

Hochschule Aalen

Fakultät Elektronik und Informatik

Optimiertes Routing mit realen Geodaten

Projektarbeit

vorgelegt am 28. Februar 2025 von

Christoph Gerling

Matrikelnummer: 3005809

Referent : Prof. Dr. Winfried Bantel

Zusammenfassung

Die vorliegende Projektarbeit befasst sich mit der Implementierung, Optimierung und Visualisierung von Routingalgorithmen auf OpenStreetMap-Daten. Im Fokus steht dabei die Entwicklung einer interaktiven webbasierten Anwendung, die verschiedene Routingalgorithmen wie Breitensuche, Dijkstra und A* vergleicht und deren Arbeitsweise visuell darstellt. Ein zentraler Beitrag ist die Implementierung einer effizienten Graphenkontraktion, die den Berechnungsaufwand durch gezielte Reduktion von Zwischenknoten erheblich verringert. Die Evaluation auf verschiedenen Datensätzen (städtisch, ländlich, Autobahnnetz) zeigt eine Laufzeitverbesserung von 10% bis 95%, abhängig von der Netzwerkstruktur und dem gewählten Algorithmus. Bei Autobahndaten mit ihrer geringen Verzweigungsdichte konnte eine Knotenreduktion von über 93% erreicht werden, was zu einer Beschleunigung der Berechnungen um den Faktor 20 führte. Die entwickelte Anwendung bietet wertvolle Einblicke in die Funktionsweise von Routingalgorithmen und demonstriert das erhebliche Optimierungspotenzial durch strukturelle Graphentransformationen für die effiziente Navigation auf realen Straßennetzen.

Inhaltsverzeichnis

Abbildungsverzeichnis	III
Tabellenverzeichnis	V
1 Einleitung	1
1.1 Motivation	1
1.2 Zielsetzung	1
1.2.1 Entwicklung einer webbasierten Anwendung	2
1.2.2 Implementierung und Optimierung von Routingalgorithmen	2
1.2.3 Simulation und Visualisierung	2
1.3 Aufbau der Arbeit	2
2 Grundlagen und Hintergründe	4
2.1 Koordinaten	4
2.1.1 WGS84	4
2.2 OpenStreetMap	5
2.2.1 Geschichte und Entwicklung	5
2.2.2 Datenstruktur	5
2.2.3 Tagging-System	6
2.2.4 Datenqualität und -vollständigkeit	6
2.2.5 Overpass API	6
2.3 OpenLayers	7
2.3.1 Architektur und Aufbau	7
2.3.2 Layer-Konzept und Datenvizualisierung	8
2.4 Graphentheorie	9
2.4.1 Graphentypen und Eigenschaften	9
2.4.2 Knotengrad	10
2.4.3 Algorithmen zur Wegfindung	11
2.4.4 Graphenkontraktion	13
2.4.5 Metriken	14
3 Implementierung	16
3.1 Datenbeschaffung	16
3.1.1 Overpass API	16
3.1.2 Datenverarbeitung	21
3.1.3 Graphenstruktur	23
3.2 Graphenkontraktion	26
3.2.1 Pseudocode	28
3.2.2 Beispiel Kontraktion	33
3.2.3 Sonderfälle bei der Kontraktion	42

3.3	Routing	43
3.3.1	Heuristiken	43
3.3.2	Optimierungen	45
3.3.3	Beschreibung des Algorithmus	47
3.4	Weboberfläche	50
3.4.1	Kartenansicht	51
3.4.2	Interaktions-Panel	53
4	Evaluation	66
4.1	Vorgehensweise	66
4.1.1	Vergleichsmetriken	66
4.2	Ergebnisse	67
4.2.1	Wilhelma - Stuttgart	67
4.2.2	Städtische Gebiete - Stuttgart	68
4.2.3	Autobahnen	69
4.2.4	Ländliche Gebiete - Backnang	71
4.2.5	Bundesländerweit - Baden-Württemberg	72
4.3	Limitationen	72
4.3.1	Overpass API Limitierung	72
4.3.2	Erstellung des Graphen	73
4.4	Vergleich / Diskussion	73
4.4.1	Einfluss der Netzwerkstruktur auf die Optimierungseffizienz	73
4.4.2	Algorithmenspezifische Unterschiede	75
4.4.3	Skalierbarkeit und praktische Anwendbarkeit	76
4.4.4	Charakteristische Muster der Optimierung	76
4.4.5	Fazit der Evaluation	77
5	Zusammenfassung und Ausblick	79
5.1	Zusammenfassung	79
5.2	Optimierungspotenzial	79
5.2.1	Speicheroptimierungen der Graphenkontraktion	80
5.2.2	Andere Optimierungsverfahren	80
5.3	Funktionalitäten	82
5.3.1	Höchstgeschwindigkeit als Metrik	82
5.3.2	Filtern von Wegtypen	82
5.3.3	Unterschiedliche Detailtiefe	82

Abbildungsverzeichnis

2.1	WGS84 Koordinatensystem [9]	4
2.2	Vereinfachte Objekthierarchie in OpenLayers [Eigene Darstellung]	8
2.3	Beispiele für Graphentypen in der Routenberechnung [Eigene Darstellung]	10
2.4	Beispiel für Knotengrade in einem ungerichteten Graphen [Eigene Darstellung]	11
2.5	Beispiel einer Graphenkontraktion [Eigene Darstellung]	14
3.1	Beispiel für die Funktionsweise der Funktion ConstructNewPath [Eigene Darstellung]	33
3.2	Beispiel Graph vor der Kontraktion [Eigene Darstellung]	34
3.3	Beispiel Graph nach der Kontraktion des Knotens 1 [Eigene Darstellung]	35
3.4	Beispiel Graph nach der Kontraktion des Knotens 2 [Eigene Darstellung]	36
3.5	Beispiel Graph nach der Kontraktion des Knotens 3 [Eigene Darstellung]	37
3.6	Beispiel Graph nach der Kontraktion des Knotens 4 [Eigene Darstellung]	38
3.7	Beispiel Graph nach der Kontraktion des Knotens 6 [Eigene Darstellung]	40
3.8	Beispiel Graph nach der Kontraktion des Knotens 5 [Eigene Darstellung]	41
3.9	Sonderfall Rundweg [Eigene Darstellung]	43
3.10	Beispiel Knoten auf einer kontrahierten Kante [Eigene Darstellung]	45
3.11	Mehrere Wege zwischen zwei Verzweigungen [Eigene Darstellung]	49
3.12	Mehrere Wege zwischen zwei Verzweigungen richtige Rekonstruktion [Eigene Darstellung]	49
3.13	Mehrere Wege zwischen zwei Verzweigungen Start- und Endknoten auf den Wegen [Eigene Darstellung]	50
3.14	Weboberfläche zur Interaktion mit dem Algorithmus [Eigene Darstellung]	50
3.15	Kartenansicht der Weboberfläche [Eigene Darstellung]	51
3.16	Darstellung des vollständigen Graphen [Eigene Darstellung]	53
3.17	Interaktions-Panel [Eigene Darstellung]	53
3.18	Ergebnisse der Routenberechnung [Eigene Darstellung]	54
3.19	Start- und Endpunkt auswahl in der Weboberfläche [Eigene Darstellung]	55
3.20	Hover Effekt bei der Auswahl von Start- und Endpunkten [Eigene Darstellung]	55
3.21	Auswahl von Start- und Endpunkten auf der Karte [Eigene Darstellung]	56
3.22	Start- und Endpunkt auswahl in der Weboberfläche [Eigene Darstellung]	56

3.23	Karten Optionen in der Weboberfläche [Eigene Darstellung]	57
3.24	Kontrahierter Graph auf der Karte [Eigene Darstellung]	57
3.25	Interaktions-Layer mit Start-, Endknoten und Pfad [Eigene Darstellung]	58
3.26	Auswahl eines Knotens und Anzeige der möglichen Start- und Endknoten [Eigene Darstellung]	59
3.27	Knoten des gefundenen Pfades auf der Karte [Eigene Darstellung]	60
3.28	Simulations Steuerelemente in der Weboberfläche [Eigene Darstellung]	61
3.29	Zwischenstand der Simulation mit dem Optimierten-A*-Algorithmus [Eigene Darstellung]	62
3.30	Endstand der Simulation mit dem Optimierten-A*-Algorithmus [Eigene Darstellung]	63
3.31	Zwischenstand der Simulation mit dem Optimierten-A*-Algorithmus [Eigene Darstellung]	63
3.32	Endstand der Simulation mit dem Optimierten-A*-Algorithmus [Eigene Darstellung]	64
3.33	Verschiedene Benachrichtigungen in der Weboberfläche [Eigene Darstellung] . .	65
4.1	Prozentuale Verbesserung der Laufzeit durch Graphenkontraktion nach Datensatztyp und Algorithmus [Eigene Darstellung]	73
4.2	Zusammenhang zwischen Verzweigungsdichte und Optimierungspotenzial durch Graphenkontraktion für verschiedene Algorithmen [Eigene Darstellung]	77
5.1	Beispiel für das Einfügen von Abkürzungen [Eigene Darstellung]	80

Tabellenverzeichnis

3.1	Knotenstruktur des Beispielgraphen	34
3.2	Identifizierte kontrahierbare Knoten	34
3.3	Knotenstruktur des Beispielgraphen nach der Kontraktion des Knotens 1	35
3.4	node_next_branching_nodes nach der Kontraktion des Knotens 1	35
3.5	nodes_uncontracted_to_contracted nach der Kontraktion des Knotens 1	36
3.6	Knotenstruktur des Beispielgraphen nach der Kontraktion des Knotens 2	36
3.7	node_next_branching_nodes nach der Kontraktion des Knotens 2	37
3.8	nodes_uncontracted_to_contracted nach der Kontraktion des Knotens 2	37
3.9	Knotenstruktur des Beispielgraphen nach der Kontraktion des Knotens 3	37
3.10	node_next_branching_nodes nach der Kontraktion des Knotens 3	38
3.11	nodes_uncontracted_to_contracted nach der Kontraktion des Knotens 3	38
3.12	Knotenstruktur des Beispielgraphen nach der Kontraktion des Knotens 4	39
3.13	node_next_branching_nodes nach der Kontraktion des Knotens 4	39
3.14	nodes_uncontracted_to_contracted nach der Kontraktion des Knotens 4	39
3.15	Knotenstruktur des Beispielgraphen nach der Kontraktion des Knotens 6	40
3.16	node_next_branching_nodes nach der Kontraktion des Knotens 6	40
3.17	nodes_uncontracted_to_contracted nach der Kontraktion des Knotens 6	40
3.18	Knotenstruktur des Beispielgraphen nach der Kontraktion des Knotens 5	41
3.19	node_next_branching_nodes nach der Kontraktion des Knotens 5	41
3.20	nodes_uncontracted_to_contracted nach der Kontraktion des Knotens 5	42
3.21	Finaler Zustand node_next_branching_nodes	42
3.22	Vergleich der verschiedenen Haversine-Implementierungen	44
4.1	Leistungsvergleich verschiedener Algorithmen auf vollständigen und kontrahierten Graphen des Wilhelma-Datensatzes	68
4.2	Leistungsvergleich verschiedener Algorithmen auf vollständigen und kontrahierten Graphen des Stuttgart-Datensatzes	69
4.3	Leistungsvergleich verschiedener Algorithmen auf vollständigen und kontrahierten Graphen des Autobahn-Deutschland-Datensatzes	70
4.4	Leistungsvergleich verschiedener Algorithmen auf vollständigen und kontrahierten Graphen des Autobahn-Sachsen-Datensatzes	71

4.5 Leistungsvergleich verschiedener Algorithmen auf vollständigen und kontrahierten Graphen des Backnang-Datensatzes	71
4.6 Leistungsvergleich verschiedener Algorithmen auf vollständigen und kontrahierten Graphen des Baden-Württemberg-Datensatzes	72

1 Einleitung

1.1 Motivation

In der heutigen, zunehmend vernetzten Welt spielen Routingalgorithmen eine zentrale Rolle im täglichen Leben. Von Navigationsanwendungen auf mobilen Endgeräten bis hin zu logistischen Systemen für die Warenlieferung – die Fähigkeit, effizient den optimalen Weg zwischen zwei Punkten zu berechnen, ist zu einer Schlüsseltechnologie geworden [1]. Täglich verlassen sich Millionen von Menschen auf diese Algorithmen, um ihren Weg zum Arbeitsplatz, zu Freizeitaktivitäten oder auf Reisen zu finden.

Die Effizienz und Genauigkeit von Routingalgorithmen hat direkte Auswirkungen auf verschiedene Bereiche:

- **Transport und Verkehr:** Optimierte Routen reduzieren Fahrzeiten, Treibstoffverbrauch und CO₂-Emissionen [2]
- **Lieferdienste:** Schnellere Zustellungen durch optimierte Touren erhöhen die Kundenzufriedenheit
- **Notfalldienste:** Schnellstmögliche Routen für Rettungskräfte können Leben retten
- **Stadtplanung:** Analyse von Verkehrsflüssen zur Optimierung der Infrastruktur

Die Basis für präzise Routingberechnungen sind qualitativ hochwertige Geodaten. Mit dem Aufkommen von OpenStreetMap (OSM) als offene, gemeinschaftlich erstellte geografische Datenbank [3] steht heute ein umfangreicher und kontinuierlich aktualisierter Datensatz zur Verfügung, der als Grundlage für Routinganwendungen genutzt werden kann. Die besondere Stärke von OSM-Daten liegt in ihrer Zugänglichkeit, Aktualität und der detaillierten Erfassung von Straßeneigenschaften wie Geschwindigkeitsbegrenzungen oder Einbahnstraßen [4].

Trotz der Fortschritte im Bereich der Routingalgorithmen stellt die effiziente Berechnung optimaler Routen in großen Straßennetzen weiterhin eine algorithmische Herausforderung dar. Besonders bei Echtzeit-Anwendungen mit mobilen Endgeräten oder Webservern, die zahlreiche gleichzeitige Anfragen bewältigen müssen, ist die Optimierung der Berechnungsgeschwindigkeit von entscheidender Bedeutung [5].

1.2 Zielsetzung

Die vorliegende Projektarbeit verfolgt mehrere zusammenhängende Ziele zur Erforschung und Visualisierung von Routingalgorithmen auf realen Geodaten:

1.2.1 Entwicklung einer webbasierten Anwendung

Primäres Ziel ist die Konzeption und Implementierung einer interaktiven Webanwendung zur anschaulichen Darstellung verschiedener Routingalgorithmen. Diese soll insbesondere:

- Die Ausführung und Visualisierung klassischer Routingalgorithmen (Breitensuche, Dijkstra, A*) ermöglichen
- Die Schrittweise Nachvollziehbarkeit der Algorithmen durch animierte Visualisierung gewährleisten
- Eine nutzerfreundliche Oberfläche zur interaktiven Auswahl von Start- und Zielpunkten bieten

1.2.2 Implementierung und Optimierung von Routingalgorithmen

Ein weiteres Kernziel besteht in der effizienten Implementierung und systematischen Optimierung von Routingalgorithmen für den Einsatz auf realen Straßennetzen:

- Anwendung grundlegender Routingalgorithmen (Breitensuche, Dijkstra, A*)
- Entwicklung und Umsetzung eines Graphenkontraktionsalgorithmus zur Beschleunigung der Routenberechnung in großen Graphen
- Optimierung der Algorithmen hinsichtlich ihrer Laufzeit
- Vergleichende Evaluierung der Algorithmen bezüglich Effizienz und Lösungsqualität

1.2.3 Simulation und Visualisierung

Die entwickelten Algorithmen sollen auf realen OpenStreetMap-Daten simuliert und visualisiert werden, um:

- Die praktische Anwendbarkeit der Algorithmen unter realistischen Bedingungen zu demonstrieren
- Die Auswirkungen verschiedener Graphenstrukturen (städtisch, ländlich, Autobahnnetz) auf die Routenberechnung zu untersuchen
- Die Effizienz der implementierten Optimierungen in verschiedenen Szenarien zu evaluieren
- Entwicklung einer schrittweisen Simulation zur Veranschaulichung der internen Arbeitsweise der Algorithmen

1.3 Aufbau der Arbeit

Die vorliegende Projektarbeit ist in mehrere aufeinander aufbauende Kapitel gegliedert:

Kapitel 2: Grundlagen und Hintergründe führt in die theoretischen Konzepte ein, die für das Verständnis der Projektarbeit wesentlich sind. Hierzu zählen Koordinatensysteme, insbesondere WGS84, die Struktur und Besonderheiten von OpenStreetMap-Daten, eine Einführung in OpenLayers als Visualisierungsbibliothek sowie grundlegende Konzepte der Graphentheorie. Darüber hinaus werden die implementierten Routingalgorithmen theoretisch vorgestellt und verschiedene Distanzmetriken erläutert.

Kapitel 3: Implementierung beschreibt detailliert die technische Umsetzung des Systems. Dies umfasst die Datenbeschaffung und -verarbeitung mittels Overpass API, den Aufbau der verwendeten Graphenstruktur sowie die implementierten Algorithmen zur Graphenkontraktion und Wegfindung. Besonderes Augenmerk liegt auf den entwickelten Optimierungsverfahren und der Behandlung von Sonderfällen. Zudem wird die Implementierung der webbasierten Benutzeroberfläche erläutert und Funktionalitäten aufgezeigt.

Kapitel 4: Evaluation und Diskussion präsentiert die Ergebnisse umfangreicher Tests zur Bewertung der implementierten Algorithmen. Dies umfasst Laufzeitanalysen, Vergleiche zwischen optimierten und nicht-optimierten Varianten sowie die Bewertung der Graphenkontraktion anhand verschiedener Datensätze. Darüber hinaus werden die Stärken und Schwächen der unterschiedlichen Algorithmen in verschiedenen Szenarien diskutiert.

Kapitel 5: Zusammenfassung und Ausblick gibt Hinweise auf mögliche zukünftige Erweiterungen und Verbesserungen des Systems.

2 Grundlagen und Hintergründe

2.1 Koordinaten

In der Geodäsie und Kartographie spielen Koordinatensysteme eine zentrale Rolle bei der präzisen Beschreibung von Positionen auf der Erdoberfläche [6]. Die Herausforderung besteht darin, die annähernd sphärische Form der Erde auf ein mathematisch handhabbares Modell abzubilden [7].

2.1.1 WGS84

Das World Geodetic System 1984 (WGS84) ist ein geodätisches Referenzsystem, das als globaler Standard für die satellitengestützte Navigation und Kartographie dient [8]. Es definiert ein dreidimensionales Koordinatensystem mit dem Erdmittelpunkt als Ursprung und verwendet ein spezifisches Referenzellipsoid zur Annäherung der Erdoberfläche [9].

2.1.1.1 Aufbau und Bedeutung

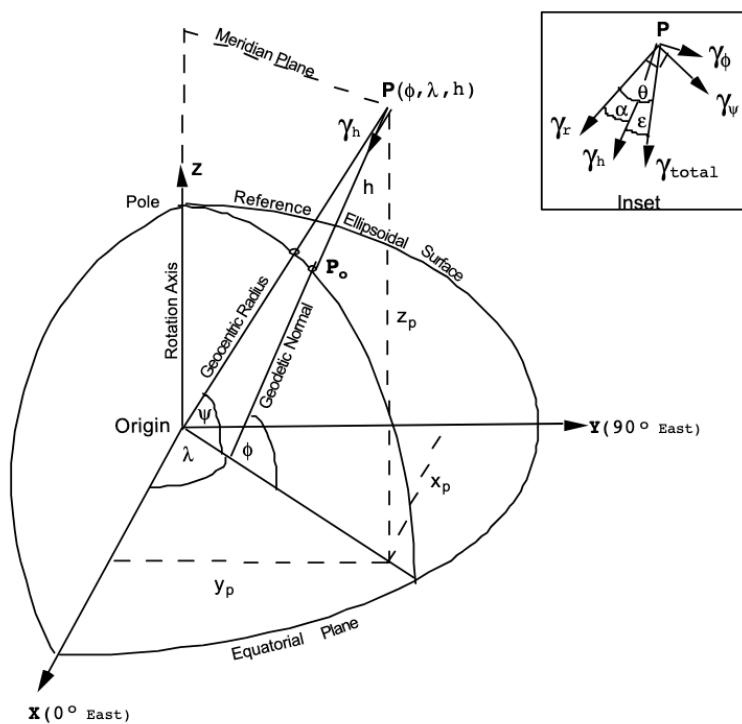


Abbildung 2.1: WGS84 Koordinatensystem [9]

Wie in Abbildung 2.1 dargestellt, beschreibt WGS84 Positionen durch [8]:

- **Breitengrad (ϕ):** Winkel zwischen Äquatorebene und Lotrichtung (-90° bis +90°)
- **Längengrad (λ):** Winkel östlich des Nullmeridians (0° bis 360° oder ±180°)
- **Höhe (h):** Vertikaler Abstand von der Ellipsoidoberfläche

Das System basiert auf einem Referenzellipsoid mit folgenden Hauptparametern [9]:

- **Große Halbachse (a):** 6.378.137,0 m
- **Abplattung (1/f):** 1/298,257223563

2.1.1.2 Relevanz für OpenStreetMap

Für die Arbeit mit OpenStreetMap-Daten ist WGS84 von besonderer Bedeutung [10, 11, 12], da:

- alle OSM-Koordinaten in diesem System gespeichert werden
- es die Basis für präzise Distanzberechnungen bildet
- es weltweite Kompatibilität gewährleistet

2.2 OpenStreetMap

OpenStreetMap (OSM) ist ein kollaboratives Projekt zur Erstellung einer frei verfügbaren Weltkarte [3]. Das Projekt wurde 2004 gegründet und hat sich seitdem zu einer der wichtigsten Quellen für geografische Daten entwickelt [4, 3]. Die Besonderheit von OSM liegt in seinem Community-basierten Ansatz, bei dem Benutzer weltweit Kartendaten erfassen, aktualisieren und verbessern können.

