DDG-*Nodes* übergeben werden.

```
14 LinkedList<Integer> wayList;
15 wayList = getOSMWayIDsForNode(startNode, osmRef, -1);
16 int wayid = wayList.getFirst();
```

Java-Code 18: DDGOptimizer.generateLightMap: *Ways* zusammen suchen, die diesen `startNode` enthalten.

Weiterhin werden ebenfalls in der Schleife die Schritte 3–6 durchgeführt. Sofern ein `startNode` gefunden wurde, müssen zu diesem zunächst alle *Ways* ermittelt werden, die diesen enthalten. Da in den hier vorkommenden Straßen in der Regel immer nur ein *Way* gefunden wird, kann je der erste verwendet werden (siehe Zeile 14–16). Allerdings sollte dennoch in Betracht gezogen werden, auch den anderen Fall abzufangen und eine dementsprechende Entscheidung zu treffen.

```
17 LinkedList<OSMNode> nodeList;
18 nodeList =  = getOSMNodesForWayID(wayid);
19 int pos = getPosition(startNode, nodeList);
20
21 if ((pos != nodeList.size() - 1) && (pos != 0))
22    startNode = nodeList.getLast();
23
24 startVertex = new Vertex();
25 startVertex.setNode(startNode);
```

Java-Code 19: DDGOptimizer.generateLightMap: ggf. neuen `startNode` setzen.

Des Weiteren muss geschaut werden, an welcher Position sich der `startNode` in der Liste befindet (Zeilen 17–19). Sollte diese nicht am Ende der Liste sein, wird der sich dort befindende *Node* als neuer `startNode` gesetzt (Zeilen 21–22). Der Grund, warum überhaupt nach der Position in der Liste geschaut wird ist der, dass wie in Abschnitt 4.2.1 beschrieben, lediglich die Nodes als Nachfolger geliefert werden, die sich je nach Suchrichtung am Anfang oder am Ende der jeweiligen *Node*-Listen befinden. Hierbei ist willkürlich gewählt worden, dass sich der `startNode` am Ende der Liste befinden soll, da ohnehin eine Suche in beide Richtungen stattfindet.

Letztendlich wird der für die Aufrufe des nachfolgenden Graphenalgorithmus *Vertex* `startVertex` erstellt, dem der `startNode` übergeben wird (Zeilen 24–25).

```
26 closedMap = new HashMap<Integer, Vertex>();
27 DFS(startVertex,osmRef,osmIntRef,closedMap,DIR_FORWARD);
28 addOSMNodes2DDGList(ddgRef nodeMap,closedMap,DIR_FORWARD);
29
30 closedMap = new HashMap<Integer, Vertex>();
31 DFS(startVertex, osmRef,osmIntRef,closedMap,DIR_BACKWARD);
32 ddOSMNodes2DDGList(ddgRef,nodeMap,closedMap,DIR_BACKWARD);
```

Java-Code 20: DDGOptimizer.generateLightMap: DFS in beide Richtungen aufrufen und die gefundenen *Nodes* in die `nodeMap` ablegen.

Anschließend sucht sich der Graphenalgorithmus DFS alle OSM-*Nodes* zu der aktuellen Referenznummer, am `startVertex` beginnend, zusammen und speichert diese in der `closedMap`. Über die Methode `ddOSMNodes2DDGList(ddgRef, nodeMap, closedMap, direction)` werden schließlich diejenigen *Nodes* in die `nodeMap` hinzugefügt, die auch wirklich gebraucht werden.

Die Implementation von `DFS`

Als Graphenalgorithmus wird hier die in Abschnitt 2.3.1 und bereits in Abschnitt 4.2.1 demonstrierte Tiefensuche verwendet, da man davon ausgehen kann, dass sich das „Ziel“ in der Regel ziemlich weit unten in dem Graphen befindet. Die Methode dazu heißt `DFS(startVertex, osmRef, osmIntRef, closedMap, direction)`.

```
1 private void DFS(Vertex startVertex, String osmRef,
2          String osmIntRef, Map<Integer, Vertex> closedMap,
3          int direction) throws SQLException
```

Java-Code 21: `DDGOptimizer.DFS`: Methodenkopf von DFS

Sie benötigt den `startVertex`, an dem die Suche beginnt, die nationale und internationale Referenznummer `osmRef & osmIntRef` – anhand der alle *Ways* anderer Referenz ausgeschlossen werden können – die bereits erwähnte `closedMap` und schließlich die Richtung `direction` in der gesucht werden soll. Es wird sich im weiteren Verlauf herausstellen, dass kein wirkliches „Ziel “ definiert werden kann. Ein Lösungsansatz dazu wird aber etwas später erläutert.

```
1 Stack<Vertex> stack = new Stack<Vertex>();
2 stack.push(startVertex);
3 Vertex vertex;
```

Java-Code 22: `DDGOptimizer.DFS`

Bekannterweise benötigt die Methode einen `Stack` auf dem alle expandierten Nachfolger abgelegt und später zur weiteren Verarbeitung wieder herausgeholt werden (Zeilen 1). Auf diesem wird zu Beginn der übergebene `startVertex` abgelegt (Zeile 2), da

die Suche bei diesem beginnt. In einer wile-Schleife werden anschließend alle in Code 23 dargestellten und bereits in Abschnitt 2.3.1 aufgeführten Anweisungen zur Tiefensuche durchgeführt, bis der eben genannte Stack leer ist.

```
4  vertex = stack.pop();
5  closedMap.put(vertex.getNode().getID(), vertex);
6
7  LinkedList<Vertex> nachfolgerList;
8  nachfolgerList = expandVertex(vertex, osmRef, osmIntRef,
9                          closedMap, stack, direction);
10
11 if (nachfolgerList != null) {
12    for (Vertex v : nachfolgerList) {
13       v.lastVertex = vertex;
14       v.idx = v.lastVertex.idx + direction;
15       stack.push(v);
16    }
17 }
```

Java-Code 23: DDGOptimizer.DFS

Es wird also pro Schleifendurchlauf das oberste Element aus dem Stack geholt. Von diesem *Vertex* werden anschließend die Nachfolger über expandVertex(vertex, osmRef, osmIntRef, closedMap, stack, direction) ermittelt und wieder auf den Stack gelegt. Hinterher wird der Vertex in die closedMap aufgenommen, damit dieser über eventuelle Zyklen nicht noch einmal expandiert wird. Außerdem erhalten die expandierten *Vertices* den aktuellen *Vertex* als Vorgänger (Zeile 13). Über diese Beziehung kann abschließend der Verlauf von Ende bis Anfang zurückverfolgt werden.

In Abbildung 4.10 ist die Vorgehensweise der Methode expandVertex Dargestellt. Sie sendet anhand der übergebenen Parameter und je nach Ergebnis maximal drei Anfragen an die Datenbank (siehe Code 24,24, Zeile 5, 8, 11).