2.2.1 Geschichte und Entwicklung

Die Entwicklung von OpenStreetMap wurde maßgeblich durch die eingeschränkte Verfügbarkeit und hohen Kosten proprietärer Geodaten motiviert. Seit seiner Gründung hat das Projekt einen exponentiellen Wachstum sowohl in der Datenmenge als auch in der Anzahl der aktiven Beiträger verzeichnet [4, 3]. Diese breite Beteiligung ermöglicht eine kontinuierliche Verbesserung und Aktualisierung der Kartendaten.

2.2.2 Datenstruktur

OpenStreetMap verwendet ein hierarchisches Datenmodell bestehend aus drei Grundelementen [10]:

- **Nodes (Knoten):**
 - Repräsentieren einzelne Punkte auf der Erdoberfläche
 - Definiert durch Längen- und Breitengrad
 - Können alleinstehend Points of Interest (POIs) darstellen
 - Bilden die Grundlage für komplexere Strukturen
- **Ways (Wege):**

- Verbinden mehrere Nodes zu linearen Strukturen
 - Repräsentieren Straßen, Wege, Flüsse oder Gebäudeumrisse
 - Können offen (Straßen) oder geschlossen (Gebäude) sein
 - Enthalten Richtungsinformationen durch Node-Reihenfolge
- **Relations (Beziehungen):**
 - Gruppieren mehrere Elemente zu logischen Einheiten
 - Definieren komplexe Strukturen wie Busrouten oder Verwaltungsgrenzen
 - Ermöglichen die Modellierung von Hierarchien und Abhängigkeiten

2.2.3 Tagging-System

Das OSM-Tagging-System ist ein flexibles System zur Beschreibung von Kartenelementen [10]:

- **Schlüssel-Wert-Paare:** Tags bestehen aus einem Schlüssel und einem zugehörigen Wert
- **Standardisierung:** Häufig verwendete Tags sind in der Community etabliert und dokumentiert
- **Erweiterbarkeit:** Neue Tags können bei Bedarf eingeführt werden

Einige Beispiele für wichtige Tags im Kontext der Routenberechnung:

- `highway=*`: Klassifiziert Straßentypen (motorway, primary, residential, etc.)
- `oneway=*`: Kennzeichnet Einbahnstraßen
- `maxspeed=*`: Definiert Geschwindigkeitsbegrenzungen

2.2.4 Datenqualität und -vollständigkeit

Die Qualität der OSM-Daten variiert in Abhängigkeit von der Region und der Art der Information. In urbanen Gebieten sind die Daten typischerweise sehr detailliert und aktuell, was auf die hohe Dichte an aktiven Beitragenden zurückzuführen ist. Ländliche Regionen weisen dagegen oft eine geringere Detailtiefe auf, wobei die grundlegende Infrastruktur wie Hauptstraßen und wichtige Landmarken in der Regel gut dokumentiert ist.[13]

Ein signifikanter Vorteil des OSM-Projekts liegt in der schnellen Aktualisierung der Daten durch die lokale Community. Änderungen in der realen Welt, wie neue Straßen oder geänderte Verkehrsführungen, werden oft zeitnah in der Kartendatenbank erfasst. Die Qualitätssicherung erfolgt dabei durch eine Kombination aus communitybasierter Kontrolle und automatisierten Prüfverfahren, die Inkonsistenzen und potenzielle Fehler identifizieren.[13]

2.2.5 Overpass API

Die Overpass API ist eine leistungsfähige Schnittstelle für den gezielten Zugriff auf OSM-Daten [14]. Sie ermöglicht:

- Komplexe räumliche Abfragen
- Filterung nach spezifischen Tags und Attributen

- Effiziente Verarbeitung großer Datenmengen

2.2.5.1 Abfragetypen

Die API unterstützt verschiedene Abfragestrategien [14]:

- **Area:** Abfrage eines definierten geografischen Bereichs
- **Around:** Suche in einem bestimmten Radius um einen Punkt
- **Way:** Abfrage spezifischer Straßen oder Wege
- **Node:** Suche nach einzelnen Punkten

2.3 OpenLayers

OpenLayers ist eine Open-Source JavaScript-Bibliothek, die eine umfangreiche Basis für die Entwicklung von interaktiven Kartenanwendungen im Web bietet [15]. Sie wurde 2006 ursprünglich von MetaCarta entwickelt und hat sich seitdem zu einem der führenden Frameworks für webbasierte Geoinformationssysteme (WebGIS) entwickelt. Im Gegensatz zu proprietären Alternativen ist OpenLayers vollständig quelloffen und ermöglicht die Integration verschiedener Geodatenquellen, darunter auch die in dieser Projektarbeit verwendete OpenStreetMap, aber auch weitere wie Google Maps oder Bing Maps [16].

Die Bibliothek unterstützt zahlreiche Kartenformate und -protokolle und zeichnet sich durch ihre Flexibilität und Erweiterbarkeit aus. Diese Eigenschaften machen OpenLayers besonders geeignet für komplexe GIS-Anwendungen und Visualisierungen von räumlichen Daten im Web-Kontext [17].

2.3.1 Architektur und Aufbau

OpenLayers folgt einem objektorientierten, modularen Aufbau, der auf verschiedenen Kernkonzepten basiert:

2.3.1.1 Basiskomponenten

Die Architektur von OpenLayers besteht aus mehreren Grundkomponenten:

- **Map:** Das zentrale Kontrollelement, das alle Kartenelemente verwaltet und Benutzerinteraktionen koordiniert. Die Map-Komponente ist verantwortlich für die Darstellung der Karte sowie die Handhabung von Events und Interaktionen [17].
- **View:** Definiert die Ansichtseigenschaften der Karte wie Zentrum, Zoom-Level und Projektion. Die View-Komponente ermöglicht die kontrollierte Navigation innerhalb der dargestellten Geodaten [16].
- **Layer:** Repräsentiert eine einzelne Datenschicht auf der Karte. OpenLayers unterstützt verschiedene Layer-Typen wie Tile (Kacheln), Vector (Vektordaten), Image (Einzelbilder) und Group (Gruppierung von Layern) [15].

- **Source:** Definiert die Datenquelle für einen Layer und bestimmt, wie Daten geladen und dargestellt werden. Quellen können lokale Dateien, Serverdienste oder dynamisch generierte Daten sein [17].

2.3.1.2 Objekthierarchie

Die OpenLayers-Architektur basiert auf einer hierarchischen Objektstruktur:

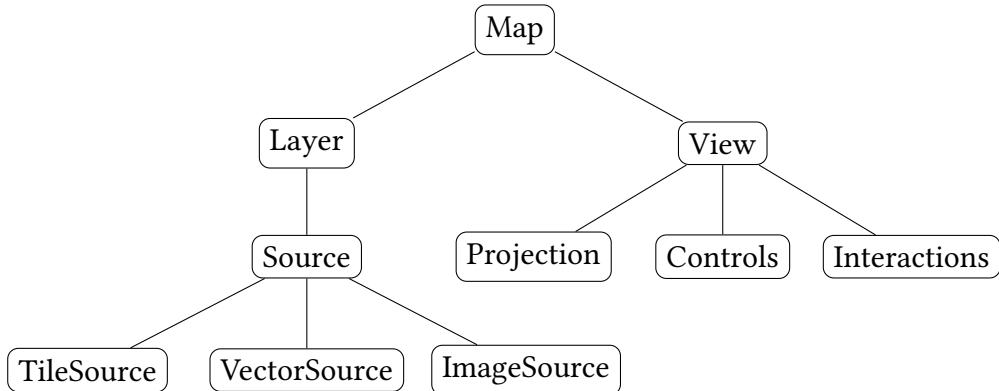


Abbildung 2.2: Vereinfachte Objekthierarchie in OpenLayers [Eigene Darstellung]

Diese Struktur ermöglicht eine klare Trennung von Zuständigkeiten und fördert die Wiederverwendbarkeit von Komponenten. Jede Komponente kann unabhängig konfiguriert und erweitert werden, was die Anpassbarkeit der Bibliothek erhöht [16, 17, 15].

2.3.2 Layer-Konzept und Datenvisualisierung

Das Layer-Konzept ist fundamental für die Darstellung von räumlichen Daten in OpenLayers. Es ermöglicht die Organisation und Visualisierung verschiedener Geodatentypen in übereinander liegenden Schichten [17].

2.3.2.1 Tile Layer

Tile Layer (Kachelkarten) sind optimiert für die effiziente Darstellung großer Kartendatensätze:

- Basieren auf vorgerenderten Bildkacheln (typischerweise 256x256 Pixel)
- Ermöglichen schnelles Laden und Rendering durch bedarfsgerechtes Nachladen
- Unterstützen verschiedene Zoomstufen durch Pyramidenstruktur
- Typische Verwendung für Basiskarten wie OpenStreetMap oder satellitenbilder

2.3.2.2 Vector Layer

Vector Layer ermöglichen die Darstellung und Interaktion mit Geometrien auf Objektebene [15]:

- Unterstützen Punkte, Linien, Polygone und komplexe Geometrien
- Bieten dynamische Stilisierungsmöglichkeiten (Farbe, Größe, Symbole)

- Ermöglichen Benutzerinteraktionen (Selektion, Drag & Drop, Editieren)

2.3.2.3 Styling und Symbolisierung

OpenLayers bietet umfangreiche Möglichkeiten zur visuellen Gestaltung von Vektordaten[15]:

- **Style:** Definiert das visuelle Erscheinungsbild von Features durch Eigenschaften wie Farbe, Linienstärke, Füllung
- **StyleFunction:** Ermöglicht dynamische, attributbasierte Stilisierung
- **Text Styling:** Unterstützt Beschriftungen mit konfigurierbaren Texteigenschaften

2.3.2.4 Events und Interaktionen

Die Interaktivität der Karte wird durch ein umfassendes Event-System und vordefinierte Interaktionen gewährleistet [15]:

- **Events:** Ermöglichen die Reaktion auf Benutzeraktionen wie Klicks, Mausbewegungen oder Zoom
- **Controls:** Bieten Standardelemente zur Kartensteuerung (Zoom, Attribution, Vollbild)

2.4 Graphentheorie

Ein Graph $G = (V, E)$ ist ein mathematisches Konstrukt, das aus einer Menge von Knoten (Vertices) V und einer Menge von Kanten (Edges) E besteht. Kanten verbinden jeweils zwei Knoten und können gerichtet oder ungerichtet sein. Im Kontext der Routenberechnung repräsentieren Knoten typischerweise Kreuzungen oder markante Punkte, während Kanten die Straßenverbindungen zwischen diesen Punkten darstellen. [18]

2.4.1 Graphentypen und Eigenschaften

In der Routenberechnung spielen verschiedene Arten von Graphen eine zentrale Rolle [18, 19]. Die grundlegenden Typen unterscheiden sich in der Art ihrer Kantenbeziehungen und deren Eigenschaften:

- **Gerichtete Graphen:** Die Kanten besitzen eine definierte Richtung und können nur in diese Richtung traversiert werden [18]. Dies ist besonders relevant für Einbahnstraßen im Kontext der Straßennavigation.
- **Ungerichtete Graphen:** Die Kanten können in beide Richtungen durchlaufen werden und repräsentieren bidirektionale Verbindungen. Dies entspricht dem häufigsten Fall im Straßennetz, wo Straßen in beide Richtungen befahrbar sind [18].
- **Gewichtete Graphen:** Den Kanten sind numerische Werte (Gewichte) zugeordnet, die beispielsweise Distanzen, Fahrzeiten oder andere Metriken darstellen. Diese Gewichte sind fundamental für die Optimierung von Routen [18, 20].

Dabei ist zu beachten, dass diese Eigenschaften nicht exklusiv sind - ein Graph kann gleichzeitig gerichtet und gewichtet sein [18]. Im Kontext der Straßennavigation ist dies besonders relevant,

wenn beispielsweise Einbahnstraßen mit unterschiedlichen Längen oder Fahrzeiten modelliert werden müssen. Die Kombination dieser Eigenschaften ermöglicht eine präzise Modellierung realer Verkehrsnetze [3].

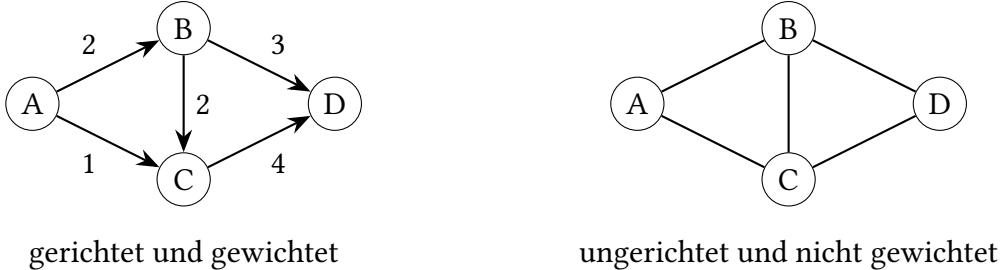


Abbildung 2.3: Beispiele für Graphentypen in der Routenberechnung [Eigene Darstellung]

In Abbildung 2.3 sind Beispiele für verschiedene Graphentypen dargestellt. Auf der linken Seite ist ein gerichteter und gewichteter Graph zu sehen, der die Beziehungen zwischen verschiedenen Knoten durch gerichtete Kanten mit Gewichten modelliert. Auf der rechten Seite ist ein ungerichteter und nicht gewichteter Graph abgebildet, der bidirektionale Verbindungen zwischen den Knoten darstellt.

2.4.2 Knotengrad

Eine fundamentale Eigenschaft eines Knotens in einem Graphen ist sein Grad (Degree). Der Grad $d(v)$ eines Knotens v gibt die Anzahl der mit diesem Knoten verbundenen Kanten an [18, 21].

In einem ungerichteten Graphen ist der Grad eines Knotens einfach die Gesamtzahl der an ihm anliegenden Kanten. Formal ausgedrückt:

$$d(v) = |\{e \in E : v \in e\}| \quad (2.1)$$

Bei gerichteten Graphen unterscheidet man hingegen zwischen dem Eingangsgrad (in-degree) und dem Ausgangsgrad (out-degree) eines Knotens:

- Der Eingangsgrad $d_{in}(v)$ eines Knotens v ist die Anzahl der Kanten, die in v enden.
- Der Ausgangsgrad $d_{out}(v)$ eines Knotens v ist die Anzahl der Kanten, die von v ausgehen.
- Der Gesamtgrad $d(v)$ ist die Summe aus Eingangs- und Ausgangsgrad: $d(v) = d_{in}(v) + d_{out}(v)$.

Im Kontext von Straßennetzen hat der Knotengrad eine besondere Bedeutung:

- Knoten mit einem Grad von 1 sind typischerweise Sackgassen oder Endpunkte.
- Knoten mit einem Grad von 2 repräsentieren Zwischenpunkte auf einer Straße ohne Abzweigungen.
- Knoten mit einem Grad größer gleich 3 stellen Kreuzungen oder Verzweigungen dar, an denen mehrere Straßen zusammenlaufen.

Besonders relevant für die in Abschnitt 3.2 beschriebene Graphenkontraktion [22] ist die Identifizierung von Knoten mit einem Grad von 2, da diese keine echten Entscheidungspunkte

darstellen und unter bestimmten Bedingungen aus dem Graphen entfernt werden können, ohne dessen topologische Struktur zu verändern.

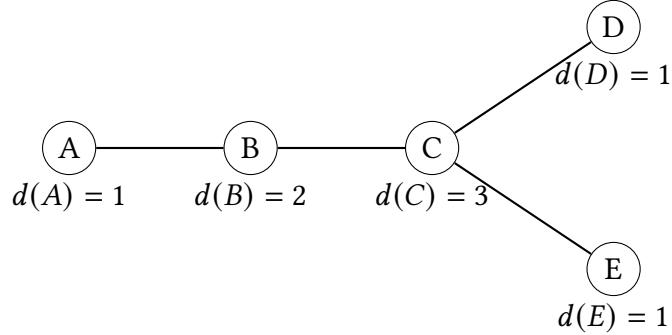


Abbildung 2.4: Beispiel für Knotengrade in einem ungerichteten Graphen [Eigene Darstellung]

In Abbildung 2.4 ist ein Beispiel für verschiedene Knotengrade dargestellt. Der Knoten C hat einen Grad von 3 und stellt eine echte Verzweigung dar, während der Knoten B einen Grad von 2 hat und potentiell durch Kontraktion entfernt werden könnte. Die Knoten A , D und E haben einen Grad von 1 und repräsentieren einen Endpunkt.

2.4.3 Algorithmen zur Wegfindung

Die Wegfindung in Graphen erfolgt durch spezialisierte Algorithmen, die sich in ihrer Herangehensweise und Effizienz unterscheiden.

2.4.3.1 Breitensuche

Die Breitensuche (Breadth-First Search, BFS) ist ein grundlegender Algorithmus zur systematischen Durchsuchung eines Graphen [23]. Die fundamentale Idee basiert auf der ebenenweisen Exploration des Graphen, bei der zunächst alle Knoten einer Entfernungsstufe vollständig untersucht werden, bevor zur nächsten Ebene übergegangen wird.

Für jeden Knoten v wird während der Suche ein Zustand verwaltet:

$$s(v) = \begin{cases} 0 & \text{nicht besucht} \\ 1 & \text{in Warteschlange} \\ 2 & \text{abgeschlossen} \end{cases} \quad (2.2)$$

Der Algorithmus verwendet eine FIFO-Warteschlange (First-In-First-Out) Q , die folgende Eigenschaften aufweist:

- Neue Knoten werden am Ende der Warteschlange eingefügt
- Knoten werden in der Reihenfolge ihrer Entdeckung verarbeitet
- Die Reihenfolge gewährleistet die ebenenweise Exploration

Die Breitensuche garantiert das Auffinden des kürzesten Weges in ungewichteten Graphen durch ihre systematische Arbeitsweise. Da die Knoten stets in der Reihenfolge ihrer Entfernung vom Startknoten entdeckt werden, enthält jeder gefundene Weg automatisch die minimale Anzahl

an Kanten. Diese Eigenschaft, kombiniert mit der optimalen Laufzeitkomplexität von $O(|V| + |E|)$ für ungewichtete Graphen, macht die Breitensuche zu einem fundamentalen Algorithmus.

Im Kontext der Routenberechnung ist die Breitensuche besonders dann relevant, wenn alle Kanten gleiche oder keine Gewichte haben. [23]

2.4.3.2 Dijkstra-Algorithmus

Der von Edsger W. Dijkstra entwickelte Algorithmus [24, 18] ist ein Verfahren zur Bestimmung kürzester Pfade in gewichteten Graphen. Die fundamentale Idee basiert auf der schrittweisen Exploration des Graphen, ausgehend vom Startknoten, wobei in jedem Schritt der Knoten mit den geringsten Gesamtkosten erweitert wird.

Für jeden Knoten v wird ein Distanzwert $d(v)$ verwaltet:

$$d(v) = \min_{u \in N} \{d(u) + w(u, v)\} \quad (2.3)$$

wobei:

- $d(v)$ die kürzeste bekannte Distanz vom Startknoten zum Knoten v ist
- u den aktuell betrachteten Vorgängerknoten auf dem Weg zum Knoten v bezeichnet
- N die Menge der bereits besuchten Nachbarknoten darstellt
- $w(u, v)$ das Gewicht der Kante zwischen dem Vorgängerknoten u und dem aktuellen Knoten v bezeichnet

Der Algorithmus garantiert die Optimalität der gefundenen Pfade unter der Voraussetzung nicht-negativer Kantengewichte. Dies wird durch das Prinzip der Optimalitätssubstruktur erreicht: Jeder Teilstumpf eines kürzesten Weges ist selbst ein kürzester Weg zwischen seinen Endpunkten [24].

Im Kontext der Routenberechnung ist der Dijkstra-Algorithmus von besonderer Relevanz, da er die Grundlage für viele fortgeschrittene Routing-Algorithmen bildet. Seine Optimalität bei bekannten Kantengewichten macht ihn zu einem zuverlässigen Werkzeug für die Wegfindung, während seine effiziente Implementierbarkeit den praktischen Einsatz auch in ressourcenbeschränkten Umgebungen ermöglicht. [24]

2.4.3.3 A*-Algorithmus

Der A*-Algorithmus ist eine Erweiterung des Dijkstra-Algorithmus um eine heuristische Komponente [25, 20]. Die Besonderheit des Algorithmus liegt in der Kombination aus tatsächlichen Wegekosten und einer Schätzung der verbleibenden Kosten zum Ziel.

Für jeden Knoten n wird ein Bewertungswert $f(n)$ berechnet:

$$f(n) = g(n) + h(n) \quad (2.4)$$

wobei:

- $g(n)$ die tatsächlichen Kosten vom Startknoten bis zum aktuellen Knoten n repräsentiert

- $h(n)$ eine Heuristik-Funktion ist, die die geschätzten Kosten vom Knoten n zum Zielknoten schätzt

Eine Heuristik $h(n)$ wird als *zulässig* bezeichnet, wenn sie die tatsächlichen Kosten zum Ziel nie überschätzt. Dies ist eine zentrale Eigenschaft für die Optimalität des Algorithmus [25]. Im Kontext der geografischen Navigation wird typischerweise die Luftliniendistanz als Heuristik verwendet, da diese:

- garantiert zulässig ist (der kürzeste Weg zwischen zwei Punkten ist immer die Luftlinie)
- einfach und effizient zu berechnen ist
- eine gute Balance zwischen Genauigkeit und Berechnungsaufwand bietet

Der A*-Algorithmus wählt in jedem Schritt den Knoten mit dem kleinsten $f(n)$ -Wert zur Expansion aus. Dies führt zu einer zielgerichteten Suche durch die gezielte Bevorzugung von Knoten, die in Richtung des Ziels liegen, basierend auf der heuristischen Komponente. Dabei werden dennoch alle potenziell optimalen Pfade berücksichtigt, was die Optimalität des Algorithmus gewährleistet. Im Vergleich zum Dijkstra-Algorithmus arbeitet A* deutlich effizienter, da durch die zielgerichtete Suche wesentlich weniger irrelevante Knoten expandiert werden müssen. [25, 20]

Die Effizienz des A*-Algorithmus wird maßgeblich durch die Qualität der verwendeten Heuristik bestimmt. Eine ideale Heuristik sollte dabei mehrere Kriterien erfüllen: Sie muss möglichst präzise die tatsächlichen Kosten zum Ziel approximieren, darf diese jedoch nie überschätzen, um die Zulässigkeit der Heuristik zu gewährleisten. Gleichzeitig sollte die Berechnung der Heuristik mit minimalem Rechenaufwand möglich sein, da sie für jeden expandierten Knoten berechnet werden muss. [25, 20]

2.4.4 Graphenkontraktion

Wolle [22] beschreibt die theoretischen Grundlagen der Edge Contraction (nachfolgend Graphenkontraktion genannt) und deren Bedeutung für die algorithmische Effizienz. Die Kontraktion bewahrt dabei die zugrunde liegenden Eigenschaften des Graphen, während sie gleichzeitig die Komplexität für nachfolgende Berechnungen signifikant reduziert. Von besonderer Bedeutung ist, dass die Optimalität der Routenberechnung durch die Kontraktion nicht beeinträchtigt wird, da alle relevanten Entscheidungspunkte (Kreuzungen und Verzweigungen) im Graphen erhalten bleiben.

Die konkrete Implementierung des, in dieser Projektarbeit entwickelten, Graphenkontraktionsalgorithmus sowie dessen spezifische Optimierungen werden detailliert in Abschnitt 3.2 behandelt.

2.4.4.1 Beispiel Graphenkontraktion

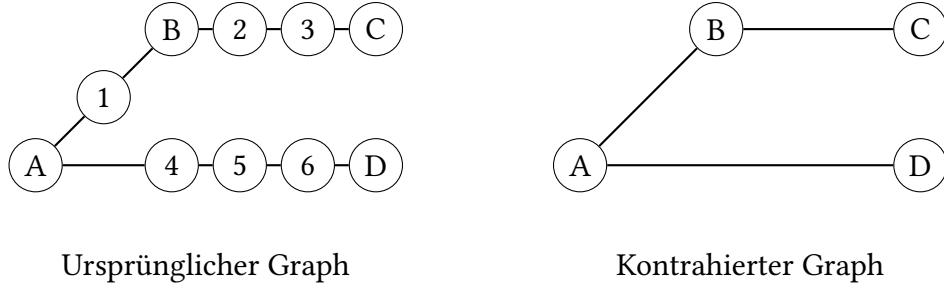


Abbildung 2.5: Beispiel einer Graphenkontraktion [Eigene Darstellung]

In Abbildung 2.5 ist eine exemplarische Graphenkontraktion dargestellt (siehe Abschnitt 3.2.2 für ein vollständiges Beispiel der in dieser Projektarbeit implementierten Graphenkontraktion). Auf der Linken Seite ist der ursprüngliche Graph mit allen Zwischenknoten zu sehen, während auf der Rechten Seite der kontrahierte Graph mit zusammengefassten Kanten abgebildet ist.

Die Knoten mit den Buchstaben A, B, C, D stellen dabei Verzweigungen dar. Die Knoten mit den Zahlen $1, 2, 3, 4, 5, 6$ sind Zwischenknoten, die durch die Kontraktion zusammengefasst werden können. Die Kanten des kontrahierten Graphen repräsentieren dabei die kürzesten Wege ohne Unterbrechungen zwischen zwei Verzweigungen, wobei die ursprünglichen Zwischenknoten entfernt wurden.

2.4.5 Metriken

In der Graphentheorie spielen Metriken eine fundamentale Rolle bei gewichteten Graphen hinsichtlich der Angabe der Kosten einer Kante zwischen zwei Knoten [19, 18, 26]. Im Kontext der geografischen Routenberechnung sind diese Metriken besonders relevant, da sie die Basis für die Bestimmung optimaler Wege bilden.

2.4.5.1 Klassische Distanzmetriken

Die am häufigsten verwendeten Metriken für die Distanzberechnung sind:

Euklidische Distanz Die euklidische Distanz d_E zwischen zwei Punkten $p_1(x_1, y_1)$ und $p_2(x_2, y_2)$ in der Ebene wird definiert als [27, 6]:

$$d_E(p_1, p_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (2.5)$$

Diese Metrik entspricht der intuitiven Vorstellung einer "Luftlinie" zwischen zwei Punkten, ist jedoch für Berechnungen auf der Erdoberfläche nur bedingt geeignet.

Manhattan-Distanz Die Manhattan-Distanz d_M , auch bekannt als L1-Metrik oder City-Block-Distanz, berechnet sich als [27, 20]:

$$d_M(p_1, p_2) = |x_2 - x_1| + |y_2 - y_1| \quad (2.6)$$

Diese Metrik ist besonders in städtischen Umgebungen mit rechtwinkligen Straßennetzen relevant, spiegelt jedoch die Realität auf der Erdoberfläche nicht adäquat wider.

2.4.5.2 Sphärische Distanzberechnung

Für die präzise Distanzberechnung auf der Erdoberfläche müssen spezialisierte Metriken verwendet werden, die der Kugelform der Erde Rechnung tragen [7]. Die bedeutendsten sind:

Großkreisdistanz Die Großkreisdistanz repräsentiert die kürzeste Verbindung zwischen zwei Punkten auf einer Kugel und wird durch den Schnitt einer Ebene durch den Kugelmittelpunkt definiert [6]. Diese theoretisch exakte Metrik bildet die Grundlage für praktische Implementierungen.

Haversine-Formel Die Haversine-Formel ist eine Methode zur Berechnung der Großkreisdistanz [9, 28, 29]. Für zwei Punkte mit den Koordinaten (ϕ_1, λ_1) und (ϕ_2, λ_2) berechnet sich die Distanz d als:

$$d = 2R \arcsin \left(\sqrt{\sin^2 \left(\frac{\phi_2 - \phi_1}{2} \right) + \cos(\phi_1) \cos(\phi_2) \sin^2 \left(\frac{\lambda_2 - \lambda_1}{2} \right)} \right) \quad (2.7)$$

wobei:

- ϕ_1, ϕ_2 die geografischen Breiten (Latitude) der Punkte 1 und 2 sind
- λ_1, λ_2 die geografischen Längen (Longitude) der Punkte 1 und 2 sind
- R der mittlere Erdradius ist

3 Implementierung

3.1 Datenbeschaffung

Im Rahmen der Projektarbeit wurden OSM-Daten verwendet. Diese Daten sind frei verfügbar und können verschieden abgefragt werden. In dieser Projektarbeit wurde die Overpass API verwendet, um die Daten abzufragen.

Es wurden Straßen- und Wegdaten, aus verschiedenen Bereichen Deutschlands, verwendet. Dabei wurde darauf geachtet verschiedene Straßentypen und Straßenverbindungen zu verwenden. So wurden beispielsweise Autobahnen, Bundesstraßen, Landstraßen und innerörtliche Straßen verwendet. Weiterhin wurden auch Wegdaten wie beispielsweise die in der Stuttgarter Wilhelma verwendet, um weitere Anwendungsbereiche abzudecken. Ebenfalls wurde darauf geachtet, dass unterschiedliche Gebiete abgedeckt werden, so wurden sowohl ländliche als auch städtische Gebiete abgedeckt.

Da nicht alle Daten auf einmal heruntergeladen werden können auf Grund von Größenlimitationen der Overpass API, wurden mehrere kleinere Datensätze abgefragt. So existieren zum Beispiel Datensätze bestehend aus primär ungerichteten Verbindungen wie die Wilhelma, oder Datensätze bestehend aus primär gerichteten Verbindungen wie Autobahndaten. Ebenfalls wurden Datensätze, die beide Arten von Verbindungen enthalten, wie die Stadt Stuttgart, abgefragt, welche eine städtische Umgebung mit gemischten Verbindungsarten darstellt. Weitere Datensätze beinhalten eine Mischung aus ländlichen und städtischen Gebieten, um eine möglichst breite Abdeckung zu gewährleisten. Zu solchen Datensätzen zählen beispielsweise die Daten von Bundesländern wie Baden-Württemberg.

3.1.1 Overpass API

Die Overpass API ist eine leistungsstarke Schnittstelle zum Abrufen spezifischer OSM-Daten. Sie verwendet eine eigene Abfragesprache, die Overpass QL, die es ermöglicht, komplexe Filterungen und Verknüpfungen von OSM-Daten durchzuführen.

3.1.1.1 Grundlegende Abfragestruktur

Eine typische Overpass-Abfrage besteht aus mehreren Komponenten:

Listing 3.1: Grundlegende Struktur einer Overpass QL Abfrage

```
[out:json]          // Ausgabeformat festlegen
[bbox:sued,west,nord,ost]; // Geografischer Bereich
// Weitere Abfragebefehle
out body;         // Ausgabe der Ergebnisse
```

3.1.1.2 Beispielabfragen zur Datenbeschaffung

In diesem Abschnitt werden verschiedene Beispiele für Overpass-Abfragen präsentiert, die zur Beschaffung von OSM-Daten herangezogen werden können und in dieser Projektarbeit genutzt wurden, um die Daten zu beschaffen.

Autobahndaten Im ersten Beispiel der Autobahndaten wird auf verschiedene Abfragemethoden näher eingegangen. Dabei werden auch die Unterschiede und Besonderheiten der jeweiligen Methode betrachtet.

Nur Autobahndaten Mit dem folgenden Overpass QL Befehl können Autobahnen in einem bestimmten geografischen Bereich abgefragt werden. Bei dieser einfachen Abfragemethode ist zu beachten, dass Autobahnen in OpenStreetMap typischerweise durch zwei separate, gerichtete Wege repräsentiert werden - jeweils einen für jede Fahrtrichtung. Diese Darstellungsweise erschwert die spätere Implementierung von Wendemöglichkeiten, da die Verbindung zwischen den beiden Richtungsfahrbahnen nicht explizit modelliert ist.

Listing 3.2: Abfrage von Autobahndaten in einem bestimmten Bereich

```
[out:json]
[bbox:48.6888,9.0201,48.8600,9.2777];
area["ISO3166-2" ~ "DE-BW"] -> .SA;

// 1. Autobahnen erfassen
way(area.SA)[highway=motorway] -> .motorways;

// 2. Ergebnisse ausgeben
(.motorways;););
(_.;>););
out body;
```

Autobahndaten mit Zubringern durch Verschachtelte Abfragen Eine erweiterte Abfragemethode ermöglicht es, Autobahnen zusammen mit ihren Zubringern und den angrenzenden Straßen zu erfassen. Diese Methode bietet den bedeutenden Vorteil, dass ausschließlich tatsächlich mit dem Autobahnnetz verbundene Straßen in den resultierenden Datensatz aufgenommen werden. Dies gewährleistet, dass keine isolierten Straßenabschnitte in den Datensatz aufgenommen werden, die nicht mit dem Autobahnnetz verbunden sind.

Allerdings weist dieser Ansatz auch einige strukturelle Limitierungen auf:

- Die Abfragetiefe muss explizit durch die Anzahl der Wiederholungen der Abfrage festgelegt werden
- Die Abfrage wird unübersichtlich, da die Logik zur Erfassung der angrenzenden Straßen mehrfach wiederholt werden muss
- Längere Wege können nicht erfasst werden, da eine bestimmte Tiefe der Abfrage festgelegt ist

Listing 3.3: Verschachtelte Abfrage für Autobahnen mit angrenzenden Straßen

```
[out:json]
[bbox:48.6888,9.0201,48.8600,9.2777];
area["ISO3166-2" ~ "DE-BW"] -> .SA;

// 1. Autobahnen und Zubringer erfassen
way(area.SA)[highway=motorway] -> .motorways;
way(area.SA)[highway=motorway_link] -> .motorway_links;

// 2. Verknüpfte Wege an den Zubringern finden
node(w.motorway_links) -> .motorway_link_nodes;
way(bn.motorway_link_nodes)
[highway~"^(primary|secondary|tertiary|trunk)(_link)?$"]
-> .connected_ways;

// 3. Wege finden, die mit den verknüpften Wege verbunden sind
node(w.connected_ways) -> .connected_nodes;
way(bn.connected_nodes)
[highway~"^(primary|secondary|tertiary|trunk)(_link)?$"]
-> .connected_ways;

// 4. Schritt 3 wiederholen
node(w.connected_ways) -> .connected_nodes;
way(bn.connected_nodes)
[highway~"^(primary|secondary|tertiary|trunk)(_link)?$"]
-> .connected_ways;

// 5. Ergebnisse zusammenführen und ausgeben
(.motorways; .motorway_links; .connected_ways; );
(..;>););
out body;
```

Autobahndaten mit Zubringern durch Around Eine alternative Herangehensweise zur Erfassung von Autobahnen und ihrem Umfeld bietet die Around-Abfrage. Diese Methode zeichnet sich durch mehrere vorteilhafte Eigenschaften aus:

- Höhere Codequalität durch Vermeidung von Redundanzen in der Abfragelogik
- Flexibilität durch einfache Anpassungsmöglichkeit des Suchradius an einem zentralen Parameter möglich (Radius erhöhen/verringern)
- Verbesserte Übersichtlichkeit durch kompakte, einmalige Formulierung der Abfrage
- Potentiell vollständigere Datenerfassung, da keine explizite Abfragetiefe festgelegt ist

Als kritischer Aspekt ist jedoch zu beachten, dass dieser Ansatz auch Straßen erfasst, die zwar innerhalb des definierten Radius liegen, aber nicht notwendigerweise mit dem Autobahnnetz verbunden sind. Dies muss entweder vorbereitet werden, oder in Kauf genommen werden. Im Rahmen dieser Projektarbeit wurde keine Nachbereitung implementiert, sodass die resultierenden Datensätze auch isolierte Straßenabschnitte enthalten können, was entsprechend beim Routenberücksichtigt werden muss.

Listing 3.4: Around-Abfrage für Autobahnen mit Umgebungsstraßen

```
[out:json]
[bbox:48.6888,9.0201,48.8600,9.2777];
area["ISO3166-2" ~ "DE-BW"] -> .SA;

// 1. Motorways und Motorway Links erfassen
way(area.SA)[highway=motorway] -> .motorways;
way(area.SA)[highway=motorway_link] -> .motorway_links;

// 2. Alle Nodes der Motorways und Motorway Links finden
way(around.motorway_links:200)[highway~"^(primary|primary_link|secondary|secondary_link|
    tertiary|trunk|trunk_link)$"] -> .surrounding;

// 3. Zusammenfugen: Motorways, Motorway Links und verbundene Wege
(.motorways; .motorway_links; .surrounding);
(_;>););
out body;
```

Wilhelma-Daten Beim Beispiel der Wilhelma-Daten können ebenfalls unterschiedliche Abfragemethoden betrachtet werden. Dabei wird jedoch speziell auf die verwendete Abfragemethode eingegangen und die Besonderheiten dieser Methode betrachtet.

Listing 3.5: Abfrage für Wilhelma Daten

```
[out:json]
[bbox:48,8,49,10];
area[name="Wilhelma"]->.searchArea;

// 1. Alle Wege in der Wilhelma (ohne Bruecken und Tunnel)
way(area.searchArea)[highway][tunnel!=yes][bridge!=yes];

// Ausgabe mit vollstaendiger Geometrie
(_;>););
out body;
```

Wie aus Listing 3.4 zu entnehmen, werden alle Wege vom Typ `highway` abgefragt jedoch mit der Einschränkung, dass keine Tunnel oder Brücken in dem Datensatz enthalten sein sollen. Dies hat den Hintergrund, dass verschiedene Wegtypen in der Wilhelma vorhanden sind, die alle korrekt in dem Datensatz enthalten sein sollen. Jedoch gibt es auch andere Wege, beziehungsweise Straßen, die in dem Bereich der Wilhelma liegen, jedoch nicht zum Wegenetz der Wilhelma gehören, wie zum Beispiel der Rosensteintunnel, der unter der Wilhelma verläuft.

Die Filterung nicht relevanter Wege kann auf unterschiedliche Weise erfolgen, beispielsweise durch Ausschließen des exakten Wegtyps des Rosensteintunnels. Dies ist jedoch nicht immer möglich, da sich der Typ des Weges in den OSM-Daten ändern kann oder ein neuer, für die Wilhelma relevanter Weg desselben Typs hinzugefügt wird, der dann ausgeschlossen wird, obwohl er relevant wäre. Alternativ bestünde die Möglichkeit, diese Logik zu invertieren, sodass ausschließlich spezifische Wegtypen in dem Datensatz enthalten sind. In diesem Fall wird eine Whitelist an Wegtypen erstellt, die in dem Datensatz enthalten sein sollen, im Gegensatz zur Blacklist an Wegtypen des vorangegangenen Beispiels, die nicht enthalten sein sollen. Auch bei diesem Ansatz besteht das Problem, dass sowohl neue Wegtypen als auch Änderungen an

bestehenden Wegtypen zu einer fehlerhaften Filterung führen können. Des Weiteren könnten auch Wege in dem Datensatz enthalten sein, die nicht relevant sind, da sie zwar im Bereich der Wilhelma liegen, aber nicht zu ihr gehören. Dies könnte beispielsweise durch eine ungenaue Abgrenzung der Wilhelma in den OSM-Daten entstehen.