Zunächst erfolgt eine genaue Anfrage über getOSMWayIDsForNode in der nur die *Ways* zurückgegeben werden, die den in dem *Vertex* gespeicherten *Node* enthalten, die Referenznummer osmRef haben und nicht die ID des *Ways* besitzten, den der Vorgänger hat. Liefert diese Anfrage ein Ergebnis kann davon ausgegangen werden, dass die sich in dieser Liste befindenden *WayIDs* für die weitere Suche der Nachfolger in jedem Fall verwendet werden können. Sollte das Ergebnis null sein, kann der Grund dafür sein, dass

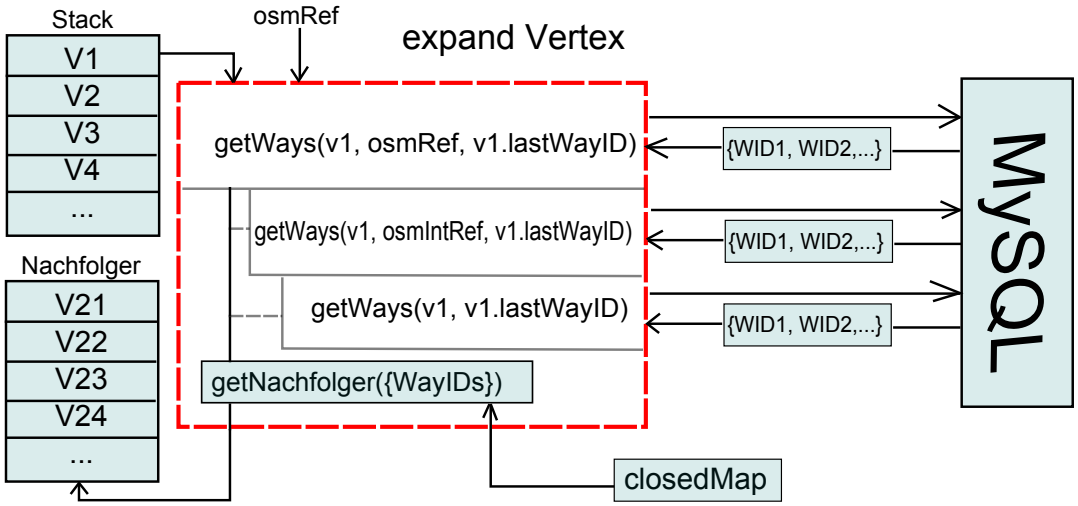

Abbildung 4.10: Vorgehensweise der Methode `expandVertex`

zu diesem *Way* keine Referenznummer in der Datenbank vorliegt, daher wird in einer zweiten Abfrage die nationale Referenznummer durch die Internationale `osmIntRef` ersetzt. Wenn auch diese `null` liefert, kann sich der aktuelle *Vertex* an einer Anschlussstelle befinden, an der nur *Ways* vom Typ *_link* angrenzen. In diesem Fall wird in der dritten Abfrage die Einschränkung etwas gelockert und die Referenznummer ganz weggelassen.

Liefert eine der Anfragen eine Liste mit *WayIDs* zurück, können anhand derer die Nachfolger bestimmt werden. Hierzu erinnere man sich an die in Abschnitt 4.2.1 gezeigte Abbildung 4.6. Wird vorwärts – also in Fahrtrichtung – gesucht, muss der letzte *Node* aus der zu dem *Way* gehörenden Liste als Nachfolger verwendet; andernfalls der erste. Dies übernimmt die Methode `getNextOSMNode` in Zeile 20. Hierbei wird die bereits öfter erwähnte laufende Nummer *Lfdnr* verwendet. Denn über diese kann auch in der Datenbank die Position eines *Nodes* in der Liste ermittelt werden.

```
1 LinkedList<Vertex> nachfolger = new LinkedList<Vertex>();
2 LinkedList<Integer> ways;
3 LinkedList<OSMNode> waynodes;
4
5 ways = getOSMWayIDsForNode(vertex.getNode(), osmRef, vertex.
     getLastWayID());
6
7 if (ways == null) {
```

```
8     ways = getOSMWayIDsForNode(vertex.getNode(), osmIntRef,
          vertex.getLastWayID());
9
10    if (ways == null) {
11      ways = getOSMWayIDsForNode(vertex.getNode(), "", vertex
            .getLastWayID());
12
13     if (ways == null)
14         return null;
15  }
```

Java-Code 24: DDGOptimizer.expandVertex

Befinden sich die so ermittelten Nachfolger weder in der `closedMap` noch auf dem `Stack` (hier `openStack`), können diese von der Methode zurückgeliefert werden (Zeile 27).

An dieser Stellte sollte vielleicht noch erwähnt werden, wie das „Ziel“ definiert ist. Im Allgemeinen lässt sich nicht ohne weiteres ein einzelner oder eine Menge an *Nodes* festmachen, an denen die Suche enden soll. Vielmehr ist hier die Idee, den Suchalgorithmus so lange laufen zu lassen, bis sich keine zu expandierenden *Nodes* mehr in dem `Stack` befinden. Um dennoch aus all diesen in der `closedMap` befindlichen *Nodes* denjenigen herauszufinden, an dem der Pfad zurückverfolgt werden soll, wird der in Zeile 14 zu sehende Index `idx` je nach Suchrichtung incrementiert oder decrementiert. Hierdurch kann der *Node* ermittelt werden, der sich am tiefsten in dem Graphen befindet und somit auch den längsten Pfad wiedergibt. Schließlich wird versucht, möglichst alle *Nodes* einer Straße zu finden.

```
16  Vertex v;
17  OSMNode n;
18  for (int wid : ways) {
19     waynodes = getOSMNodesForWayID(wid);
20     n = getNextOSMNode(waynodes, vertex.getNode().getID(),
          direction);
21     v = new Vertex(n, wid, vertex.getNode());
22
23     if (!closedMap.containsKey(v.getNode().getID()) && !
          openStack.contains(v.getNode().getID()) && (v.getNode
          ().getID() != vertex.getNode().getID()))
24        nachfolger.add(v);
```

```
25 }
26 return nachfolger;
```

Java-Code 25: `DDGOptimizer.expandVertex`

Zum Verständnis soll Abbildung 4.11 demonstrieren, wie letztendlich anhand der *Vertex*-Indices und der Vorgängerbeziehung, ein Pfad rekonstruiert werden kann. Aus der `closedMap` wird, wie schon erwähnt, der *Vertex* mit dem größten Index ausgewählt. In diesem Fall ist das der *Vertex* 42. In diesem ist *Vertex* 13 als Vorgänger vermerkt. Wird diese Relation soweit zurückverfolgt, bis schließlich `null`, also kein weiterer Vorgänger zu finden ist, können alle abgelaufenen *Nodes* der *Vertices* in eine Liste aufgenommen werden.