Zur Problemlösung der zuvor genannten Hindernisse und zur Gewährleistung der Wartbarkeit des Abfragecodes wurde die Entscheidung getroffen, eine Filterung aller Wege durchzuführen, die in einem Tunnel oder auf einer Brücke liegen. Diese Methode erweist sich als effektiv und einfach, um die relevanten Wege zu filtern, ohne die Erstellung einer Whitelist oder Blacklist für bestimmte Wegtypen erforderlich zu machen. Darüber hinaus ist diese Methode zumindest hinsichtlich sich verändernde Wegtypen zukunftssicher, da neue Wegtypen, die in der Wilhelma relevant sind, automatisch in dem Datensatz enthalten sind, solange sie nicht in einem Tunnel oder auf einer Brücke liegen. Ein potenzielles Problem besteht jedoch, wenn in der Wilhelma ein Tunnel oder eine Brücke gebaut wird, da diese ebenfalls herausgefiltert würden.

Stuttgart-Daten Beim Beispiel der Stuttgart-Daten ist zu beachten, dass die gefundenen Wege nicht immer nur Straßen oder Wege beinhalten, sondern auch Flächen, wie beispielsweise Fußgängerzonen, als Wege von OSM interpretiert sind. Dies führt zu Problemen bei der späteren Routenberechnung, da Flächen keine klar definierte Wege beinhalten sondern lediglich einen Bereich darstellen, in welchem eine freie Bewegung stattfinden kann. Um dieses Problem zu umgehen, wurde eine Filterung auf Wege mit dem Attribut `area` ungleich `yes` durchgeführt, um nur explizite Wege zu erhalten, wie in Listing 3.7 dargestellt.

Listing 3.6: Abfrage für Stuttgart Daten

```
[out:json]
[bbox]:48.6700,9.0400,48.8800,9.3200];
area["ISO3166-2"="DE-BW"]->.BW;

// 1. Alle Wege in Stuttgart (ohne Flächen)
way(area.BW)[highway][area!="yes"];

// Ausgabe mit vollständiger Geometrie
(..;>);
out body;
```

Bundesländer-Daten Beim Beispiel der Bundesländer-Daten ist zu beachten, dass die Abfrage auf jeweils ein Bundesland beschränkt ist. Dies hat den Hintergrund, dass die Datenmenge für ganz Deutschland zu groß ist, um sie in einem einzigen Datensatz abzufragen. Daher wurde die Abfrage auf ein Bundesland beschränkt, um die Datenmenge zu reduzieren und die Abfragezeit zu verkürzen. In Listing 3.7 ist die Abfrage für das Bundesland Baden-Württemberg dargestellt. Ebenfalls ist zu sehen, dass nur noch spezifische Wegtypen abgefragt werden, um die Datenmenge weiter zu reduzieren. Hierbei wurden die Wegtypen auf genau typisierte Straßenarten beschränkt, was eine spätere Routenberechnung lediglich für Autos ermöglicht. Dies hat den Hintergrund, dass die Straßenarten für Fußgänger und Fahrradfahrer nicht in den Daten enthalten sind, da diese in OSM anders repräsentiert werden.

Listing 3.7: Abfrage für Bundesländer Daten

```
[out:json];
area["ISO3166-2" ~ "DE-BW"] -> .BW;

// 1. Alle Wege in Baden-Württemberg (White-List)
way(area.BW)[highway=motorway_link] -> .motorway_links;
way(area.BW)[highway=trunk_link] -> .trunk_links;
way(area.BW)[highway=primary_link] -> .primary_links;
way(area.BW)[highway=secondary_link] -> .secondary_links;
way(area.BW)[highway=tertiary_link] -> .tertiary_links;
way(area.BW)[highway=motorway] -> .motorways;
way(area.BW)[highway=trunk] -> .trunk;
way(area.BW)[highway=primary] -> .primary;
way(area.BW)[highway=secondary] -> .secondary;
way(area.BW)[highway=tertiary] -> .tertiary;
way(area.BW)[highway=residential] -> .residential;

// 2. Zusammenfügen der Daten
(.motorways; .trunk; .primary; .secondary; .tertiary; .residential; .motorway_links; .
trunk_links; .primary_links; .secondary_links; .tertiary_links;);

// Ausgabe mit vollständiger Geometrie
(..;>););
out body;
```

3.1.2 Datenverarbeitung

Die Datenverarbeitung erfolgt in zwei Schritten. Zunächst wird überprüft, ob für den gewünschten Datensatz bereits existierende, zwischengespeicherte Daten auf dem lokalen Laufwerk vorhanden sind. Falls dies der Fall ist, werden die bereits berechneten Daten geladen und der rechenintensive Graphenaufbau übersprungen. Sind keine gespeicherten Daten vorhanden, wird der Datensatz eingelesen und der Graphenaufbau durchgeführt. Dieser Schritt spart bei großen Datenmengen erheblich Zeit, da der Graphenaufbau der rechenintensivste Schritt ist.

Es ist wichtig zu beachten, dass die Webanwendung Graphen für ungewohnte Daten lediglich erstellt, jedoch nicht speichert, da es sich um eine clientseitige Anwendung handelt. Das bedeutet, dass die Daten nicht auf dem lokalen Laufwerk gespeichert werden können und somit nur temporär verfügbar sind, das heißt beim Neuladen der Seite gehen diese verloren. Um Graphen zu speichern und dauerhaft zu erstellen, muss das Testscript verwendet werden, das direkt mit Node.js ausgeführt wird und dadurch die Möglichkeit bietet, Daten lokal zu speichern.

Im nachfolgenden Abschnitt ist die Funktionsweise des Graphenaufbaus näher beschrieben.

3.1.2.1 Graphenaufbau

Sollten keine zwischengespeicherten Daten vorliegen, so wird die implementierte Funktion `fromOpenStreetMap` verwendet. Diese erwartet ein JSON Objekt, welches OSM-Daten der folgenden Struktur (siehe Listing 3.8) enthält:

Listing 3.8: Beispielhafte Struktur von OSM-Daten

```
{
  {excluded metaInformation},
  "elements": [
    {
      "type": "node",
      "id": 258355787,
      "lat": 48.8060032,
      "lon": 9.1974121,
      "tags": {
        "access": "customers",
        "entrance": "yes",
        "fee": "yes",
        "name": "Eingang_Rosensteinpark",
        "wheelchair": "yes"
      }
    },
    {
      "type": "way",
      "id": 24019138,
      "nodes": [
        260471546,
        260471755,
        260471558
      ],
      "tags": {
        "highway": "footway",
        "lit": "no",
        "surface": "asphalt"
      }
    },
    ...
  ],
}
```

Wichtig hierbei ist das `elements`-Array, welches die einzelnen Knoten und Wege enthält. Dabei besteht ein Element des Typs `node` mindestens aus den folgenden Attributen:

- **type:** Typ des Elements (`node`)
- **id:** Eindeutige ID des Elements
- **lat:** Breitengrad des Elements
- **lon:** Längengrad des Elements
- **tags:** Objekt mit weiteren Informationen über das Element (z.B. Name, Art des Elements, ...)

Ein Element des Typs `way` besteht aus den folgenden Attributen:

- **type:** Typ des Elements (`way`)
- **id:** Eindeutige ID des Elements
- **nodes:** Array mit den IDs der Knoten, die den Weg bilden
- **tags:** Objekt mit weiteren Informationen über den Weg (z.B. Art des Weges, Oberfläche, ...)

Das übergebene JSON-Objekt der OSM-Daten wird von der `fromOpenStreetMap`-Funktion verarbeitet, um die Knoten und Kanten des Graphen zu erstellen. Dabei werden die Knoten und Kanten jeweils in einem eigenen Objekt gespeichert. Der Aufbau der jeweiligen Objekte wird im nachfolgenden Abschnitt 3.1.3 genauer erläutert.

Das Einlesen der Daten erfolgt dabei in zwei Schritten. Im ersten Schritt werden alle Elemente des Typs `node` durchlaufen und die entsprechende Grund-Knotenstruktur angelegt. Im zweiten Schritt werden die Elemente des Typs `way` durchlaufen. Dabei wird den Knoten, welche in dem Weg enthalten sind die entsprechenden eingehenden und ausgehenden Kanten hinzugefügt. Dabei wird zwischen gerichteten und ungerichteten Kanten unterschieden. Ist eine Kante ungerichtet, so wird sie sowohl als eingehende als auch als ausgehende Kante dem Knoten hinzugefügt. Handelt es sich um eine gerichtet Kante, so wird sie entsprechend lediglich als eingehende oder ausgehende Kante hinzugefügt. Insofern einem Knoten eine neue Kante hinzugefügt wird, wird sein Grad ums eins erhöht. Dies ist für die spätere Graphenkontraktion relevant, da Knoten mit einem Grad von zwei gegebenenfalls entfernt werden können, solange es sich nicht um Verzweigungen handelt.

Nach dem Aufrufen von `fromOpenStreetMap` kann über den Aufruf der implementierten Funktion `to_OL_VectorLayer` aus der erstellten Knotenstruktur die OpenLayers-Features für die Kanten erstellt und zu einem OpenLayers-VectorLayer hinzugefügt werden. Dies ermöglicht im Anschluss die Darstellung des Graphen auf einer Karte. Siehe Abschnitt 3.1.3 für weitere Informationen zur Darstellung des Graphen.

3.1.3 Graphenstruktur

In diesem Abschnitt wird die Struktur des Graphen-Objekts genauer betrachtet. Dabei wird auf den Aufbau der Knoten-Objekte, der Kanten-Objekte und der Hilfsstrukturen eingegangen.

3.1.3.1 Knoten

Die einzelnen Knoten werden im Graphen als Unterobjekte des `nodes`-Objektes gespeichert. Dabei enthält jedes Knoten-Objekt die folgenden Attribute:

- **k:** Die Koordinaten des Knotens (Breitengrad, Längengrad)
- **o:** Die ausgehenden Kanten des Knotens
- **i:** Die eingehenden Kanten des Knotens
- **g:** Grad des Knotens

Durch diesen Aufbau sind alle relevanten Informationen über einen Knoten in einem Objekt gespeichert. Die ID des Knotens ist dabei die ID des Unterobjekts, in dem der Knoten gespeichert ist. Die ausgehenden und eingehenden Kanten sind dabei als Array von Objekten implementiert, wobei der Aufbau wie folgt ist:

- **id:** ID der Kante, zu der der Knoten verbunden ist
- **w:** Gewicht der Kante
- **p:** Entfernte Knoten, die in der Kante im kontrahierten Graphen enthalten sind

Die ID der Kante ist dabei so zusammengesetzt, dass die kleinere ID der Knoten zwischen welchen die Kante verläuft zuerst steht, gefolgt von zwei Bindestrichen und der größeren ID. Dies ermöglicht es, die Kante eindeutig zu identifizieren. Das Gewicht der Kante ist dabei die

Distanz zwischen den beiden Knoten, welche durch die Kante verbunden sind. Die Distanz und damit das Gewicht wird dabei über die Haversine-Formel bestimmt und ist in Metern angeben. Das Attribut path wird benötigt, um die kontrahierten Knoten auf einer Kante im kontrahierten Graphen speichern. Dies ist nötig, um eine Rekonstruktion der Route zu ermöglichen.

Listing 3.9: Exemplarische Struktur des nodes-Objektes

```
{
  "nodes": {
    "258355787": {
      "k": [48.8060032, 9.1974121],
      "o": [
        {
          "id": "258355764--258355787",
          "w": 140,
          "path": [258355766, 258355776, 258355740]
        },
        ...
      ],
      "i": [
        {
          "id": "258355787--258355970",
          "w": 80,
          "path": [258355788, 258355794]
        },
        ...
      ],
      "g": 2
    },
    ...
  }
}
```

Das Beispiel in Listing 3.9 zeigt den Aufbau des nodes-Objektes, wobei der Knoten mit der ID 258355787 exemplarisch dargestellt ist. Dabei sind die ausgehenden Kanten des Knotens in dem Attribut o und die eingehenden Kanten in dem Attribut i gespeichert. Der Grad des Knotens ist dabei 2, da der Knoten zwei nicht identische Kanten hat. Nicht identisch meint in diesem Zusammenhang, dass sich die Kanten_IDs unterscheiden, da zwei unterschiedliche Nachbarknoten verbunden sind. Die Kanten_IDs unterscheiden sich jeweils in einer Zahl, da die ID des eigenen Knotens ebenfalls in der Kanten_ID enthalten sein muss. Die ausgehende Kante mit der ID 258355764-258355787 hat dabei eine Länge, beziehungsweise Gewicht von 140 Metern, wobei drei entfernte Knoten auf dieser Kante liegen. Die eingehende Kante mit der ID 258355787-258355970 hat eine Länge / Gewicht von 80 Metern und beinhaltet ebenfalls zwei entfernte Knoten.

3.1.3.2 Kanten

Die einzelnen Kanten werden im Graphen als Unterobjekte des edges-Objektes gespeichert. Dabei wird lediglich die ID der Kante und eine Referenz auf das OpenLayers-Feature gespeichert, da das edges-Objekt lediglich zur Darstellung des Graphen auf der Karte dient. Ein exemplarischer Aufbau des edges-Objektes ist in Listing 3.10 dargestellt.

Listing 3.10: Exemplarische Struktur des edges-Objektes

```
{
  "edges": {
    "258355787--258355788": OL_Feature,
    ...
  }
}
```

In diesem Beispiel ist die Kante mit der ID 258355787-258355788 und dem dazugehörigen OL-Feature gespeichert. Dies ermöglicht es, zu einem späteren Zeitpunkt das OL-Feature dieser Kante abzurufen, um sie entsprechend auf der Karte einzulegen. Der Aufbau der Kanten_ID ist dabei, wie bereits zuvor beschrieben, so gewählt, dass die kleinere Knoten_ID zuerst steht, gefolgt von zwei Bindestrichen und der größeren Knoten_ID. Dies ermöglicht es, die Kante eindeutig zu identifizieren und beim einzeichnen der Kanten in der Karte keine Kante doppelt einzulegen.

3.1.3.3 Hilfsstrukturen

Zur Bestimmung der Start- und Endknoten im kontrahierten Graphen, werden verschiedene Hilfsstrukturen verwendet. Dadurch wird eine schnellere Routenberechnung ermöglicht. Die dabei verwendeten Hilfsstrukturen sind:

node_next_branching_nodes Diese Struktur speichert zu jeder Knoten_ID die dazugehörigen nicht kontrahierten Nachbarknoten_IDs eingehender (**incoming** *i*) und ausgehender (**outgoing** *o*) Kanten mit den entsprechenden Wegkosten. Zum Referenzieren von Objekten wird die Knoten_ID des jeweilig kontrahierten Knotens als Schlüssel und die ein- und ausgehenden Knoten als Array von Objekten gespeichert. Dabei stellen Knoten_IDs in der **incoming**-Liste die nächsten Endknoten für den späteren Routingalgorithmus dar. Knoten_IDs in der **outgoing**-Liste stellen die nächsten Startknoten dar. Diese Struktur ermöglicht es, die nächsten Start- und Endknoten im kontrahierten Graphen eines jeden Knotens effizient durch Zugriff auf das entsprechende Objekt zu bestimmen. Ein exemplarischer Aufbau der Struktur ist in Listing 3.11 dargestellt:

Listing 3.11: Beispielhafte Struktur von node_next_branching_nodes

```
{
  "contracted_nodeID": {
    i: [
      {
        id: nodeID_uncontracted,
        w: weight
      },...
    ]
    o: [
      {
        id: nodeID_uncontracted,
        w: weight
      },...
    ]
  },...
}
```

nodes_uncontracted_to_contracted Da die nächsten Start- und Endknoten eines entfernten Knotens nicht direkt bekannt sind, sondern während des Kontrahierens des Graphen dynamisch bestimmt werden, wird die Hilfsstruktur `nodes_uncontracted_to_contracted` verwendet.

Diese Datenstruktur dient dabei zur Laufzeit effizienten Änderung der Knotenreferenzen auf nicht kontrahierte Start- und Endknoten eines bereits entfernten Knotens während des Kontrahierens. Dabei wird für jeden nicht kontrahierten Knoten die Referenz auf den nächsten bereits entfernten Knoten gespeichert, welcher diesen nicht kontrahierten Knoten als seinen Startbeziehungsweise Endknoten referenziert. Diese Struktur hilft dabei zur Bestimmung derjenigen Knoten, welche geändert werden müssen, sollte ein noch nicht kontrahierter Knoten ebenfalls entfernt werden.

Ohne die Verwendung dieser Struktur müsste durch alle bereits entfernten Knoten in der Hilfsstruktur `node_next_branching_nodes` iteriert werden, um diejenigen Knoten zu bestimmen, welche auf einen nun ebenfalls kontrahierten Knoten verweisen. Dabei steigt die Laufzeit mit der Anzahl der zu kontrahierenden Knoten erheblich, da bei jedem Entfernen eines Knotens durch alle bereits entfernten Knoten iteriert werden muss. Dieses Vorgehen führt zu einer exponentiell wachsenden Laufzeit, was den Ansatz für große Datenmengen unbrauchbar macht. Durch die Verwendung der Zwischenspeicher-Struktur `nodes_uncontracted_to_contracted` wird die Laufzeit konstant gehalten, da lediglich auf das entsprechende Objekt eines zu entfernenden Knotens zugegriffen werden muss. Die Struktur ist dabei wie folgt aufgebaut:

Listing 3.12: Beispielhafte Struktur von `nodes_uncontracted_to_contracted`

```
{
  "nodeID_uncontracted": {
    i: [nodeID_contracted, ...],
    o: [nodeID_contracted, ...],
  }
  ...
}
```

Wie in Listing 3.12 zu sehen, wird zu jedem nicht kontrahierten Knoten die Referenzen auf bereits entfernte Knoten gespeichert, welche den nicht kontrahierten Knoten als ein- oder ausgehenden Knoten referenzieren. Dabei wird die Knoten_ID des nicht kontrahierten Knotens als Schlüssel verwendet und die ein- und ausgehenden Knoten als Array von Knoten_IDs gespeichert. Diese Struktur ermöglicht es, diejenigen bereits entfernten Knoten, welche auf einen nicht kontrahierten Knoten verweisen, effizient zu bestimmen, um die Referenzen gegebenenfalls zu aktualisieren.

3.2 Graphenkontraktion

In diesem Abschnitt wird der implementierte Graphenkontraktionsalgorithmus genauer betrachtet. Dabei wird auf die Idee hinter dem Algorithmus, sowie eine detaillierte Beschreibung des Algorithmus eingegangen. Ebenfalls wird ein Beispiel zur Verdeutlichung der Graphenkontraktion durchgeführt.

Der implementierte Graphenkontraktionsalgorithmus basiert auf der Idee den Graphen soweit wie möglich verlustfrei zu reduzieren, sodass die Anzahl der enthaltenen Knoten und Kanten möglichst gering ist und nur noch relevanten Verzweigungen enthalten sind. Dies hat den Hintergrund, dass die Anzahl der Knoten und Kanten die Laufzeit der Routenberechnung direkt

beeinflusst. Je mehr Knoten und Kanten vorhanden sind und sich zwischen dem Start- und Zielknoten befinden, desto länger dauert die Berechnung der Route. Daher ist es sinnvoll, den Graphen und damit die Anzahl an Knoten und Kanten grundlegend soweit wie möglich zu reduzieren.

Dies erfordert im ersten Schritt die Identifikation von Knoten, die entfernt werden können, ohne Informationen über den Graphen zu verlieren. Das entspricht allen Knoten mit Grad zwei, welche keine Verzweigungen darstellen. Diese Knoten können entfernt werden, da sie keine relevanten Informationen über die Verzweigungen im Graphen liefern und bei der Routenberechnung daher nicht berücksichtigt werden müssen. Listing 3.13 zeigt die Identifizierung der Knoten, welche entfernt werden können. Dabei werden alle Knoten durchlaufen und die Knoten mit einem Grad von zwei identifiziert. Diese Knoten werden anschließend in einem Array gespeichert, um sie später entfernen zu können.

Listing 3.13: Identifizierung zu entfernender Knoten

```
const nodeIdsToRemove = Object.entries(this.nodes)
  .filter(([_, nodeData]) => nodeData.g === 2)
  .map(([nodeId, _]) => nodeId);
```

In nächsten Schritt ist allerdings zu beachten, dass nicht alle Knoten mit Grad zwei entfernt werden dürfen. So können gerade im Randbereich eines Datensatzes Knoten existieren, welche zwar den Grad zwei besitzen, jedoch in einem größeren Datensatz eine Verzweigung darstellen. Sind diese Verzweigungen nicht vollständig im Datensatz enthalten, kann der Knoten den Grad 2 erhalten. Solche Knoten können nicht entfernt werden, da sie beispielsweise nur zwei verschiedene ausgehende Kanten besitzen und keine eingehende Kante, wodurch der Grad zwei zustande kommt. Daher ist es zusätzlich notwendig neben dem Grad des Knotens auch zu überprüfen, ob die Anzahl ein- und ausgehender Kanten übereinstimmt. Nur wenn dies der Fall ist, kann der Knoten entfernt werden.

Ist ein solcher Knoten gefunden, so muss zunächst überprüft werden, ob die neue Kante gerichtet oder ungerichtet ist. Dies geschieht über die Anzahl an ein- und ausgehenden Kanten des zu entfernenden Knotens. Ist die Anzahl eins (1), so handelt es sich um eine gerichtete Kante, da der Knoten lediglich in eine Richtung durchwandert werden kann. Ist die Anzahl jedoch zwei (2), so handelt es sich um eine ungerichtete Kante, da der Knoten in beide Richtungen durchwandert werden kann. Basierend auf diesem Wissen, kann die neue Kante erstellt werden und der Graph entsprechend angepasst werden.

Bevor der Algorithmus weiter beschrieben wird, muss klar sein, dass beim Routing auf dem kontrahierten Graphen die nächsten nicht kontrahierten Knoten bekannt sein müssen. Diese dienen als Start- und Endpunkte, da der kontrahierte Graph nicht mehr alle ursprünglichen Knoten enthält.

Daher wird zu jedem kontrahierten Knoten eine Referenz auf die verbleibenden Nachbarknoten gespeichert – inklusive deren IDs und der Verbindungskosten. Diese Kosten ergeben sich aus der Summe der Kanten entlang des Pfads zum nächsten nicht kontrahierten Knoten.

Um die Bestimmung dieser Knoten effizient zu gestalten, werden die Werte in der Hilfsstruktur `node_next_branching_nodes` gespeichert.

3.2.1 Pseudocode

Dieser Abschnitt zeigt die Funktionen zur Graphenkontraktion in Pseudocode mit kurzer Beschreibung der jeweiligen Funktionsweise.

Die Funktion `ContractGraph` (siehe Algorithmus 1) führt die Graphenkontraktion durch. Sie bestimmt die zu entfernenden Knoten und entscheidet darüber, welche Funktion zur Zusammenführung der Kanten aufgerufen wird. Ebenfalls fügt sie zum Schluss alle nicht entfernte Knoten der Hilfsstruktur `node_next_branching_nodes` hinzu, wobei die Knoten auf sich selbst verweisen. Dies ermöglicht die spätere Verwendung einer simplen Abfrage um die möglichen Start- und Endknoten zu einer gegebenen Knoten_ID zu bestimmen.

Algorithm 1 ContractGraph

```

1: function CONTRACTGRAPH
2:   nodeIdsToRemove  $\leftarrow$  Object.entries(nodes)
   .FILTER(([_, nodeData])  $\mapsto$  nodeData.g = 2)
   .MAP(([nodeId, _])  $\mapsto$  nodeId)
3:   node_next_branching_nodes  $\leftarrow$  {}
4:   nodes_uncontracted_to_contracted  $\leftarrow$  {}
5:   for all cId  $\in$  nodeIdsToRemove do
6:     cData  $\leftarrow$  nodes[cId]
7:     inEdgeCount  $\leftarrow$  cData.i.length
8:     outEdgeCount  $\leftarrow$  cData.o.length
9:     if inEdgeCount  $\neq$  outEdgeCount then
10:      continue
11:    end if
12:    if cData.i.EVERY(inE  $\mapsto$  cData.o.EVERY(outE  $\mapsto$  inE.id = outE.id)) then
13:      continue ▷ Circle detected
14:    end if
15:    if inEdgeCount = 1 then
16:      MERGEDIRECTEDEDGES(cId, cData) ▷ Node on a one-way street
17:    else if inEdgeCount = 2 then
18:      MERGEUNDIRECTEDEDGES(cId, cData) ▷ Node on a two-way street
19:    else
20:      throw error "Unexpected edge count"
21:    end if
22:  end for
23:  nodeIDsInNodesObj  $\leftarrow$  Object.keys(nodes)
24:  for all nId  $\in$  nodeIDsInNodesObj do
25:    node_next_branching_nodes[nId]  $\leftarrow$  {
26:      i : [{id : nId, w : 0}],
27:      o : [{id : nId, w : 0}]
28:    }
29:  end for
30: end function
```

Je nach Anzahl der ein- und ausgehenden Kanten des Knotens wird eine der beiden Funktionen `MergeDirectedEdges` oder `MergeUndirectedEdges` aufgerufen. Diese Funktionen sind in

den Algorithmen 2 und 3 dargestellt. Beide Funktionen rufen die Funktion `MergeEdge` auf, die in Algorithmus 4 dargestellt ist. Diese Funktion führt die eigentliche Zusammenführung der Kanten durch. Nach dem Zusammenführen der Kanten wird der zu entfernende Knoten sowohl aus dem Graphen als auch aus der Hilfsstruktur `nodes_uncontracted_to_contracted` entfernt. Das Zusammenführen einer ungerichteten Kante unterscheidet sich dabei lediglich dadurch, dass die `MergeEdge` Funktion zweimal aufgerufen wird, um beide Kanten zu verbinden. Dabei wird im ersten Durchlauf die eine Richtung der Kante und im zweiten Durchlauf die andere Richtung der Kante zusammengeführt.

Algorithm 2 MergeDirectedEdges

```

1: function MERGEDIRECTEDEDGES(cId, cData)
2:   inEdge  $\leftarrow$  cData.i[0]
3:   outEdge  $\leftarrow$  cData.o[0]
4:   MERGEEDGE(cId, cData, inEdge, outEdge)
5:   delete nodes_uncontracted_to_contracted[cId]
6:   delete nodes[cId]
7: end function
```

Algorithm 3 MergeUndirectedEdges

```

1: function MERGEUNDIRECTEDEDGES(cId, cData)
2:   [inEdge1, inEdge2]  $\leftarrow$  cData.i
3:   [outEdge1, outEdge2]  $\leftarrow$  cData.o
4:   MERGEEDGE(cId, cData, inEdge1, outEdge2)
5:   MERGEEDGE(cId, cData, inEdge2, outEdge1)
6:   delete nodes_uncontracted_to_contracted[cId]
7:   delete nodes[cId]
8: end function
```

Algorithm 4 MergeEdge

```

1: function MERGEEDGE(cId, cData, inEdge, outEdge)
2:   inEId  $\leftarrow$  inEdge.id; outEId  $\leftarrow$  outEdge.id
3:   nInId  $\leftarrow$  GETNEIGHBORID(inEId, cId); nOutId  $\leftarrow$  GETNEIGHBORID(outEId, cId)
4:   cWeight  $\leftarrow$  inEdge.w + outEdge.w
5:   newEId  $\leftarrow$  CREATENEWEDGEID(nInId, nOutId)
6:   inPath  $\leftarrow$  inEdge.p; outPath  $\leftarrow$  outEdge.p
7:   newPath  $\leftarrow$  CONSTRUCTNEWPATH(inPath, cId, outPath)
8:
9:   STOREMAPPINGFORCROSSING(cId, nInId, nOutId, inEdge, outEdge)
10:  STOREINHELPERS(cId, nInId, nOutId)
11:  UPDATEHELPERS(cId, nInId, inEdge, nOutId, outEdge)
12:  REPLACEOUTEDGE(nInId, inEId, newEId, cWeight, newPath)
13:  REPLACEINEDGE(nOutId, outEId, newEId, cWeight, newPath)
14: end function
```

Die Funktion `MergeEdge` (siehe Algorithmus 4) wird verwendet, um die Kanten zu einem gegebenen Knoten zusammenzuführen. Dabei werden jeweils die Kanten der Nachbarknoten

entsprechend mit neuen IDs, Knoten Referenzen, Gewichten und Pfaden aktualisiert. Sodass die entsprechenden Nachbarknoten des entfernten Knotens miteinander verbunden sind.

Algorithm 5 StoreMappingForCrossing

```

1: function STOREMAPPINGFORCROSSING( $cId, nInId, nOutId, inEdge, outEdge$ )
2:   if  $node\_next\_branching\_nodes[cId] = \text{null}$  then
3:      $node\_next\_branching\_nodes[cId] \leftarrow \{$ 
4:        $i : [\{id : nInId, w : inEdge.w\}],$ 
5:        $o : [\{id : nOutId, w : outEdge.w\}]$ 
6:      $\}$ 
7:   else
8:      $node\_next\_branching\_nodes[cId].i.PUSH(\{id : nInId, w : inEdge.w\})$ 
9:      $node\_next\_branching\_nodes[cId].o.PUSH(\{id : nOutId, w : outEdge.w\})$ 
10:   end if
11: end function
```

Die Funktion `StoreMappingForCrossing` (siehe Algorithmus 5) wird verwendet, um die Hilfsstruktur `node_next_branching_nodes` mit den neuen Knoteninformationen zu aktualisieren.

Algorithm 6 StoreInHelpers

```

1: function STOREINHELPERS( $cId, nInId, nOutId$ )
2:   if  $nodes\_uncontracted\_to\_contracted[nInId] = \text{null}$  then
3:      $nodes\_uncontracted\_to\_contracted[nInId] \leftarrow \{i : [], o : []\}$ 
4:   end if
5:    $nodes\_uncontracted\_to\_contracted[nInId].i.PUSH(cId)$ 
6:   if  $nodes\_uncontracted\_to\_contracted[nOutId] = \text{null}$  then
7:      $nodes\_uncontracted\_to\_contracted[nOutId] \leftarrow \{i : [], o : []\}$ 
8:   end if
9:    $nodes\_uncontracted\_to\_contracted[nOutId].o.PUSH(cId)$ 
10: end function
```

Die Funktion `StoreInHelpers` (siehe Algorithmus 6) wird verwendet, um die Hilfsstruktur `nodes_uncontracted_to_contracted` mit den neuen Knoteninformationen zu aktualisieren.

Algorithm 7 UpdateHelpers

```

1: function UPDATEHELPERS( $cId, nInId, inEdge, nOutId, outEdge$ )
2:    $nodesToUpdate \leftarrow nodes\_uncontracted\_to\_contracted[cId]$ 
3:   if  $nodesToUpdate = \text{null}$  then return
4:   end if
5:   for all  $nId \in nodesToUpdate.i$  do
6:     if  $\neg nodes\_uncontracted\_to\_contracted[nInId].i.\text{INCLUDES}(nId)$  then
7:        $nodes\_uncontracted\_to\_contracted[nInId].i.\text{PUSH}(nId)$ 
8:     end if
9:     if  $node\_next\_branching\_nodes[nId].i.\text{FIND}(e \mapsto e.id = nInId) \neq \text{null}$  then
10:      continue
11:    end if
12:     $inEToUpdate \leftarrow node\_next\_branching\_nodes[nId].i.\text{FIND}(e \mapsto e.id = cId)$ 
13:    if  $inEToUpdate \neq \text{null}$  then
14:       $inEToUpdate.id \leftarrow nInId$ 
15:       $inEToUpdate.w \leftarrow inEToUpdate.w + inEdge.w$ 
16:    else
17:      throw error "Incoming edge not found"
18:    end if
19:   end for
20:   for all  $nId \in nodesToUpdate.o$  do
21:     if  $\neg nodes\_uncontracted\_to\_contracted[nOutId].o.\text{INCLUDES}(nId)$  then
22:        $nodes\_uncontracted\_to\_contracted[nOutId].o.\text{PUSH}(nId)$ 
23:     end if
24:     if  $node\_next\_branching\_nodes[nId].o.\text{FIND}(e \mapsto e.id = nOutId) \neq \text{null}$  then
25:       continue
26:     end if
27:      $outEToUpdate \leftarrow node\_next\_branching\_nodes[nId].o.\text{FIND}(e \mapsto e.id = cId)$ 
28:     if  $outEToUpdate \neq \text{null}$  then
29:        $outEToUpdate.id \leftarrow nOutId$ 
30:        $outEToUpdate.w \leftarrow outEToUpdate.w + outEdge.w$ 
31:     else
32:       throw error "Outgoing edge not found"
33:     end if
34:   end for
35: end function

```

Die Funktion `UpdateHelpers` (siehe Algorithmus 7) wird verwendet, um die Hilfsstrukturen `node_next_branching_nodes` und `nodes_uncontracted_to_contracted` entsprechend zu aktualisieren. Dabei werden anhand der Hilfsstruktur `nodes_uncontracted_to_contracted` diejenigen Knoten bestimmt, die bislang auf den jetzt entfernten Knoten verwiesen haben. Diese Knoten werden entsprechend aktualisiert, sodass sie auf die neuen, korrekten nicht entfernten Knoten verweisen. Ebenfalls wird die Hilfsstruktur `node_next_branching_nodes` aktualisiert, um weiterhin die richtige Verbindung der Knoten zu speichern.

Algorithm 8 ReplaceOutEdge

```

1: function REPLACEOUTEDGE(nId, cEId, newEId, newW, newP)
2:   nodes[nId].o  $\leftarrow$  nodes[nId].o.FILTER(e  $\mapsto$  e.id  $\neq$  cEId)
3:   nodes[nId].o.PUSH({id : newEId, w : newW, p : newP})
4: end function
```

Die Funktion ReplaceOutEdge (siehe Algorithmus 8) wird verwendet, um die ausgehenden Kanten eines Knotens zu aktualisieren. In diesem Fall handelt es sich um die ausgehende Kante des Nachbarknotens, welcher über eine eingehende Kante mit dem aktuellen Knoten verbunden ist. Dieselbe Kante existiert bei diesem eingehenden Nachbarknoten in den ausgehenden Kanten, welche dann entsprechend aktualisiert wird.

Algorithm 9 ReplaceInEdge

```

1: function REPLACEINEDGE(nId, cEId, newEId, newW, newP)
2:   nodes[nId].i  $\leftarrow$  nodes[nId].i.FILTER(e  $\mapsto$  e.id  $\neq$  cEId)
3:   nodes[nId].i.PUSH({id : newEId, w : newW, p : newP})
4: end function
```

Die Funktion ReplaceInEdge (siehe Algorithmus 9) wird verwendet, um entsprechend die eingehenden Kanten eines Knotens zu aktualisieren. In diesem Fall handelt es sich um die eingehende Kante des Nachbarknotens, welcher über eine ausgehende Kante mit dem aktuellen Knoten verbunden ist. Dieselbe Kante existiert bei diesem ausgehenden Nachbarknoten in den eingehenden Kanten, welche dann entsprechend aktualisiert wird.

Algorithm 10 GetNeighborId

```

1: function GETNEIGHBORID(eId, cId)
2:   [n1, n2]  $\leftarrow$  eId.SPLIT("--")
3:   return n1 = cId?n2 : n1
4: end function
```

Die Funktion GetNeighborId (siehe Algorithmus 10) wird verwendet, um die ID des Nachbarknoten zu einem gegebenen Knoten und Kante zu erhalten.

Algorithm 11 CreateNewEdgeId

```

1: function CREATENEWEDGEID(n1, n2)
2:   return min(parseInt(n1), parseInt(n2)) + " --" + max(parseInt(n1), parseInt(n2))
3: end function
```

Die Funktion CreateNewEdgeId (siehe Algorithmus 11) wird verwendet, um eine neue Kanten-ID zu erstellen.

Algorithm 12 ConstructNewPath

```

1: function CONSTRUCTNEWPATH(prevIds, cId, nextIds)
2:   prevIds.PUSH(cId)
3:   return prevIds.CONCAT(nextIds)
4: end function

```

Die Funktion `ConstructNewPath` (siehe Algorithmus 12) wird verwendet, um Pfade der Kanten zu verbinden. Hierbei wird der Pfad der eingehenden Kante um die Knoten ID des aktuellen Knotens erweitert. Danach wird der weitere Pfad der ausgehenden Kante an diesen erweiterten Pfad ebenfalls angehängt. Dadurch wird die Reihenfolge der Knoten in Richtung des ausgehenden Knotens beibehalten, was eine spätere Rekonstruktion des Gesamtpfades ermöglicht. Ein Beispiel für die Zusammenführung von Kanten ist in Abbildung 3.1 dargestellt.

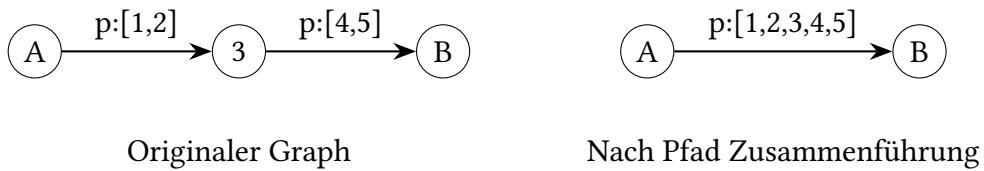


Abbildung 3.1: Beispiel für die Funktionsweise der Funktion `ConstructNewPath` [Eigene Darstellung]

3.2.2 Beispiel Kontraktion

In diesem Abschnitt wird exemplarisch die Kontraktion eines Graphen durchgeführt, wie sie die Funktion `ContractGraph` (siehe Algorithmus 1) durchführt. Dabei wird ein Graph mit verschiedenen Knoten und Kanten betrachtet und die Kontraktion durchgeführt. Es wird auf die einzelnen Schritte der Kontraktion eingegangen und die Auswirkungen auf den Graphen gezeigt. Ebenfalls werden die Hilfsstrukturen `node_next_branching_nodes` und `nodes_uncontracted_to_contracted` betrachtet und deren durch die Kontraktion entstehenden Änderungen gezeigt.

3.2.2.1 Schritt 0: Graph vor der Kontraktion

In Abbildung 3.2 ist der Graph vor der Kontraktion dargestellt. In Tabelle 3.1 ist die Knotenstruktur des Graphen vor der Kontraktion dargestellt. Knoten mit Buchstaben symbolisieren Verzweigungen, welche nicht kontrahierbar sind. Knoten mit Zahlen sind hingegen als kontrahierbar einzustufen, da ihre Positionierung auf einer Kette lediglich die Verbindung zu zwei weiteren Knoten zulässt und eine Verzweigung an diesen Punkten nicht möglich ist. Knoten mit einer gerichteten Verbindung sind dabei durch eine Pfeilspitze dargestellt, während Knoten mit einer ungerichteten Verbindung durch eine durchgezogene Linie dargestellt sind. Die Gewichte der Kanten sind dabei in den Kanten dargestellt.

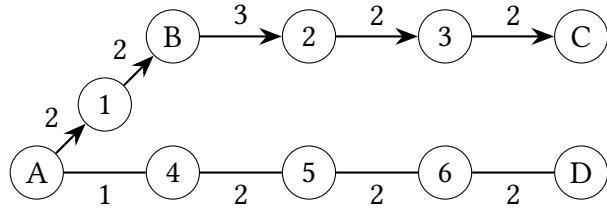


Abbildung 3.2: Beispiel Graph vor der Kontraktion [Eigene Darstellung]

Tabelle 3.1: Knotenstruktur des Beispielgraphen

ID	k	o	i	g
A	[0,0]	{id: "A-1", w: 2}, {id: "A-4", w: 1}	{id: "A-4", w: 1}	>2
B	[2,2]	{id: "B-2", w: 3}	{id: "1-B", w: 2}	>2
C	[8,2]	-	{id: "3-C", w: 2}	>2
D	[8,0]	{id: "6-D", w: 2}	{id: "6-D", w: 2}	>2
1	[1,1]	{id: "1-B", w: 2}	{id: "A-1", w: 2}	2
2	[4,2]	{id: "2-3", w: 2}	{id: "B-2", w: 3}	2
3	[6,2]	{id: "3-C", w: 2}	{id: "2-3", w: 2}	2
4	[2,0]	{id: "4-5", w: 2}, {id: "A-4", w: 1}	{id: "4-5", w: 2}, {id: "A-4", w: 1}	2
5	[4,0]	{id: "5-6", w: 2}, {id: "4-5", w: 2}	{id: "5-6", w: 2}, {id: "4-5", w: 2}	2
6	[6,0]	{id: "D-6", w: 2}, {id: "5-6", w: 2}	{id: "D-6", w: 2}, {id: "5-6", w: 2}	2

3.2.2.2 Schritt 1: Identifikation der zu entfernenden Knoten

Im ersten Schritt wird die Identifikation der zu entfernenden Knoten durchgeführt. Diese Knoten werden anschließend in einem Array gespeichert, um sie nacheinander zu entfernen. In diesem Beispiel sind alle Knoten mit Zahlen kontrahierbar. Sei nachfolgend die Identifikation der zu entfernenden Knoten in Tabelle 3.2 dargestellt:

Tabelle 3.2: Identifizierte kontrahierbare Knoten

ID
1
2
3
4
6
5

Die Reihenfolge der zu kontrahierenden Knoten in Tabelle 3.2 ist dabei willkürlich gewählt und für die Kontraktion nicht relevant, da die Kantenkontraktion unabhängig von der Reihenfolge der Knoten durchgeführt werden kann und das gleiche Ergebnis erzielt wird.