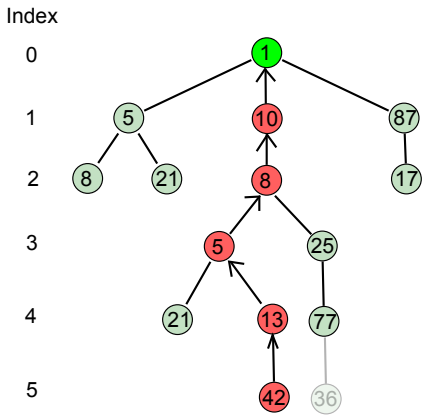

Abbildung 4.11: Inhalt der `closedMap` durch einen Graphen dargestellt

In den meisten Fällen reicht die resultierende Node-Liste der Vorwärtssuche (oder auch Rückwärtssuche) nicht aus. Im Gegenteil; es gibt nur wenige Ausnahmen, wie an dem bereits gezeigtem Beispiel der 'A31', das Ergebnis des DFS-Algorithmus in nur einer Richtung zum Erfolg führt. In der Regel befinden sich die `startNodes` in einem Teilbereich innerhalb der Straße. Daher wird der DFS-Algorithmus stets in beide Richtungen ausgeführt.

Die durch die DFS(backward) gefundenen Nodes müssen anschließend entsprechend vor den bereits in der DFS(forward) erstellen Liste angefügt werden. Diesen Schritt übernimmt die Methode `addOSMNodes2DDGList`. Letztendlich ergibt sich dadurch das in Abbildung 4.13 im Beispiel der 'A31' dargestellte Ergebnis.

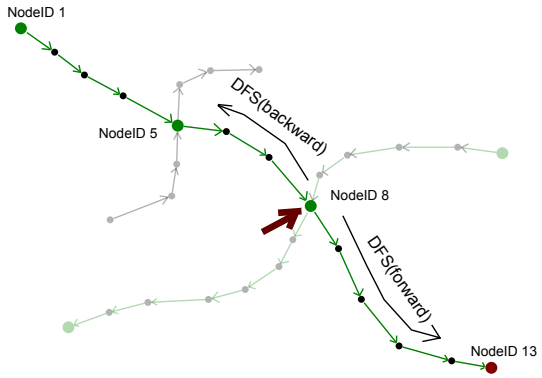

Abbildung 4.12: DFS muss immer in beiden Richtungen suchen.

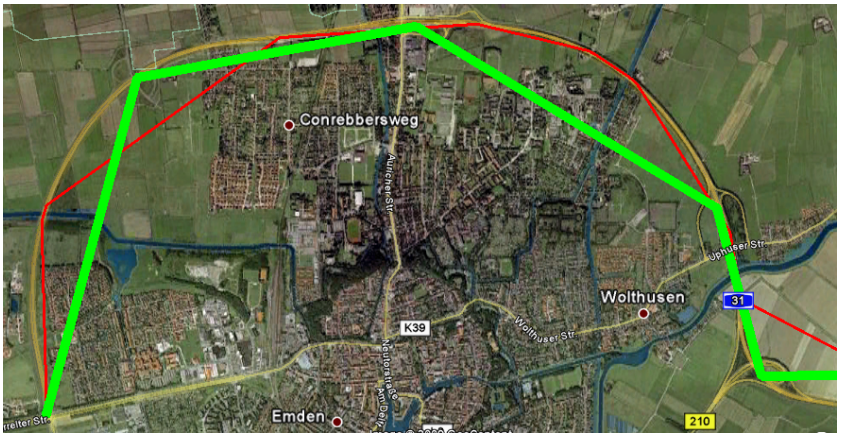

Abbildung 4.13: alte DDG Daten (grün), teile der neuen OSM-*Nodes* (rot)

4.3.2. Relationen zwischen den Daten der DDG und OSM herstellen

In diesem Abschnitt werden die in Abbildung 4.5 dargestellten Tabellen `ddosm_nn` und `ddgosm_wn` erzeugt (siehe Zeile 1–2). Diese stellen zum einen die DDG-*Nodes* und OSM-*Nodes* und zum anderen die DDG-*Ways* und OSM-*Nodes* zueinander in Beziehung.

```
1 createDDGOSMNodeRefTable();
2 createDDGOSMWayNodeRefTable();
```

Java-Code 26: `DDGOptimizer.generateRelations`:

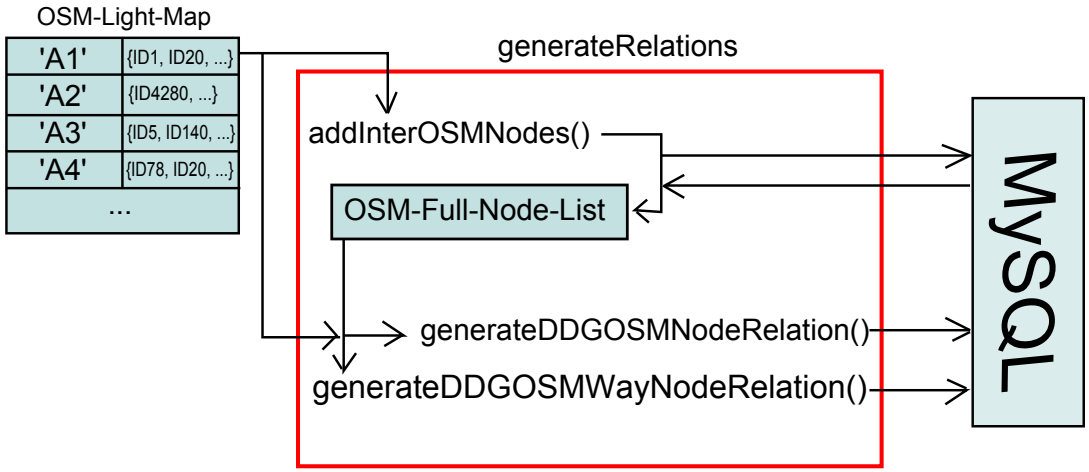

Abbildung 4.14: Funktionsweise der Methode `generateRelations`.

Wie der Abbildung 4.14 zu entnehmen ist, wird hierfür die zuvor erstellte `osmNodeLightMap` benötigt. Für jeden Eintrag dieser *Map* wird die enthaltene *Node*-Liste mit allen verbleibenden *Nodes* gefüllt, die während des DFS-Algorithmus außer Acht gelassen wurden. Die Implementation ist trivial – je zwei aufeinander folgende *Nodes* aus dieser Liste stellen aufgrund der Implementation von DFS Anfang und Ende eines *Ways* dar. Daher muss lediglich *der Way* gefunden, der eben diese beiden *Nodes* enthält ist zudem eindeutig. Diese Suchanfrage ist eindeutig und liefert stets nur einen *Way*. Alle zu diesem gefundenen *Way* gehörenden *Nodes* müssen anschließend in der richtigen Reihenfolge in die aktuelle Liste eingefügt werden. Dies erledigt die Methode `addInterNodes` in den Zeilen 3–5.

```
3 ddgNodeList = getDDGNodesForRef(makeDDGRef(osmRef));
4 fullOSMNodeList = new LinkedList<OSMNode>(osmNodeLightMap.
    get(osmRef));
5 addInterOSMNodes(fullOSMNodeList);
```

Java-Code 27: `DDGOptimizer.generateRelations`:

Anschließend stellen die Aufrufe der Methoden in Zeile 6 und 7 die für das Ziel dieser Arbeit wichtigen Relationen her.

```
6 generateDDGOSMNodeRelation(ddgNodeList, fullOSMNodeList);
7 generateDDGOSMWayNodeRelation(fullOSMNodeList, osmRef);
```

Java-Code 28: DDGOptimizer.generateRelations:

Die *Node* zu *Node* Relation wird hergestellt, indem zu jedem EGT-*Node*, der Abstand zu allen OSM-*Nodes* in der `osmFullNodeMap` berechnet wird. Die ID des OSM-*Nodes* mit dem geringsten Abstand zu dem aktuellen EGT-*Node* wird zusammen mit dessen ID in die Tabelle `ddgosm_nn` gespeichert. Diese Methode sorgt in den *meisten* Fällen für eine korrekte Zuordnung.

Nun müssen anschließend nur noch die OSM-*Nodes* zwischen dem ersten und dem letzten EGT-*Node* der aktuellen *WayID* in die Tabelle `ddgosm_wn` gespeichtert werden. Dies gelingt, in dem über die zuvor erstelle Beziehung die entsprechende Position der OSM-*Nodes* zu dem ersten und letzten EGT-Nodes ermittelt, und alle sich dazwischen befindlichen *Nodes* an die Datenbank gesendet werden (siehe Abbildung 4.15).

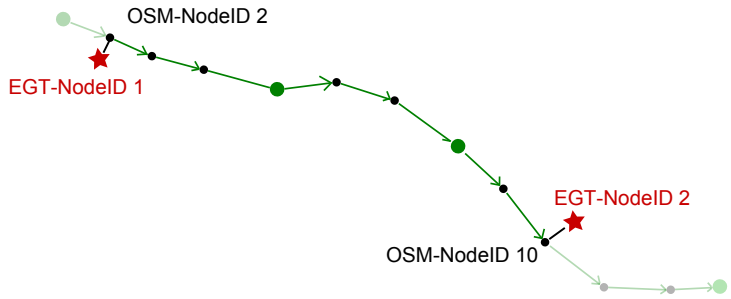

Abbildung 4.15: Darstellung der beiden Verfahren zur Erstellung der Relationen.

4.4. Formaler Ablauf

Hier ist abschließend der gesamte Ablauf des Optimierungsalgorithmus formal und vereinfacht aufgeführt:

1. Erstelle eine Liste mit allen DDG-Straßen (Referenznummern).
2. Nimm für jeden Eintrag der Liste die Referenz heraus.
   - i Suche über die Datenbank alle DDG-*Nodes*, die zu dieser Referenz gehören und nimm den ersten.
   - ii Finde einen passenden OSM-*Node*.
   - iii Starte DFS in beide Richtungen und füge alle *Nodes*, die zu dem Verlauf der Straße beitragen in die `OSMNodeLightMap<String, OSMNode>`.
3. Nimm für jeden Eintrag der `OSMNodeLightMap` die Referenz heraus.
   - Stelle DDG-*Node* – OSM-*Node* Relation her:
     - i Erstelle eine Liste mit den DDG-*Nodes*, die zu dieser Referenz gehören, sortiert nach *Lfdnr*.
     - ii Füge der zu der Referenz gehörenden Node-Liste *alle* OSM-*Nodes* der Straße hinzu.
     - iii Berechne für jeden EGT-*Node* aus der erstellten Liste den Abstand über die Koordinaten zu jedem OSM-*Node*.
     - iv Die OSM-*Nodes* mit dem geringsten Abstand zu je einem DDG-*Node* wird in die Tabelle `egtosm_nn` übernommen.
   - Stelle DDG-*Way* – OSM-*Node* Relation her:
     - i Erstelle eine Liste mit den DDG-*WayIDs*, die zu dieser Referenz gehören, sortiert nach *Lfdnr*.
     - ii Stelle anhand der DDG-*Node* – OSM-*Node* Relation und der Liste mit *allen* OSM-*Nodes* zu der Referenz die Beziehung DDG-*Way* – OSM-*Node* her und übernimm dies in die Tabelle `egtosm_wn`.

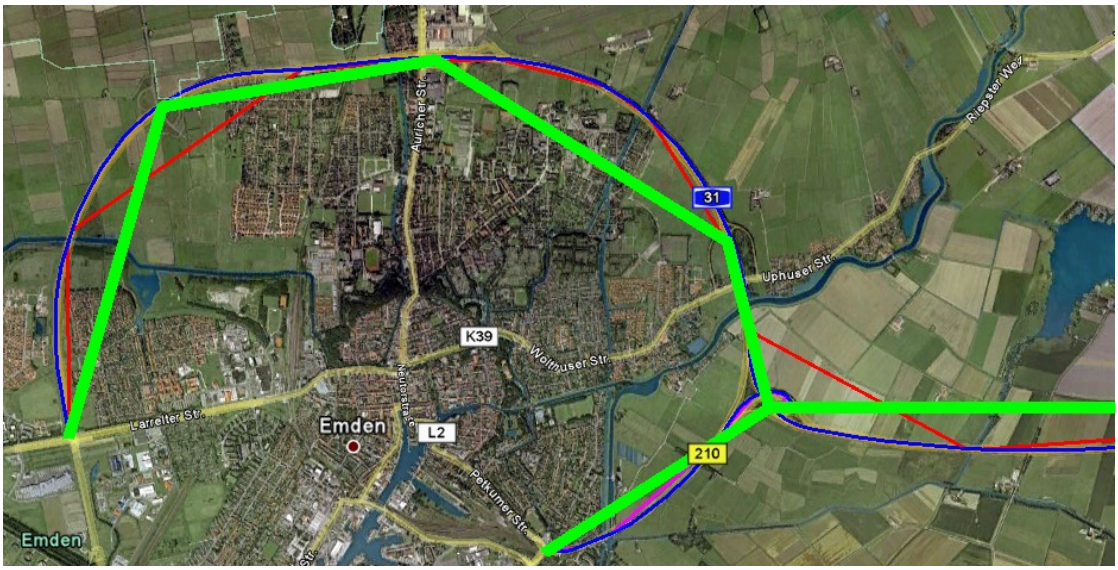

Abbildung 4.16: Straßenverlauf der ursprünglichen EGT (grün), der im ersten Verfahren erstellen Light-Map (rot) und der schließlich fertigen Full-Map (blau), nach dem die Relationen hergestellt wurden.

5. Fazit und Ausblick

Die im Rahmen dieser Bachelorarbeit erzielten Ergebnisse sind im Großen und Ganzen sehr zufriedenstellend. Zwar bedarf es an einigen Stellen der Implementierung einer etwas umfassenderen Fallunterscheidung, allerdings reichen die bisher implementierten Algorithmen für eine erste Optimierung der DDG-Daten völlig aus.

`http://project.informatik.uni-osnabrueck.de/pbertram/`

Unter diesem Link können die drei nachfolgenden und während des Optimierungsprozesses entstandenen KML-Files

`egt_streets.kml`, `lightMap.kml`, `fullMap.kml`

über *GoogleEarth* oder *GoogleMaps* begutachtet werden. Für die Verwendung von *GoogleMaps* einfach den Link mit der gewünschten Datei in das Suchfeld hineinkopieren.