3.2.2.3 Schritt 2: Kontraktion des Knotens 1

Im nächsten Schritt wird der erste Knoten aus dem Array der zu kontrahierenden Knoten entfernt. In diesem Beispiel wird der Knoten mit der ID 1 entfernt. Bei diesem Knoten handelt es sich um einen Knoten auf einer gerichteten Verbindung. Die Auswirkungen auf den Graphen sind in Abbildung 3.3 dargestellt. Die Knotenstruktur des Graphen nach der Kontraktion des Knotens 1 ist in Tabelle 3.3 dargestellt.

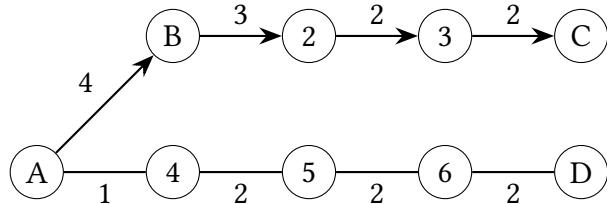


Abbildung 3.3: Beispiel Graph nach der Kontraktion des Knotens 1 [Eigene Darstellung]

Tabelle 3.3: Knotenstruktur des Beispielgraphen nach der Kontraktion des Knotens 1

ID	k	o	i	g
A	[0,0]	{id: "A-B", w: 4, p:[1]}, {id: "A-4", w: 1}	{id: "A-4", w: 1}	>2
B	[2,2]	{id: "B-2", w: 3}	{id: "A-B", w: 4, p: [1]}	>2
C	[8,2]	-	{id: "C-3", w: 2}	>2
D	[8,0]	{id: "D-6", w: 2}	{id: "D-6", w: 2}	>2
2	[4,2]	{id: "2-3", w: 2}	{id: "B-2", w: 3}	2
3	[6,2]	{id: "C-3", w: 2}	{id: "2-3", w: 2}	2
4	[2,0]	{id: "4-5", w: 2}, {id: "A-4", w: 1}	{id: "4-5", w: 2}, {id: "A-4", w: 1}	2
5	[4,0]	{id: "5-6", w: 2}, {id: "4-5", w: 2}	{id: "5-6", w: 2}, {id: "4-5", w: 2}	2
6	[6,0]	{id: "D-6", w: 2}, {id: "5-6", w: 2}	{id: "D-6", w: 2}, {id: "5-6", w: 2}	2

Durch die Kontraktion des Knotens 1 wurden die betroffene Kante des Knotens A und B aktualisiert. Durch diese Aktualisierung erhält die neu verbundene Kante zwischen A und B die Summe der Gewichte der vorherigen Kanten als Gewicht. Ebenfalls erhält diese neue Kante die kontrahierte Knoten_ID als Pfad-Element, des Pfades (p).

Neben der Aktualisierungen der Kanten wurden auch die beiden neuen Hilfsstrukturen `node_next_branching_nodes` und `nodes_uncontracted_to_contracted` angepasst.

Tabelle 3.4: `node_next_branching_nodes` nach der Kontraktion des Knotens 1

ID	o	i
1	{id: "B", w: 2}	{id: "A", w: 2}

In Tabelle 3.4 ist die Hilfsstruktur `node_next_branching_nodes` nach der Kontraktion des Knotens 1 dargestellt. Dabei ist zu sehen, dass der Knoten 1 als kontrahierter Knoten mit dem nächsten Startknoten B und Endknoten A und den entsprechenden Kosten eingetragen ist.

Tabelle 3.5: nodes_uncontracted_to_contracted nach der Kontraktion des Knotens 1

ID	o	i
A	[]	[1]
B	[1]	[]

In Tabelle 3.5 ist die Hilfsstruktur nodes_uncontracted_to_contracted nach der Kontraktion des Knotens 1 dargestellt. Dabei ist zu sehen, dass die Knoten A und B als nicht kontrahierte Knoten mit dem kontrahierten Knoten 1 eingetragen sind. Dies bedeutet, dass bei einer möglichen Kontraktion des Knotens A oder B entsprechend an der incoming oder outgoing Liste des Knotens 1 Änderungen vorgenommen werden müssen. Dabei ist zu sehen, dass die 1 als eingehender Knoten zu A eingetragen ist, da der Knoten A in den eingehenden Knoten von dem Knoten 1 in der node_next_branching_nodes Struktur eingetragen ist. Für den Knoten B ist der Knoten 1 als ausgehender Knoten eingetragen.

3.2.2.4 Schritt 3: Kontraktion des Knotens 2

Nun wird der nächste Knoten aus der Liste der zu kontrahierenden Knoten betrachtet und kontrahiert. In diesem Beispiel ist das der Knoten mit der ID 2. Die Auswirkungen sind dabei in Abbildung 3.4 und den Tabellen 3.6, 3.7 und 3.8 dargestellt.

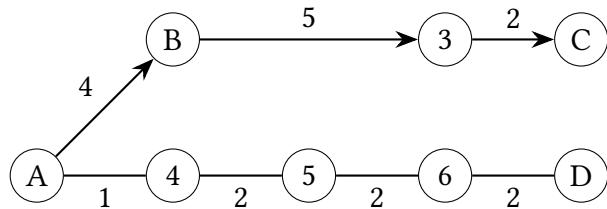


Abbildung 3.4: Beispiel Graph nach der Kontraktion des Knotens 2 [Eigene Darstellung]

Tabelle 3.6: Knotenstruktur des Beispielgraphen nach der Kontraktion des Knotens 2

ID	k	o	i	g
A	[0,0]	{id: "A-B", w: 4, p:[1]}, {id: "A-4", w: 1}	{id: "A-4", w: 1}	>2
B	[2,2]	{id: "B-3", w: 5, p:[2]}	{id: "A-B", w: 4, p: [1]}	>2
C	[8,2]	{id: "D-6", w: 2}	{id: "C-3", w: 2}	>2
D	[8,0]	{id: "D-6", w: 2}	{id: "D-6", w: 2}	>2
3	[6,2]	{id: "C-3", w: 2}	{id: "B-3", w: 5, p:[2]}	2
4	[2,0]	{id: "4-5", w: 2}, {id: "A-4", w: 1}	{id: "4-5", w: 2}, {id: "A-4", w: 1}	2
5	[4,0]	{id: "5-6", w: 2}, {id: "4-5", w: 2}	{id: "5-6", w: 2}, {id: "4-5", w: 2}	2
6	[6,0]	{id: "D-6", w: 2}, {id: "5-6", w: 2}	{id: "D-6", w: 2}, {id: "5-6", w: 2}	2

Durch die Kontraktion des Knotens 2 wurden die betroffene Kante des Knotens B und 3 aktualisiert. Durch diese Aktualisierung erhält die neu verbundene Kante zwischen B und 3

die Summe der Gewichte vorherigen Kanten als Gewicht. Ebenfalls erhält diese neue Kante die kontrahierte Knoten_ID als Pfad-Element, des Pfades (p).

Tabelle 3.7: node_next_branching_nodes nach der Kontraktion des Knotens 2

ID	o	i
1	{id: "B", w: 2}	{id: "A", w: 2}
2	{id: "3", w: 2}	{id: "B", w: 3}

In Tabelle 3.7 ist die Hilfsstruktur node_next_branching_nodes nach der Kontraktion des Knotens 2 dargestellt. Dabei ist zu sehen, dass der Knoten 2 als kontrahierter Knoten mit dem nächsten Startknoten 3 und Endknoten B und den entsprechenden Kosten eingetragen ist.

Tabelle 3.8: nodes_uncontracted_to_contracted nach der Kontraktion des Knotens 2

ID	o	i
A	[]	[1]
B	[1]	[2]
3	[2]	[]

3.2.2.5 Schritt 4: Kontraktion des Knotens 3

In diesem Schritt wird der Knoten 3 kontrahiert. Die Auswirkungen sind dabei in Abbildung 3.5 und den Tabellen 3.9, 3.10 und 3.11 dargestellt.

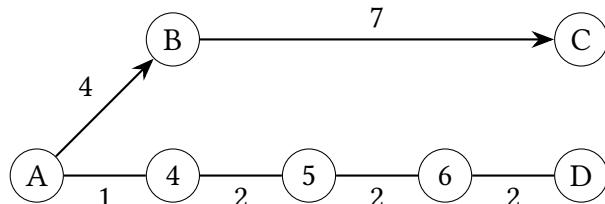


Abbildung 3.5: Beispiel Graph nach der Kontraktion des Knotens 3 [Eigene Darstellung]

Tabelle 3.9: Knotenstruktur des Beispielgraphen nach der Kontraktion des Knotens 3

ID	k	o	i	g
A	[0,0]	{id: "A-B", w: 4, p:[1]}, {id: "A-4", w: 1}	{id: "A-4", w: 1}	>2
B	[2,2]	{id: "B-C", w: 7, p:[2, 3]}	{id: "A-B", w: 4, p: [1]}	>2
C	[8,2]	-	{id: "B-C", w: 7, p:[2, 3]}	>2
D	[8,0]	{id: "D-6", w: 2}	{id: "D-6", w: 2}	>2
4	[2,0]	{id: "4-5", w: 2}, {id: "A-4", w: 1}	{id: "4-5", w: 2}, {id: "A-4", w: 1}	2
5	[4,0]	{id: "5-6", w: 2}, {id: "A-5", w: 3, p: [4]}	{id: "5-6", w: 2}, {id: "A-5", w: 3, p: [4]}	2
6	[6,0]	{id: "D-6", w: 2}, {id: "5-6", w: 2}	{id: "D-6", w: 2}, {id: "5-6", w: 2}	2

Durch die Kontraktion des Knotens 3 wurden die betroffene Kante des Knotens B und c aktualisiert. Ebenfalls erhält diese neue Kante die kontrahierte Knoten_ID als Pfad-Element, des Pfades (p). Hierbei existiert bereits ein vorhandener Pfad, welcher durch die Kontraktion des Knotens 2 entstanden ist. Dieser Pfad wird um den kontrahierten Knoten 3 erweitert.

Tabelle 3.10: node_next_branching_nodes nach der Kontraktion des Knotens 3

ID	o	i
1	{id: "B", w: 2}	{id: "A", w: 2}
2	{id: "C", w: 4}	{id: "B", w: 3}
3	{id: "C", w: 2}	{id: "B", w: 5}

Wie in Tabelle 3.10 dargestellt ist, ist der Knoten 3 als kontrahierter Knoten mit seinem entsprechenden Startknoten B und Endknoten C und den entsprechenden Gewichten in dem node_next_branching_nodes-Objekt eingetragen. Ebenfalls ist zu beachten, dass die Knoten-Referenz des bereits kontrahierten Knotens 2 aktualisiert wurde. Die Aktualisierung der Stellen, an welchen der Knoten 3 bislang als Start- oder Endknoten referenziert wird, wurde dabei über die Funktion UpdateHelpers (siehe Algorithmus 7) durchgeführt. Dadurch wurden auch die entsprechenden Referenzen in den Hilfsstrukturen aktualisiert. In diesem Schritt wird ebenfalls der Knoten 3 aus der Struktur nodes_uncontracted_to_contracted entfernt, da er nun kontrahiert ist. Weiterhin wurde der nicht kontrahierte Knoten C in die Struktur aufgenommen und mit den entsprechenden Referenzen eingetragen. Die Struktur nodes_uncontracted_to_contracted ist in Tabelle 3.11 dargestellt.

Tabelle 3.11: nodes_uncontracted_to_contracted nach der Kontraktion des Knotens 3

ID	o	i
A	[]	[1]
B	[1]	[2,3]
C	[2,3]	[]

Nach diesem Schritt sind bereits alle gerichteten Kanten mit ihren Knoten vollständig kontrahiert. Anhand der node_next_branching_nodes Struktur ist nun leicht zu erkennen, welche Möglichkeiten bestehen, um zu einem gegebenen kontrahierten Knoten zu gelangen, siehe hierzu die eingehenden Knoten_IDs und welche Möglichkeiten bestehen von diesem Knoten aus weiter zu gehen, siehe hierfür die ausgehenden Knoten_IDs.

3.2.2.6 Schritt 5: Kontraktion des Knotens 4

In Schritt 5 wird der Knoten 4 kontrahiert. Die Auswirkungen sind dabei in Abbildung 3.6 und den Tabellen 3.12, 3.13 und 3.14 dargestellt.

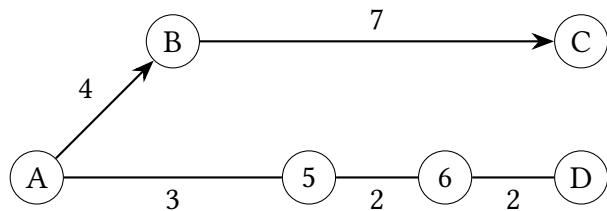


Abbildung 3.6: Beispiel Graph nach der Kontraktion des Knotens 4 [Eigene Darstellung]

Tabelle 3.12: Knotenstruktur des Beispielgraphen nach der Kontraktion des Knotens 4

ID	k	o	i	g
A	[0,0]	{id: "A-B", w: 4, p:[1]}, {id: "A-5", w: 3, p: [4]}	{id: "A-5", w: 3, p: [4]}	>2
B	[2,2]	{id: "B-C", w: 7, p:[2, 3]}	{id: "A-B", w: 4, p: [1]}	>2
C	[8,2]	-	{id: "B-C", w: 7, p:[2, 3]}	>2
D	[8,0]	{id: "D-6", w: 2}	{id: "D-6", w: 2}	>2
5	[4,0]	{id: "5-6", w: 2}, {id: "A-5", w: 3, p: [4]}	{id: "5-6", w: 2}, {id: "A-5", w: 3, p: [4]}	2
6	[6,0]	{id: "D-6", w: 2}, {id: "5-6", w: 2}	{id: "D-6", w: 2}, {id: "5-6", w: 2}	2

Durch die Kontraktion des Knotens 4 wurden die betroffenen Kanten des Knotens A und 5 aktualisiert.

Tabelle 3.13: node_next_branching_nodes nach der Kontraktion des Knotens 4

ID	o	i
1	{id: "B", w: 2}	{id: "A", w: 2}
2	{id: "C", w: 4}	{id: "B", w: 3}
3	{id: "C", w: 2}	{id: "B", w: 5}
4	{id: "A", w: 1}, {id: "5", w: 2}	{id: "A", w: 1}, {id: "5", w: 2}

Wie in Tabelle 3.13 dargestellt ist, ist der Knoten 4 als kontrahierter Knoten mit seinen entsprechenden Start- und Endknoten B und C und den entsprechenden Gewichten in dem node_next_branching_nodes-Objekt eingetragen. Hierbei ist zu beachten, dass es sich um eine ungerichtete Verbindung handelt, weshalb beide Knoten als Start- sowie Endknoten agieren können.

Tabelle 3.14: nodes_uncontracted_to_contracted nach der Kontraktion des Knotens 4

ID	o	i
A	[4]	[1,4]
B	[1]	[2,3]
C	[2,3]	[]
5	[4]	[4]

Tabelle 3.14 zeigt den Stand nach dem Kontrahieren des Knotens 4. Dabei ist die 4 sowohl bei dem Knoten A, als auch 5 in den eingehenden und ausgehenden Knoten eingetragen, da es sich um eine ungerichtete Verbindung handelt, welche in beide Richtungen traversierbar ist.

3.2.2.7 Schritt 6: Kontraktion des Knotens 6

In diesem Schritt wird der Knoten 6 kontrahiert. Die Auswirkungen sind dabei in Abbildung 3.7 und den Tabellen 3.15, 3.16 und 3.17 dargestellt.

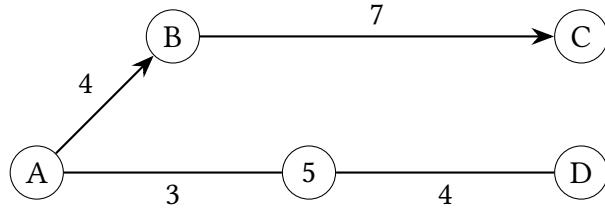


Abbildung 3.7: Beispiel Graph nach der Kontraktion des Knotens 6 [Eigene Darstellung]

Tabelle 3.15: Knotenstruktur des Beispielgraphen nach der Kontraktion des Knotens 6

ID	k	o	i	g
A	[0,0]	{id: "A-B", w: 4, p:[1]}, {id: "A-5", w: 3, p: [4]}	{id: "A-5", w: 3, p: [4]}	>2
B	[2,2]	{id: "B-C", w: 7, p:[2, 3]}	{id: "A-B", w: 4, p: [1]}	>2
C	[8,2]	-	{id: "B-C", w: 7, p:[2, 3]}	>2
D	[8,0]	{id: "A-D", w: 7, p: [6,5,4]}	{id: "A-D", w: 7, p: [4,5,6]}	>2
5	[4,0]	{id: "A-D", w: 7, p: [6,5,4]}, {id: "A-5", w: 3, p: [4]}	{id: "A-D", w: 7, p: [4,5,6]}, {id: "A-5", w: 3, p: [4]}	2

Durch die Kontraktion des Knotens 6 wurden die betroffenen Kanten des Knotens 5 und D aktualisiert.

Tabelle 3.16: node_next_branching_nodes nach der Kontraktion des Knotens 6

ID	o	i
1	{id: "B", w: 2}	{id: "A", w: 2}
2	{id: "C", w: 4}	{id: "B", w: 3}
3	{id: "C", w: 2}	{id: "B", w: 5}
4	{id: "A", w: 1}, {id: "D", w: 6}	{id: "A", w: 1}, {id: "D", w: 6}
6	{id: "A", w: 5}, {id: "D", w: 2}	{id: "A", w: 5}, {id: "D", w: 2}

Wie in Tabelle 3.16 dargestellt ist, ist der Knoten 6 als kontrahierter Knoten mit seinen entsprechenden Start- und Endknoten 5 und D und den entsprechenden Gewichten in dem node_next_branching_nodes-Objekt eingetragen.

Tabelle 3.17: nodes_uncontracted_to_contracted nach der Kontraktion des Knotens 6

ID	o	i
A	[4]	[1,4]
B	[1]	[2,3]
C	[2,3]	[]
5	[4]	[4]

Tabelle 3.17 zeigt den Stand nach dem Kontrahieren des Knotens 6. Dabei ist die 6 sowohl bei dem Knoten 5, als auch D in den eingehenden und ausgehenden Knoten eingetragen, da es sich um eine ungerichtete Verbindung handelt, welche in beide Richtungen traversierbar ist.

3.2.2.8 Schritt 7: Kontraktion des Knotens 5

In Schritt 7 wird der Knoten 5 kontrahiert. Die Auswirkungen sind dabei in Abbildung 3.8 und den Tabellen 3.18, 3.19 und 3.20 dargestellt.