Die `egt_streets` beinhaltet die Daten der EGT. Hier werden die ungenauen Straßenverläufe bereits bei größerer Entfernung sichtbar. In der `lightMap` sind alle OSM-Nodes enthalten, die über die in Abschnitt 4.3.1 behandelten Verfahren entstanden sind. Auch diese ist nicht Vollständig, denn sie enthält – zur Erinnerung – lediglich die sich am Ende oder Anfang eines *Ways* befindenden *Nodes*. Dennoch ist bereits hier ein wesentlich genauerer Straßenverlauf zu erkennen. Letztendlich enthält die `fullMap` die aus den Relationen generierten OSM-*Nodes*. Diese enthält zwar bei Weitem eine größere Anzahl an Nodes als beide KML-Files zuvor, dennoch weniger Straßenabschnitte, als die `lightMap`.

Einer der Gründe dafür ist in Abbildung 5.1 dargestellt. Der hervorgehobene Kartenausschnitt zeigt den Verlauf der 'A31' ohne (hier rot) und mit Lücke (hier blau). Dieser Effekt kommt daher zustande, dass die 'A31' in der EGT noch nicht vollständig und in zwei große Wegstücke aufgeteilt ist. Das eine Stück reicht von Emden bis, wie in der Abbildung zu sehen, etwa Meppen und findet erst weiter südlich bei etwa Bad Bentheim ihren Anschluss wieder. Da der DDG zur dieses Teilstück ohnehin keine Verkehrsdaten vorliegen, ist es vollkommen in Ordnung, wenn hierfür auch keine OSM-*Nodes* Referenziert werden.

Zudem stehen die Daten der in der vorliegenden Arbeit erzeugten Datenbank unter folgendem Link ebenfalls zur Verfügung:

`http://dbs.informatik.uos.de/phpmyadmin/`

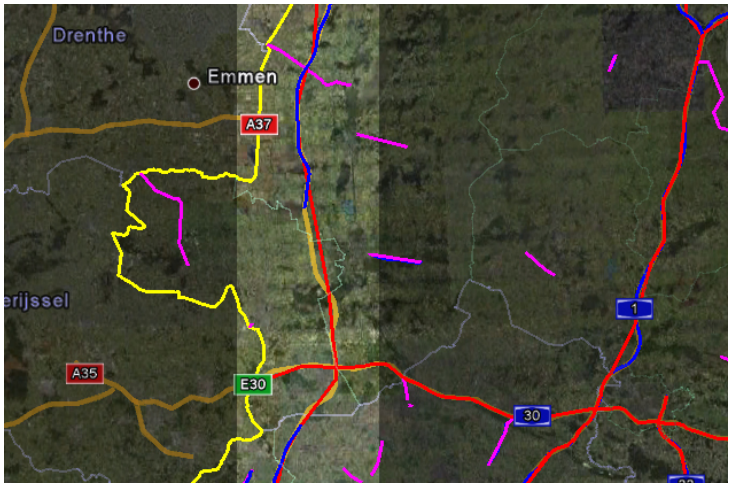

Abbildung 5.1: Fehlende Teilstrecke der Autobahn 'A31'

Eine bisher außer Acht gelassene Problematik liegt zudem in dem Aufbau der Straßen über die OpenStreetMap Daten. In der Regel besteht eine Straße aus mehreren Spuren. Die in Abschnitt 4.3.1 vorgestellte Methode zum Auffinden eines `startNodes` anhand des geringsten Abstandes zu dem erst EGT-*Node*, könnte hier nicht unbedingt den am besten geeigneten OSM-*Node* liefern. Sie liefert lediglich einen Node auf irgendeinen dieser Spuren, in Abhängigkeit des Abstandes, die eventuell weniger gut den gesamten Verlauf der Autobahn oder Bundesstraße wiedergibt. Diese könnte beispielsweise durch Fahrbahnverengungen eher enden, als andere. Hier sollte in Zukunft ein etwas geeigneteres Verfahren implementiert werden, welches auch derartige Fälle ausschließt.

Außerdem könnte in Betracht gezogen werden, die Möglichkeit der direkten *online*-Verbindung zu der Geodatenbank zu implementieren. Folglich kann das zeitaufwendige Parsen und Importieren der OSM-Daten vermieden und außerdem die Aktualität der Daten gewährleistet werden.

Letztendlich erlaubt der weitestgehend modulare Aufbau eine relativ problemlose Erweiterung des Algorithmus.

6. Zusammenfassung

Durch die Vielzahl der innovativen Errungenschaften in der IT-Branche in den letzten Jahren nehmen computerbasierte Systeme einen immer größeren Stellenwert in der heutigen Gesellschaft ein. Es werden kaum noch Endgeräte hergestellt, die nur einem einzigen Zweck dienen. Der Trend bewegt sich in Polyfunktionalität. Als populärstes Beispiel sollen die derzeitigen Mobiltelefone erwähnt werden. Neben den noch vor ein paar Jahren üblichen Funktionen wie Telefonieren oder dem versenden Textnachrichten, verfügen die heutigen *Smartphones* über ein Sammelsurium an Applikationen. Darunter gewinnt auch der Gebrauch des GPS immer mehr an Bedeutung, welches hauptsächlich in den Navigationssystemen der Autoindustrie Verwendung findet. Diese benötigen stets aktuelle und genaue Informationen über den Verkehr und den Straßenverlauf.

Die *Gesellschaft für Verkehrsdaten* (DDG) ist ein Unternehmen, welches verkehrstelematisch interessante Informationen europaweit sammelt und zur Verfügung stellt. Sie müssen also fortwährend Messungen durchführen und ihren Datenbestand aktualisieren um den Autofahrern zuverlässig aktuelle Prognosen liefern zu können. Aber auch andere Unternehmen, unter denen sich ebenso kommerzielle wie unkommerzielle befinden, führen derartige Messungen durch. Zu dem populärsten kommerziellen Unternehmen gehört die *Google Inc.*. Sie gibt ihre Daten aber nur in Verbindung hoher Kosten oder mit strengen Richtlinien heraus. Anders ist die Philosophie des seit 2004 gegründeten Projects *OpenStreetMap*. Sie erhebt ebenfalls, aber über freiwillige Mitglieder, geografische Daten und stellt sie der Öffentlichkeit in einer freien Geodatenbank zur Verfügung.

Unter diesem Aspekt können zwar keine verkehrsrelevanten Informationen, dafür aber geografische Daten aus der Geodatenbank von OpenStreetMap zur Optimierung und Aktualisierung der Straßennetze genutzt werden. Das Ziel dieser Arbeit war, einen dafür vorgesehenen Algorithmus zu entwickeln, der anhand der Geodaten von OpenStreetMap eben diese Straßennetzdaten der DDG optimiert.

Herausgekommen ist letztendlich ein Algorithmus, der beide Datensammlungen in eine lokale Datenbank schreibt und anhand dieser eine Relation zwischen ihnen herstellt. Aus diesen gewonnenen Informationen können über Angabe der Straßen der DDG alle zu diesen gehörenden Straßenknoten von OpenStreetMap geholt werden.