Abbildung 3.8: Beispiel Graph nach der Kontraktion des Knotens 5 [Eigene Darstellung]

Tabelle 3.18: Knotenstruktur des Beispielgraphen nach der Kontraktion des Knotens 5

ID	k	o	i	g
A	[0,0]	{id: "A-B", w: 4, p:[1]}, {id: "A-D", w: 7, p: [4,5,6]}	{id: "A-D", w: 7, p: [6,5,4]}	>2
B	[2,2]	{id: "B-C", w: 7, p:[2, 3]}	{id: "A-B", w: 4, p: [1]}	>2
C	[8,2]	-	{id: "B-C", w: 7, p:[2, 3]}	>2
D	[8,0]	{id: "A-D", w: 7, p: [6,5,4]}	{id: "A-D", w: 7, p: [4,5,6]}	>2

Durch die Kontraktion des Knotens 5 wurden die betroffenen Kanten der Knoten A und D aktualisiert. Ebenfalls werden die pfad-Listen der Kanten aktualisiert. Hierbei ist zu beachten, dass die Pfade je nach Richtung des Traversierens der Kante umgekehrt sind. So ist der Pfad der kontrahierten Knoten auf der Kante "A-D" in der Richtung A->D der Pfad [4, 5, 6], während er in der Richtung D->A [6, 5, 4] ist. Dies kommt daher, dass beim Kontrahieren eines Knotens auf einer nicht gerichteten Verbindung einmal gerichtet in die eine Richtung und einmal in die andere Richtung die Funktion `MergeUndirectedEdges` (siehe Algorithmus 3) aufgerufen wird. Dies kombiniert die existierenden Pfade der Kanten und aktualisiert dabei entsprechend die p-Arrays der Kanten. Das hat zur Folge, dass die Pfade jeweils in die Richtung ihrer Kante eine korrekte Reihenfolge der kontrahierten Knoten enthalten.

Tabelle 3.19: node_next_branching_nodes nach der Kontraktion des Knotens 5

ID	o	i
1	{id: "B", w: 2}	{id: "A", w: 2}
2	{id: "C", w: 4}	{id: "B", w: 3}
3	{id: "C", w: 2}	{id: "B", w: 5}
4	{id: "A", w: 1}, {id: "D", w: 6}	{id: "A", w: 1}, {id: "D", w: 6}
6	{id: "A", w: 5}, {id: "D", w: 2}	{id: "A", w: 5}, {id: "D", w: 2}
5	{id: "A", w: 3}, {id: "D", w: 4}	{id: "A", w: 3}, {id: "D", w: 4}

In Tabelle 3.19 ist der Knoten 5 als kontrahierter Knoten mit seinen entsprechenden Start- und Endknoten A und D und den entsprechenden Gewichten in dem node_next_branching_nodes-

Objekt eingetragen. Ebenfalls wurden die Referenzen, in denen auf den Knoten 5 verwiesen wurde korrekt aktualisiert, sodass nun lediglich auf Knoten mit Buchstaben und damit auf Verzweigungsknoten, welche nicht kontrahierbar sind, verwiesen wird.

Die Struktur nodes_uncontracted_to_contracted ist in Tabelle 3.20 dargestellt.

Tabelle 3.20: nodes_uncontracted_to_contracted nach der Kontraktion des Knotens 5

ID	o	i
A	[4,5,6]	[1,4,5,6]
B	[1]	[2,3]
C	[2,3]	[]
D	[6,5,4]	[6,5,4]

Tabelle 3.20 zeigt den Stand nach dem Kontrahieren des Knotens 5.

3.2.2.9 Schritt 8: Vervollständigung der Hilfsstruktur

In diesem letzten Schritt sind alle Kontraktionsschritte abgeschlossen und es müssen lediglich noch die nicht kontrahierbaren Knoten in die Struktur node_next_branching_nodes eingetragen werden. Nach diesem Schritt sieht die Struktur node_next_branching_nodes wie in Tabelle 3.21 dargestellt aus:

Tabelle 3.21: Finaler Zustand node_next_branching_nodes

ID	o	i
1	{id: "B", w: 2}	{id: "A", w: 2}
2	{id: "C", w: 4}	{id: "B", w: 3}
3	{id: "C", w: 2}	{id: "B", w: 5}
4	{id: "A", w: 1}, {id: "D", w: 6}	{id: "A", w: 1}, {id: "D", w: 6}
6	{id: "A", w: 5}, {id: "D", w: 2}	{id: "A", w: 5}, {id: "D", w: 2}
5	{id: "A", w: 3}, {id: "D", w: 4}	{id: "A", w: 3}, {id: "D", w: 4}
A	{id: "A", w: 0}	{id: "A", w: 0}
B	{id: "B", w: 0}	{id: "B", w: 0}
C	{id: "C", w: 0}	{id: "A", w: 0}
D	{id: "D", w: 0}	{id: "A", w: 0}

3.2.3 Sonderfälle bei der Kontraktion

3.2.3.1 Rundweg

Ein Rundweg beschreibt in diesem Fall eine Situation wie sie in Abbildung 3.9 dargestellt ist. Dabei handelt es sich um eine Verbindung von Knoten beginnend und endend bei demselben Verzweigungsknoten (**B**). Dieser Fall tritt beispielsweise bei Kreisverkehren mit nur einer Ein- und derselben Ausfahrt, bei größeren Wendeschleifen oder auch in Wohngebieten auf.

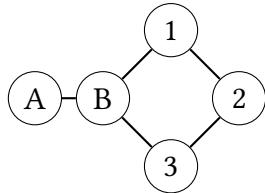


Abbildung 3.9: Sonderfall Rundweg [Eigene Darstellung]

Tritt ein solcher Fall auf, so kann der Algorithmus zur Graphenkontraktion nicht angewendet werden, da bei der Kontraktion des letzten Knotens alle Kanten zum gleichen Knoten (**B**) zeigen. Dies führt dazu, dass die Hilfsstrukturen nicht richtig aktualisiert werden können und die Routenberechnung nicht korrekt durchgeführt werden kann. Daher ist es notwendig, in solchen Fällen einen Knoten auf dem Rundweg zu belassen, um die korrekte Funktionalität des Algorithmus zu gewährleisten. Dieser Knoten wird für den weiteren Verlauf als Verzweigung betrachtet und bleibt im kontrahierten Graphen enthalten. Zur Bestimmung dieses Knotens muss eine weitere Abfrage vor dem Kontrahieren eines Knotens eingefügt werden. Diese Abfrage ist in Listing 3.14 dargestellt.

Listing 3.14: Überprüfung auf Rundweg

```

if (
    currNodeData.i.every(incomingEdge =>
        currNodeData.o.every(outgoingEdge => incomingEdge.id === outgoingEdge.id)
    )
) {
    return;
}
}

```

Es wird überprüft, ob alle eingehenden Kanten des Knotens zu allen ausgehenden Kanten des Knotens identisch sind. Ist dies der Fall, so handelt es sich um einen Rundweg und der Knoten wird nicht entfernt.

3.3 Routing

In diesem Abschnitt wird der implementierte Algorithmus zur Berechnung der kürzesten Route durch den kontrahierten Graphen näher erläutert. Dabei wird auf das grundlegende Vorgehen, sowie die Optimierungen eingegangen, welche die Laufzeit der Routenberechnung reduzieren.

3.3.1 Heuristiken

Im Rahmen der verwendeten Heuristiken kommt lediglich die Luftliniendistanz zwischen zwei Punkten zum Einsatz. Diese Heuristik wird durch die Haversine-Formel bestimmt. Die verwendete Haversine-Formel ist in der Formel 2.7 dargestellt.

Wie in der Literatur gezeigt, gibt es unterschiedliche, äquivalente Formulierung der Haversine-Formel, welche nachfolgend in den Formeln 3.1, 3.2 und 3.3 dargestellt sind.

$$d = 2R \arcsin \left(\sqrt{\sin^2 \left(\frac{\varphi_2 - \varphi_1}{2} \right) + \left(1 - \sin^2 \left(\frac{\varphi_2 - \varphi_1}{2} \right) - \sin^2(\varphi_m) \right) \cdot \sin^2 \left(\frac{\lambda_2 - \lambda_1}{2} \right)} \right) \quad (3.1)$$

$$d = 2R \arcsin \left(\sqrt{\sin^2 \left(\frac{\varphi_2 - \varphi_1}{2} \right) \cdot \cos^2 \left(\frac{\lambda_2 - \lambda_1}{2} \right) + \cos^2(\varphi_m) \cdot \sin^2 \left(\frac{\lambda_2 - \lambda_1}{2} \right)} \right) \quad (3.2)$$

$$d = 2R \arcsin \left(\sqrt{\frac{1 - \cos(\varphi_2 - \varphi_1) + \cos \varphi_1 \cdot \cos \varphi_2 \cdot (1 - \cos(\lambda_2 - \lambda_1))}{2}} \right) \quad (3.3)$$

Diese wurden im Rahmen der Projektarbeit jeweils implementiert und mit verschiedenen vorgegebenen Distanzen getestet. Dabei wurde festgestellt, dass die Formulierungen zu unterschiedlichen Ergebnissen führen. Die Ergebnisse dieser Tests sind in Tabelle 3.22 dargestellt.

Tatsächliche Distanz [km]	Formeln 2.7, 3.1 3.2	Formel 3.3
0	0	0
0.01	0.01	0
0.02	0.02	0
0.05	0.04999999999999996	0
0.1	0.09999999999999999	0.09504164755344391
0.2	0.19999999999999998	0.19008329510688782
0.5	0.4999999999999994	0.5029131272405261
1	0.9999999999999999	1.0013258907237497
2	1.9999999999999998	2.0003952716813727
5	4.999999999999999	5.000310798586052
10	9.999999999999998	10.000169969309063
20	19.999999999999996	19.999888295462796
50	50.000000000000001	50.00003689154773
100	100.000000000000001	99.99998345474181
200	200.000000000000003	199.9999894979184
500	499.9999999999999	499.9999869285745
1000	999.9999999999998	999.999981538936

Tabelle 3.22: Vergleich der verschiedenen Haversine-Implementierungen

Wie in Tabelle 3.22 dargestellt, liefern die Formeln 2.7, 3.1 und 3.2 die gleichen Ergebnisse, während die Formel 3.3 zu abweichenden Ergebnissen führt. Diese Abweichungen sind deutlich größer, als die der anderen Formeln, was darauf schließen lässt, dass bei der Implementierung der Formel 3.3 ein numerisches Problem vorliegt. Dieses Problem wurde jedoch im Rahmen dieser Projektarbeit nicht weiter untersucht. Da die Formeln 2.7, 3.1 und 3.2 die gleichen Ergebnisse liefern, wurde die Formel 2.7 für die Implementierung der Haversine-Formel verwendet.

3.3.2 Optimierungen

In diesem Abschnitt werden verschiedene Optimierungen vorgestellt, welche im Rahmen der Routenberechnung implementiert wurden. Diese Optimierungen dienen dazu, die Laufzeit der Routenberechnung zu reduzieren und die Effizienz des Algorithmus zu steigern.

3.3.2.1 Problematik bei kontrahierten Knoten

Durch die implementierte Graphenkontraktion wird die Anzahl der Knoten und Kanten im Graphen zwar deutlich kontrahiert, was zu einer allgemein schnelleren Routenberechnung führt, allerdings sei angemerkt, dass sich durch die Kontraktion der Knoten und Kanten ein neues Problem ergibt, welches im nicht kontrahierten Graphen nicht existiert. So kann es vorkommen, dass sich ein Start- beziehungsweise Endknoten (z.B. Knoten 1) auf einem kontrahierten Teilstück befindet, siehe Abbildung 3.10.



Abbildung 3.10: Beispiel Knoten auf einer kontrahierten Kante [Eigene Darstellung]

Knoten A und B stellen dabei Verzweigungen, also nicht kontrahierte Knoten dar. Knoten 1 stellt einen kontrahierten Knoten dar, welcher in dem kontrahierten Graphen nicht mehr existiert. Bei einer Routenberechnung ab dem Knoten 1 müssen mehrere Möglichkeiten in Betracht gezogen werden. Ist der Knoten auf einer gerichteten Verbindung, so existiert ein möglicher Startpunkt im kontrahierten Graphen. Dies ist die nächste Verzweigung in Richtung der Verbindung. Ist die Verbindung jedoch ungerichtet, so existieren zwei mögliche Startpunkte, welche beide in Betracht gezogen werden müssen. Dies führt zu einer erhöhten Komplexität bei der Routenberechnung. Da für den Zielknoten ebenfalls die gleiche Problematik besteht, müssen auch hier alle möglichen Endpunkte im kontrahierten Graphen in Betracht gezogen werden. Dies führt zu einer erhöhten Anzahl an Routenberechnungen, da für jeden möglichen Start- und Endpunkt eine Route berechnet werden muss, um zu bestimmen, welche Route die kürzeste ist. Hierbei ist zu beachten, dass bis zu **vier** Routen berechnet werden müssen, um die kürzeste Route zu bestimmen.

3.3.2.2 Verhindern von unnötigen Routenberechnungen

Um der Problematik im vorangegangenen Abschnitt entgegen zu wirken, wird bereits vor einer Routenberechnung geprüft, ob die Route überhaupt berechnet werden muss. Dies geschieht über eine Abschätzung der kleinst möglichen Routen-Kosten basierend auf den Kosten zum Start- und Endpunkt im kontrahierten Graphen, sowie die Luftliniendistanz zwischen den beiden Punkten. In der Formel 3.4 ist die Berechnung der geschätzten Kosten für die Route dargestellt.

$$\text{estimatedCosts} = \text{startCosts} + \text{endCosts} + \text{haversineDistance}(\text{start}, \text{end}) \quad (3.4)$$

Sollten die geschätzten Kosten der Route bereits größer sein, als die Kosten der aktuell kürzesten Route, so kann die Routenberechnung abgebrochen werden, da die Kosten der Route nicht geringer werden können. Dies führt zu einer Reduzierung der Anzahl an Routenberechnungen und damit zu einer Reduzierung der Laufzeit.

Um diesen Abgleich durchzuführen muss allerdings die erste Route immer berechnet werden, da ansonsten keine Abschätzung möglich ist. Zuerst eine Schätzung aller Start- und Endknoten Kosten durchzuführen ist dabei nicht möglich, da die kleinsten geschätzten Kosten nicht auch die kürzesten Pfad-Kosten sein müssen. Dies liegt daran, dass vorab keine Aussage über den Verlauf des Graphen zwischen dem Start- und Endknoten getroffen werden kann. Daher ist es notwendig, die erste Route zu berechnen, um eine Abschätzung der Kosten im weiteren Verlauf durchführen zu können.

Ebenfalls kann die Funktion für das Routing optimiert werden. Hierfür kann zum Beispiel frühzeitig abgebrochen werden, wenn eine neu berechnete Route länger ist, als eine bereits gefundene kürzeste Route. Dies kann beispielsweise über eine PriorityQueue realisiert werden, wobei die Funktion zum auslesen des nächsten Knotens immer den Knoten mit den geringsten Kosten zurück gibt. Bei dieser Implementierung kann abgebrochen werden, sobald die Kosten des nächsten Knoten zu groß sind. Sollte dieser Fall eintreten, so kann die weitere Routenberechnung abgebrochen werden, da auch alle anderen Knoten in der Queue größere Kosten haben müssen. Eine andere Möglichkeit besteht darin erst gar keine weiteren Knoten hinzuzufügen, deren Kosten größer sind, als die Gesamtkosten des aktuell kürzesten Weges.

3.3.2.3 Anpassungen an den Basis-Algorithmen

Ebenfalls können die Routing-Funktionen angepasst werden, um die Laufzeit weiter zu reduzieren. Hierbei kann bereits frühzeitig abgebrochen werden, wenn der Zielknoten nicht gefunden wurde und alle weiteren Möglichkeiten höhere Kosten haben, als die aktuell kürzeste Route.

In diesem Zusammenhang ist jedoch zu beachten, dass diese Optimierung lediglich mit Algorithmen basierend auf Kosten arbeitet. Sollte beispielsweise die Breitensuche verwendet werden, so ist diese Optimierung nicht möglich. Bei Algorithmen wie dem A*- oder Dijkstra-Algorithmus ist sie jedoch möglich und führt dadurch zu einer Reduzierung der Laufzeit.

Optimierung mit PriorityQueue Um dieses frühzeitige Abbrechen zu realisieren, kann eine PriorityQueue verwendet werden. Diese PriorityQueue wird dabei verwendet, um die Knoten in der Reihenfolge der Kosten zu sortieren. Dabei wird der Knoten mit den geringsten Kosten immer zuerst aus der Queue entnommen. Sollten die Kosten dieses Knotens bereits größer sein, als die Kosten der aktuell kürzesten Route, so kann die Routenberechnung abgebrochen werden, da alle weiteren Knoten in der Queue ebenfalls höhere Kosten haben müssen.

Optimierung ohne spezielle Queue Struktur Durch die Verwendung einer PriorityQueue kann die Laufzeit der Routenberechnung zwar reduziert werden, allerdings ist damit die Implementierung der Queue vorgegeben. Eine andere Möglichkeit besteht darin, die Kosten der Knoten vor dem Hinzufügen in die Queue zu betrachten, um basierend darauf bereits zu entscheiden, ob der Knoten in die Queue aufgenommen werden soll. Sollten die Kosten des Knotens bereits größer sein, als die Kosten der aktuell kürzesten Route, so kann das Betrachten des Knotens verworfen werden und er wird nicht in die Queue aufgenommen. Dies führt zu einer Reduzierung der Anzahl an Knoten in der Queue und damit zu einer Reduzierung der Laufzeit. Es ist zu beachten, dass dabei die gleiche Funktionalität und Laufzeit wie bei der Verwendung einer PriorityQueue

erreicht wird, allerdings ohne die Verwendung einer speziellen Queue-Struktur. In einem großen, stark verzweigten Graphen kann mit diesem Ansatz auch der Speicherbedarf im Vergleich zu der Lösung mit einer PriorityQueue reduziert werden, da Knoten mit hohen Kosten gar nicht erst in die Queue aufgenommen werden.

3.3.3 Beschreibung des Algorithmus

Die Funktion `CalculateOptimizedPath` (siehe Algorithmus 13) berechnet die kürzeste Route zwischen zwei Knoten unter Verwendung eines angegebenen Routingalgorithmus. Zunächst wird anhand des `algo`-Parameters entschieden, welcher Algorithmus verwendet werden soll. Sollte ein ungültiger Algorithmus angegeben werden, wird eine Fehlermeldung ausgegeben.

Die Start- und Endknoten-IDs werden anschließend in Strings umgewandelt, um eine konsistente Verarbeitung zu gewährleisten. Falls der Start- und der Endknoten identisch sind, wird direkt ein trivialer Pfad mit Kosten von null zurückgegeben.

Danach werden die möglichen Start- und Endknoten für die Pfadberechnung ermittelt. Es wird eine leere Liste `paths` erstellt, in der alle berechneten Pfade gespeichert werden, sowie eine Variable `bestPath`, um den aktuell besten Pfad zu speichern.

Nun wird für jede mögliche Kombination von Start- und Endknoten geprüft, ob ein direkter Pfad existiert. Dies ist der Fall, wenn der Endknoten bereits auf dem Weg zum Startknoten liegt. Daraufhin wird ein verkürzter Pfad mit den entsprechenden Kosten berechnet und direkt als potenziell bester Pfad gespeichert.

Falls kein direkter Pfad existiert, wird geprüft, ob eine Berechnung des Pfads notwendig ist. Falls nicht, wird diese Kombination aus Start- und Endknoten übersprungen. Andernfalls wird der gewählte Routingalgorithmus aufgerufen, um den besten Weg zwischen den beiden Knoten zu finden. Dabei wird sichergestellt, dass nur Wege berücksichtigt werden, die eine niedrigere Gesamtkosten als der bisher beste Pfad haben.

Die berechneten Pfade werden anschließend überprüft und der beste Pfad aktualisiert. Falls nach allen Berechnungen kein gültiger Pfad gefunden wurde, wird ein Misserfolg zurückgegeben. Andernfalls wird der endgültige Pfad aufgebaut, wobei zwischen kontrahierten und vollständigen Pfaden unterschieden wird.

Abschließend wird die gesamte Berechnungszeit erfasst und das Ergebnis zurückgegeben, einschließlich der gesamten Pfadlänge, des berechneten Pfads und der Laufzeit.

3.3.3.1 Sonderfall mehrerer Wege zwischen zwei Verzweigungen

Ein Sonderfall, der bei der Routenberechnung auftreten kann, ist die Existenz mehrerer Wege zwischen zwei Verzweigungen. Die trifft beispielsweise in Wohngebieten oder auch bei Fußgängerbewegen auf. Ein Beispiel ist in Abbildung 3.11 dargestellt.