Es liegt nahe für den weiteren Gebrauch eine Möglichkeit der direkten Verbindung zu der Geodatenbank zu implementieren. Dadurch entfällt das zeitaufwendige Parsen und Importieren der Daten in die lokale Datenbank.

7. Summary

By the multiplicity of innovative achievements in the IT-industry within the past years, computer-based systems take up a snowballing significance in today's society. Devices which serve only one purpose are barely produced. The trend moves toward polyfunctionality. As the most popular example, present mobile phones are to be mentioned: Next to functions such as telephoning or text messaging, which were usual a few years ago, today's smartphones have a lot of applications at their disposal. Among them, the usage of GPS becomes more and more important. This technology is mainly used in automobile navigation systems of the automobile industry, where it always is in need of current, exact information on traffic and, furthermore, the course of the road. The «Gesellschaft für Verkehrsdaten» (DDG) is a company, which collects and provides traffic-relevant information within europe. Hence, they have to take measurements and to update their database continually to deliver dependably and current prognoses to the car drivers. But also other companies, among them commercial as well as uncommercial ones, take such measurements. One of the most popular commercial companies is Google Inc., which, however, only commits its data in conjunction with high costs and strict guidelines. On the contrary, the 2004 established project OpenStreetMap raises geographic data with the help of volunteers and, hence, is able to provide them to public within a free geo data base.

As per this aspect, no traffic-relevant information, but instead geographic data out of this OSM database can be used for optimizing and updating the road network. The aim of this work was the development of a designated algorithm, which is able to optimize the DDG road network data by means of the OSM geo data.

The final result is an algorithm, which writes both data pools into one local data bank and, on the basis of that,establishes a relation between them. Out of the gained information, every appropriate OSM street node can be imported by only naming the street in DDG.

It stands to reason to implement a possibility of direct connection to the geo data base for further purpose in the future. Thus, time-consuming parsing and importing of data into the local database are dropped out.

Literatur- und Webverzeichnis

[1] **Dieter Jungnickel**, *Graphs, Networks and Algorithms*, 2nd Edition, Springer 2005

[2] **Brett McLaughlin**, *Java and XML*, , O'Reilly 2000

[3] **A. Kempler & A. Eickler**, *Datenbanksysteme - Eine Einführung*, 3. Auflage, R. Oldenburg Verlag München, 1999

[4] **Michael Kifer**, **Arthur Bernstein** & **Philip M. Lewis**, *Database Systems - An Application-Oriented Approach*, 2nd Edition, Pearson/Addison Wesley, 2006

[5] **Manfred Nitsche**, *Graphen für Einsteiger - Rund um das Haus vom Nickolaus*, 3. Auflage, Vieweg+ Teubner 2009

[6] **Stuart Russel & Peter Norvig**, *Atrificial Intelligence - A Modern Approach*, 2nd Edition, Prentice Hall, 2002

[7] **Helmut Vonhoegen**, *Einstieg in XML – Grundlagen, Praxis, Referenzen*, , Galileo Computing 2007 Application-Oriented Approach, 2nd Edition, Person International Edition,

[8] **Oliver Vornberger**, *Graphenalgorithmen* (Vorlesungsskript), 1998
`www-lehre.informatik.uni-osnabrueck.de/~graph/skript/skript.html`

[9] **v.A.**, *Richtlinie für die Vergabe von Geocodes v1.1*

[10] **v.A.**, *ddg Gesellschaft für Verkehrsdaten*
`www.ddg.de`

[11] **v.A.**, *JDBC*
`java.sun.com/javase/technologies/database/`
`java.sun.com/products/jdbc/download.html#corespec40`

[12] **v.A.**, *FAQ zur OpenStreetMap*
`www.openstreetmap.de/faq.html#was_ist_osm`

[13] **v.A.**, *OpenStreetMap Datenprimitive*
`wiki.openstreetmap.org/wiki/Node#Node`

[14] **v.A.**, *OpenStreetMap MapFeature*
`wiki.openstreetmap.org/wiki/Map_Features`

[15] **v.A.**, *OpenStreetMap Relation*
`wiki.openstreetmap.org/wiki/Relations`

[16] **v.A.**, *OpenStreetMap Relation:Route*
`wiki.openstreetmap.org/wiki/DE:Relation:route`

[17] **v.A.**, *OpenStreetMap API*
`wiki.openstreetmap.org/index.php/OSM_Protocol_Version_0.6`

Aufgrund der Aktualität der in dieser Bachelorarbeit eingesetzten Techniken sind zum Beleg viele Internetseiten und verlinkte pdf-Dokumente aufgeführt. Da sich die Erreichbarkeit und der Inhalt dieser verlinkten Seiten schnell ändern können, kann nicht sichergestellt werden, dass die unter den Links zu findenden Informationen noch aktuell sind. Zum Zeitpunkt der Fertigstellung dieser Bachelorarbeit am 14.12.2009 waren alle im Literatur- und Webverzeichnis, sowie in den Fußnoten aufgeführte Quellen zu finden.

Abbildungsverzeichnis

Quellcodeverzeichnis

A. OSM Doctype Definition

```
<?xml version=" 1 . 0 " encoding="UTF..8" ?>
<!ELEMENT osm (node|relation|way)*>
<!ATTLIST osm version (0.5) #REQUIRED}>
<!ATTLIST osm generator CDATA #REQUIRED}>

<!ELEMENT node (tag*)>
<!ATTLIST node id CDATA #REQUIRED>
<!ATTLIST node lat CDATA #REQUIRED>
<!ATTLIST node lon {CDATA #REQUIRED>
<!ATTLIST node visible CDATA #IMPLIED>
<!ATTLIST node user CDATA #IMPLIED>
<!ATTLIST node timestamp CDATA #IMPLIED>

<!ELEMENT way (tag*, nd, tag*, nd, (tag|nd)*)>
<!ATTLIST way id CDATA #REQUIRED>
<!ATTLIST way visible CDATA #IMPLIED>
<!ATTLIST way user CDATA #IMPLIED>
<!ATTLIST way timestamp CDATA #IMPLIED>

<!ELEMENT nd EMPTY>
<!ATTLIST nd ref CDATA #REQUIRED>

<!ELEMENT relation ((tag|member)+)>
<!ATTLIST relation id CDATA #REQUIRED>
<!ATTLIST relation visible CDATA #IMPLIED>
<!ATTLIST relation user CDATA #IMPLIED>
<!ATTLIST relation timestamp CDATA #IMPLIED>

<!ELEMENT member EMPTY>
<!ATTLIST member type (way|node|relation) #REQUIRED>
<!ATTLIST member ref CDATA #REQUIRED>
<!ATTLIST member role CDATA #IMPLIED>

<!ELEMENT tag EMPTY>
<!ATTLIST tag k CDATA #REQUIRED>
<!ATTLIST tag v CDATA #REQUIRED>
```

B. Nachtrag zur XML-Manipulation

XSLT baut auf der logischen Baumstruktur eines XML-Dokumentes auf und dient zur Definition von Umwandlungsregeln. XSLT-Programme, sogenannte XSLT-Stylesheets, sind dabei selbst nach den Regeln des XML-Standards aufgebaut. Die Stylesheets werden von spezieller Software, den XSLT-Prozessoren, eingelesen, die mit diesen Anweisungen ein oder mehrere XML-Dokumente in das gewünschte Ausgabeformat umwandeln. XSLT-Prozessoren sind auch in vielen modernen Webbrowsern integriert.

C. Tabelle zu den Spaltennamen der EGT

Spaltenname	Beschreibung
EGT_ID	beschreibt die fortlaufende Nummer eines Eintrages in der EGT.
EGT_GEOCODE	Jedes topographische Objekt, welches für verkehrstelematische Dienste relevant sein kann, bekommt einen Eintrag in der Endgerätetabelle und damit einen Geocode. Der Geocode muss bei allen Einträgen in der Endgerätetabelle gesetzt sein. Im Falle der Straßenknoten können mehrere Einträge der Endgerätetabelle den gleichen Geocode besitzen, da sie mehreren Straßen zugeordnet sein können. Eindeutig bestimmt werden können sie nur in Verbindung mit dem Attribut Straße.
EGT_LFDNR	Hier wird die laufende Nummer innerhalb einer Strecke angegeben.
EGT_TYP	Der Typ gibt die Art des Geocodes an. Der Geocodetyp muss bei allen Einträgen in der Endgerätetabelle gesetzt sein. Die Anzahl der möglichen Geocodetypen ist auf 256 begrenzt (0 bis 255)[19].
EGT_STRASSE	Das Attribut Straße wird nur bei Einträgen von Straßenknoten (Geocodetypen 1,2,3,5,6,7,8,30,31,32,33,34,35) sowie von Straßen-POI belegt. Es nennt die offizielle Bezeichnung der Straße, die sich aus dem Kürzel des Straßentyps und der Straßennummer zusammensetzt, wobei die Nummer ohne Leerzeichen hinter dem Typkürzel steht. Folgende Straßentypen sind möglich: Autobahn (z.B. A1), Bundesstraße (z.B. B1) und sonstige Straßen wie Landstraße (z.B. L1), Staatsstraße (z.B. ST1) und Kreisstraße (z.B. K1) etc.
EGT_NAME	Im Attribut Name steht die offizielle Benennung des kodierten Objekts (vgl. 5). Der Name muß bei allen Einträgen in der Endgerätetabelle gesetzt sein.

[19] Tabelle im Anhang zu finden

EGT_ REFGEOCODE	Referenz-Geocodes werden nur bei Nicht-Autobahnknoten, den Verwaltungsgebieten sowie den Points of Interest vergeben. Sie dienen einer besseren Beschreibung der kodierten Objekte. So können über den Referenz-Geocode beispielsweise Rückschlüsse über die Lage eines Geocodes innerhalb einer Stadt gezogen werden. Bei Straßenknoten von Bundes- oder sonstigen Straßen wird als Referenz der Geocode der Gemeinde/Stadt angegeben, in dem der Mittelpunkt des Knotens liegt. Ist dies eine Stadt mit mehr als 150.000 Einwohnern, wird, sofern vorhanden, der Geocode des Stadtteils angegeben. Gibt es keinen Stadtteil, wird der Geocode des Stadtbezirks angegeben. Ebenso wird bei Points of Interest verfahren. Bei den Verwaltungsgebieten wird als Referenz der Geocode der hierarchisch unmittelbar höherstehenden Gebietskörperschaft angegeben. Ausnahmen sind der Landkreis mit dem Bundesland als Referenz (und nicht ggf. dem Regierungsbezirk) sowie der Staat ohne Referenz. Einem Stadtteil wird generell der Geocode der Stadt/Gemeinde zugewiesen, der er angehört.
EGT_ NUMMER-RICHTUNG-(VON/NACH)ORT	Die Attribute Nummer in Richtung von Ort und Nummer in Richtung nach Ort enthalten bei Geocodes von Straßenknoten die offiziell für diesen Knoten vergebene Nummer. Da diese Nummern von der Fahrtrichtung auf der jeweiligen Straße abhängen können, sind zwei Spalten für die Nummern vorgesehen, abhängig von der Fahrtrichtung <von-Ort> und <nach-Ort>. Die Nummern werden den amtlichen Verzeichnissen entnommen.

EGT_ (VON/NACH)ORT	Strecken sind Teilstücke von Straßen, die durch die Richtungsangaben (Fernziele) delimitiert werden. Die Festlegung der Strecken sowie die Auswahl der Richtungsorte richtet sich bei Autobahnstrecken nach der vor Ort vorhandenen Beschilderung. Bei den untergeordneten Straßen werden als Fernziele/Richtungsorte primär Benennungen von überregional bekannten Ortschaften gewählt. Durch die Angabe der Richtungsorte in Verbindung mit den Attributen Vorgänger-Knoten und Nachfolger-Knoten ist auch die „Default-Richtung“einer Straße bzw. eines Streckenabschnitts festgelegt.
EGT_ (LÄNGE/BREITE) (1/2)	Die x- und y-Koordinaten der Straßenknoten erhalten ihren Eintrag in diesen Spalten. Für den Geocodetyp 92 (Straßenabschnitte, vgl. 3.4) beschreiben Laenge1 und Breite1 den Startpunkt und Laenge2 und Breite2 den Endpunkt des Straßenabschnitts.
EGT_ STRASSEINT	Die internationale numerische Bezeichnung der Straße setzt sich aus dem Kürzel des Straßentyps (E) und der Straßennummer zusammen, wobei die Nummer ohne Leerzeichen hinter dem Typkürzel steht.
EGT_NAMEKURZ	Alle Objekte, außer die der Geocodetypen 92, 98, 99, erhalten einen abgekürzten Namen in dieser Spalte.
EGT_ VORGAENGER/ NACHFOLGER	Bei Straßenknoten werden hier die Geocodes der Vorgänger- bzw. Nachfolger-Knote angegeben. Das Feld Vorgänger-Knoten wird nicht belegt, wenn es sich bei dem Knoten um den ersten Knoten einer Straße (=Straßenanfang) oder um den ersten Knoten nach einer Unterbrechung der Straße handelt. Ebenso wird das Feld Nachfolger-Knoten nicht belegt, wenn es sich um den letzten Knoten einer Straße (=Straßenende) oder um den letzten Knoten vor einer Unterbrechung der Straße handelt. Der Knoten sowie die für diesen Knoten angegebenen Vorgänger- bzw. Nachfolger-Knoten gehören generell zur gleichen Straße.

D. Berechnung der Geocodes

Anhand von Geocodes wird eine für Europa einheitliche Geocodierung vorgesehen. Folgende Vorgaben werden gemacht:

- Benutzung eines festen Bezugspunktes für Europa (2 x 16 Bit)
- Verwendung einer einzigen Länderregion (ca. 6500 km x 6500 km)

Die Länderregion deckt ein Gebiet von ca. 6500x6500 km^2 ab. Um Punkte in diesem Gebiet angemessen kodieren zu können, werden für die Kodierung 2 x 16 Bit (2x65536) verwendet.

D.1. Umrechnung Geographische Koordinate → Geocode

Um diese Umrechnung durchführen zu können, wird ein Bezugspunkt benötigt. Der Bezugspunkt für das Gebiet von Europa lautet geodätisch in Gradangaben im WGS84 Koordinatensystem: 52,5° Breitengrad und 13,5° Längengrad. Dieser Punkt ist gleichfalls der „Nullpunkt“in dem 6500x6500 km^2 Gebiet. Dies heißt, dass Abstände in westlicher und südlicher Richtung negative Vorzeichen haben. Die Abstände werden in 16 Bit im Zweierkomplement kodiert.

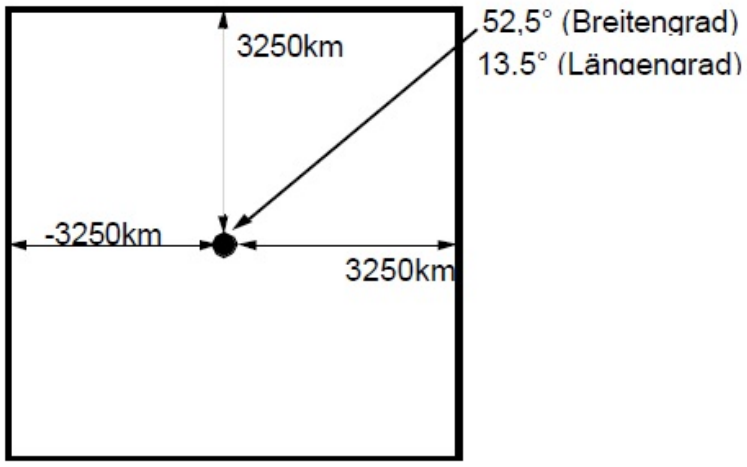

Abbildung D.1: Bezugspunkt

Die Formeln für die Umrechnung der WGS84-Koordinaten in die Abstände zum Bezugs-

punkt (hier für Europa) lauten:

$$\begin{aligned} x &= (\lambda_1 - \lambda)/(360^\circ) \cdot 400300 \cdot \cos\phi_1 \\ y &= (\phi_1 - \phi)/(360^\circ) \cdot 400300 \end{aligned}$$

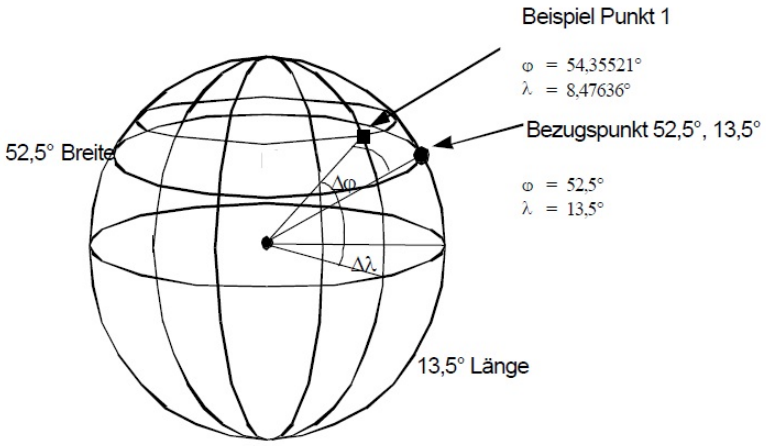

Abbildung D.2: Berechnung des Geocodes

Ein Punkt, der in realen geographischen Koordinaten (bezogen auf das WGS84-Ellipsoid, Dezimaldarstellung mit 5 Nachkommastellen) vorliegt, wird mit obigen Vorgaben wie folgt kodiert. Als Beispiel wird der Punkt 54,35521 ; 8,47636 (WGS84-Ellipsoid) genommen. Zur Ermittlung der Abstände zu dem festen Bezugspunkt (siehe oben) wird für die vereinfachte Umrechnung die Erde als Kugel mit einem festen Radius von 6371 km angenommen (Erdumfang 40030 km).

Zuerst wird der Abstand des Punktes zum Bezugspunkt in ost- westlicher Richtung in der Einheit [100 m] ermittelt. Dieser beträgt:

D.2. Umrechnung Geocode → Geographische Koordinate

Für die Umrechnung des Geocodes zurück in die geographischen Koordinaten wird wieder ein fester Erdradius von 6371 km (Umfang 40030 km) und der feste Bezugspunkt verwendet. Für die Umrechnung kommen folgende Formeln zur Anwendung:

$$\begin{aligned} \phi_1' &= y \cdot 360^\circ/(400300) + \phi \\ \lambda_1' &= x \cdot 360^\circ/(400300 \cdot \cos\phi_1') + \lambda \end{aligned}$$

Bei dieser Umrechnung ist zu beachten, dass zuerst mit dem Wert y der Breitengrad zurückgerechnet wird. Dieser Wert fließt in die Berechnung des Längengrades mit ein. Für die Werte x und y werden die Abstandswerte mit einer Auflösung von 100 m eingesetzt, so wie sie in dem Geocode in 2X16 Bit übermittelt wurden.

Für die Errechnung des Kosinus reichen 9 signifikante Nachkommastellen (32Bit) für eine ausreichende Genauigkeit aus. Die errechneten Koordinaten werden auf fünf Nachkommastellen auf- bzw. abgerundet.

E. Danksagung

An dieser Stelle möchte ich allen danken, die mich bei der Erstellung dieser Bachelorarbeit unterstützt und es mir so überhaupt erst ermöglicht haben, diese zu vollenden.

- Herrn Prof. Dr. Oliver Vornberger danke ich für die Bereitstellung des interessanten Themas, die Ermöglichung der Durchführung.
- Frau Dipl. Math. Dorothee Kunze danke ich für die sehr gute Betreuung, sowie Beratung und Hilfestellung bei allen Fragen und Anliegen.
- Herrn Dipl. Math. Patrick Fox möchte ich für die stellvertretende und überaus hilfreiche Betreuung der Arbeit, sowie für die Beratung und Hilfestellung bei allen Fragen und Anliegen, ebenfalls danken.
- Für die Übernahme des Zweitgutachtens möchte ich mich herzlich bei Frau Jun. Prof. Dipl.Ing Elke Pulvermüller bedanken.
- Danke an Tim Jödden, der stets Zeit gefunden hat, mir bei diversen Problemen zu helfen.
- Zu guter letzt gilt ein großer Dank meiner Freundin Agnieszka Katharina Musiol. Ohne Dich wär ich vor allem in der Endphase meiner Arbeit verzweifelt und aufgeflogen. Vielen Dank für Deine Ausdauer und Bereitschaft mir zu helfen und alles für mein Wohlbefinden zu tun. Vielen Dank auch dafür, dass Du mir gelegentlich Druck gemacht hast, ohne den ich mit Sicherheit nicht so weit gekommen wäre. Danke, dass Du mir meine Macken verzeihst und mir immer wieder aufs neue beweist, wie sehr Du mich liebst. Ich hoffe du weißt, dass ich sehr stolz auf Dich bin und dich über alles liebe.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Osnabrück, 14. Dezember 2009 _____

Unterschrift Phillipp Bertram