Algorithm 13 CalculateOptimizedPath

```

1: function CALCULATEOPTIMIZEDPATH(startNodeId, endNodeId, algo)
2:   startTime ← CurrentTime()
3:   if algo ≠ ValidAlgorithms then return Error("Ungültiger Routingalgorithmus")
4:   end if
5:   startNodeIdStr ← ToString(startNodeId)
6:   endNodeIdStr ← ToString(endNodeId)
7:   if startNodeIdStr = endNodeIdStr then
8:     return {path: [startNodeId], cost: 0, time: CurrentTime() - startTime}
9:   end if
10:  possibleStartNodes ← GetPossibleStartNodes(startNodeId)
11:  possibleEndNodes ← GetPossibleEndNodes(endNodeId)
12:  paths ← []; bestPath ← null
13:  for all start ∈ possibleStartNodes do
14:    for all end ∈ possibleEndNodes do
15:      if DirectPathExists(start, end) and end IsOnWayTo start then
16:        shortenedPath ← CalculateShortenedPath(start, end)
17:        paths.Add(shortenedPath)
18:        if BetterThan(shortenedPath, bestPath) then
19:          bestPath ← shortenedPath
20:        end if
21:        continue
22:      end if
23:      if not IsPathCalculationRequired(start, end) then
24:        continue
25:      end if
26:      calculatedPath ← CallRoutingAlgorithm(algo, start, end, bestPath.cost)
27:      if calculatedPath ≠ null then
28:        paths.Add(calculatedPath)
29:        if BetterThan(calculatedPath, bestPath) then
30:          bestPath ← calculatedPath
31:        end if
32:      end if
33:    end for
34:  end for
35:  if bestPath = null then
36:    return {success: false, message: "Kein gültiger Pfad gefunden"}
37:  end if
38:  finalPath ← (bestPath.isContracted) ? ExpandContractedPath(bestPath.path) :
39:    bestPath.path
40:  return FinalFormattedPathResult(finalPath, bestPath.cost, CurrentTime() - startTime)
end function

```



Abbildung 3.11: Mehrere Wege zwischen zwei Verzweigungen [Eigene Darstellung]

Dieser Fall ist nicht weiter problematisch für die Berechnung, da die kürzere Möglichkeit entsprechend der Kosten verwendet wird. Allerdings muss bei der Rekonstruktion des kompletten Pfades darauf geachtet werden, dass der richtige, in diesem Fall der kürzeste aller Möglichkeiten in den Pfad integriert wird, sodass die Schlussendliche Ausgabe wie in Abbildung 3.12 aussieht.

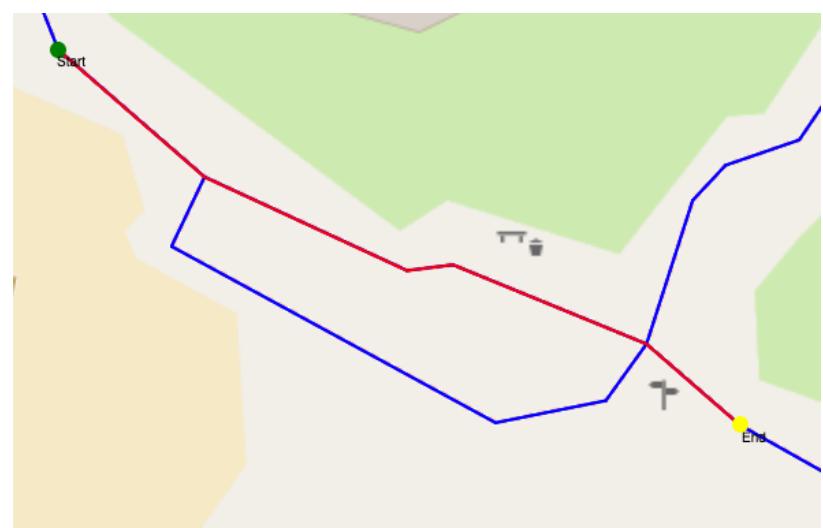


Abbildung 3.12: Mehrere Wege zwischen zwei Verzweigungen richtige Rekonstruktion [Eigene Darstellung]

Ebenfalls tritt dieser Fall auf, wenn sich zum Beispiel der Start Knoten auf einem solchen Weg befindet. In diesem Fall muss ebenfalls der richtige Weg ausgewählt werden, um die korrekte Route zu berechnen. Dafür muss zu jeder Kante zwischen den beiden Verzweigungen überprüft werden, ob sich der entsprechende Knoten auf dem Weg befindet. Ein Beispiel ist in Abbildung 3.13 dargestellt.

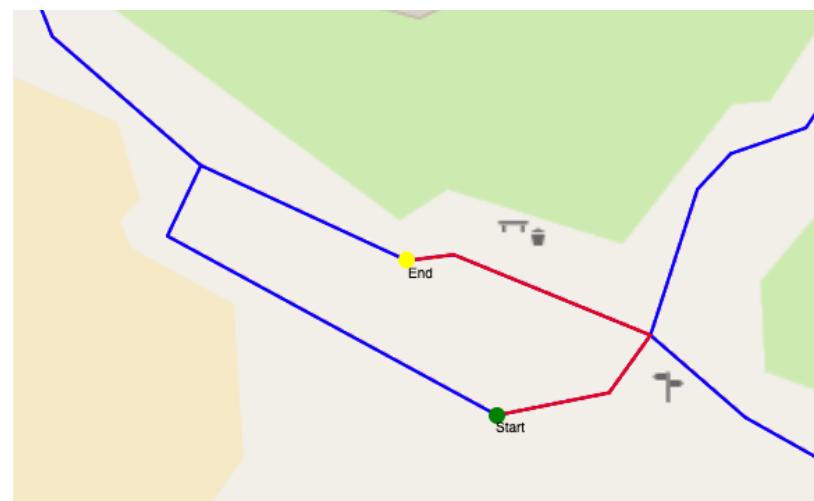


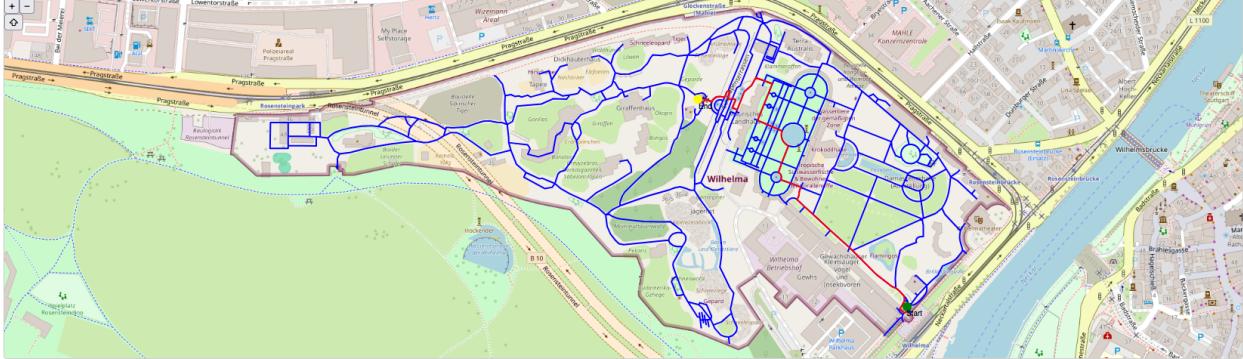
Abbildung 3.13: Mehrere Wege zwischen zwei Verzweigungen Start- und Endknoten auf den Wegen [Eigene Darstellung]

3.4 Weboberfläche

In diesem Abschnitt werden die verschiedenen Features der Weboberfläche vorgestellt, welche im Rahmen dieser Projektarbeit implementiert wurden. Die Weboberfläche dient dabei als Benutzeroberfläche zur Interaktion mit dem implementierten Algorithmus zur Routenberechnung. In Abbildung 3.14 ist die Weboberfläche dargestellt.

Route Optimization Demo

Compare different pathfinding algorithms on real map data



Results

Last Algorithm: **Optimized A_star**

Last Algorithm Runtime: **3.00 ms**

Path Length: **619.74 m**

Nodes in Path: **106**

Start/End Points

-
-
-
-

Basic Routing

-
-
-

Optimized Routing

-
-
-

Map Options

- Durchsuchen... Keine Da...ewählt.
-
- Select Node to show possible routing nodes
- Show Contracted Graph
- Show Nodes of Path
- Show Only Interactions Layer

Simulation Controls

-
-
-
-
- Speed:

Abbildung 3.14: Weboberfläche zur Interaktion mit dem Algorithmus [Eigene Darstellung]

50

3.4.1 Kartenansicht

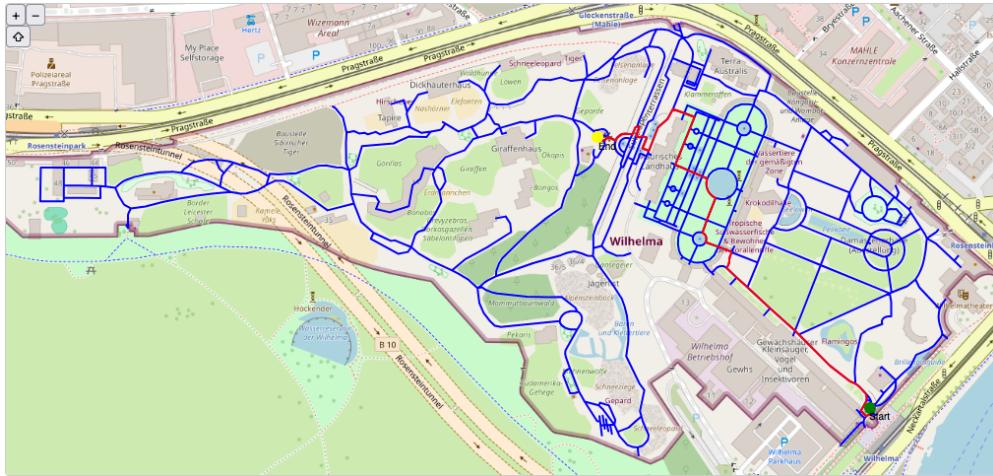


Abbildung 3.15: Kartenansicht der Weboberfläche [Eigene Darstellung]

Die Kartenansicht (siehe Abbildung 3.15) dient zur Darstellung der Karte, auf der die Routenberechnung durchgeführt wird. Dabei wird die Karte über die Bibliothek OpenLayers dargestellt. Die Karte wird dabei in verschiedene Schichten (Layer) unterteilt, um unterschiedliche Daten darzustellen.

3.4.1.1 OSM-Layer

Listing 3.15: OpenLayers Schichten Aufbau

```
this.map = new ol.Map({
  target: 'div_map',
  view: new ol.View({
    center: [0, 0],
    zoom: 2
  }),
  layers: [
    new ol.layer.Tile({
      source: new ol.source.OSM({
        attributions: []
      })
    })
  ]
});
```

In Listing 3.15 ist der Grundaufbau des `map` Objekts dargestellt, durch welches die Kartenansicht generiert wird. Dieses enthält bei der Initialisierung lediglich eine Schicht. Dabei handelt es sich um die Hintergrund Kartenbilder aus OpenStreetMap. Hierbei wird die `ol.source.OSM` Quelle verwendet, um die Kartenbilder von OpenStreetMap zu laden. Die Kartenbilder werden dabei über die `ol.layer.Tile` Schicht dargestellt. Diese Schicht dient zur Darstellung von Kartenbildern, welche in Kacheln geladen werden. Ebenfalls werden die Standardattributions von OSM durch `attributions: []` deaktiviert, um deren Anzeige zu verhindern.

3.4.1.2 Interaktions-Layer

Listing 3.16: Interaktions-Layer

```
this.vecInteractions = new ol.source.Vector();

this.layerInteractions = new ol.layer.Vector({ source: this.vecInteractions });

...

this.map.addLayer(this.layerInteractions);
```

In Listing 3.16 ist der Aufbau des Interaktions-Layers dargestellt. Dieser dient zur Darstellung der Interaktionen auf der Karte, wie beispielsweise die Auswahl von Start- und Endpunkten. Dabei wird eine `ol.source.Vector` Quelle verwendet, um die Interaktionen zu speichern. Diese Quelle wird anschließend in eine `ol.layer.Vector` Schicht eingebettet, um die Interaktionen auf dieser Schicht darzustellen. Die Schicht wird anschließend der Karte hinzugefügt, um die Interaktionen auf der Karte anzuzeigen. Folgende weitere Informationen werden auf der Interaktions-Schicht dargestellt:

- Start- und Endpunkte der Routenberechnung (siehe Abschnitt 3.4.2.2)
- Gefundene Routen (siehe Abbildung 3.25b)
- Linie für den Hover-Effekt bei der Auswahl von Knoten (siehe Abschnitt 3.4.2.2)
- Knoten eines gefundenen Pfades (siehe Abschnitt 3.4.2.4)
- Auswahl eines Knotens für die Darstellung der Nachbarn (siehe Abschnitt 3.4.2.4)
- Simulations-Darstellungen (siehe Abschnitt 3.4.2.5)

3.4.1.3 Layer für vollständigen Graphen

Der Aufbau des Layers für den vollständigen Graphen ist identisch zu dem Aufbau des Interaktions-Layers. Es wird ebenfalls ein `ol.source.Vector` verwendet, um die Kanten und Knoten des vollständigen Graphen zu speichern. Diese Quelle wird anschließend in eine `ol.layer.Vector` Schicht eingebettet, um die Kanten und Knoten des vollständigen Graphen darzustellen, welche dann ebenfalls der Karte hinzugefügt wird.

Durch die Darstellung des vollständigen Graphen können die Straßenverläufe des Graphen betrachtet werden. Ebenfalls ist direkt möglich die Grenzen des Datensatzes zu erkennen und zu überprüfen, ob die Daten korrekt geladen wurden. Allerdings sei angemerkt, dass die Darstellung des vollständigen Graphen sehr inperformant ist, da sehr viele Kanten dargestellt werden müssen. Dies führt zu einer starken Beeinträchtigung der Kartenansicht und der Interaktionen mit der Karte und sollte deshalb bei größeren Datenmengen deaktiviert werden. Eine Darstellung des vollständigen Graphen ist in Abbildung 3.16 gezeigt.

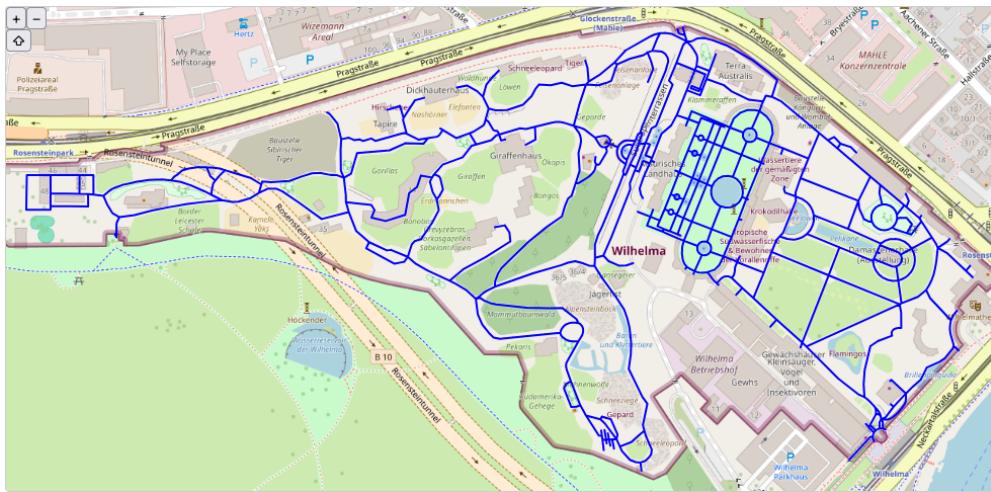


Abbildung 3.16: Darstellung des vollständigen Graphen [Eigene Darstellung]

3.4.1.4 Layer für den kontrahierten Graphen

Der Aufbau des Layers für den kontrahierten Graphen ist ebenfalls identisch zu den anderen beiden bereits vorgestellten. Es wird ebenfalls ein `ol.source.Vector` verwendet, um die Kanten und Knoten des kontrahierten Graphen zu speichern. Diese Quelle wird anschließend in eine `ol.layer.Vector` Schicht eingebettet, um die Kanten und Knoten des kontrahierten Graphen darzustellen, welche dann ebenfalls der Karte hinzugefügt wird.

Durch die Darstellung des kontrahierten Graphen, kann die Kontraktion der Knoten und Kanten im Graphen betrachtet werden und welche Auswirkungen diese haben. Ebenfalls ist es möglich die Korrektheit der Kontraktion und das richtige Anzeigen der nächsten Nachbarn im kontrahierten Graphen zu einem gegebenen Knoten zu überprüfen. Allerdings sei angemerkt, dass die Darstellung des kontrahierten Graphen ebenfalls inperformant ist und nicht für größere Datenmengen geeignet ist. Diese sollte primär für Debugging-Zwecke verwendet werden. Eine Darstellung des kontrahierten Graphen ist in Abbildung 3.24 gezeigt.

3.4.2 Interaktions-Panel

Das Interaktions-Panel (siehe Abbildung 3.17) dient zur Interaktion mit der Routenberechnung. Dabei können verschiedene Einstellungen vorgenommen werden, um die Berechnung zu beeinflussen. Im Folgenden werden die verschiedenen Einstellungen und Funktionen des Interaktions-Panels im Detail vorgestellt.

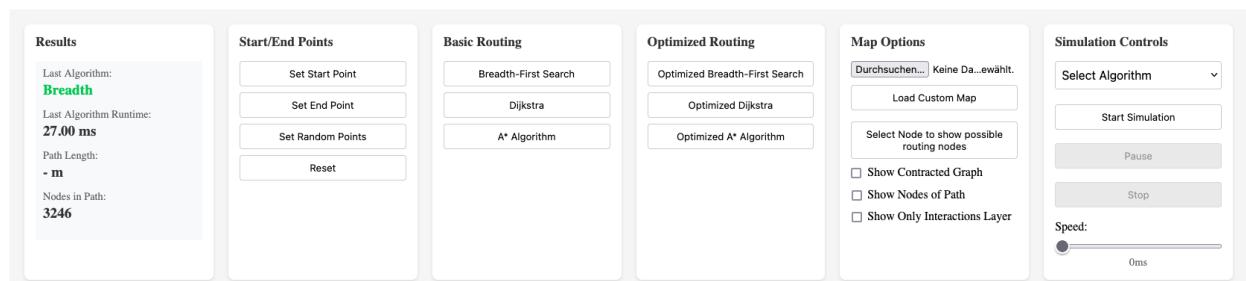


Abbildung 3.17: Interaktions-Panel [Eigene Darstellung]

3.4.2.1 Informationen zur Routenberechnung

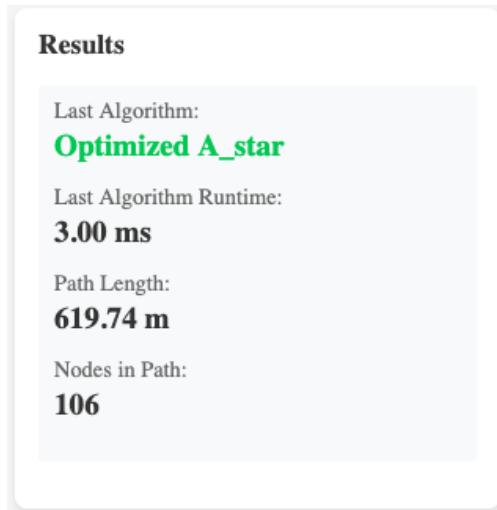


Abbildung 3.18: Ergebnisse der Routenberechnung [Eigene Darstellung]

Informationen zur aktuell dargestellten Route können über das Results-Subpanel (siehe Abbildung 3.18) abgerufen werden. Dabei wird der Name des verwendeten Algorithmus, die Dauer der Berechnung, die Kosten beziehungsweise die Länge, der Route sowie die Anzahl der berechneten Knoten im Pfad angezeigt.

Zu beachten ist hierbei, dass die Berechnungsdauer Ungenauigkeiten aufweist, da diese über den Browser berechnet wird. Die Berechnungsdauer kann daher stark schwanken und ist nicht als genaue Angabe zu verstehen. Ebenfalls ist die Genauigkeit der verwendeten Bibliothek Performance zur Zeitmessung im Browser auf Millisekunden beschränkt [30]. Zur genaueren Zeitmessung ist daher eine Messung auf Serverseite notwendig. Dies wird durch Ausführung in einer Node.js Umgebung ermöglicht, welche in Kapitel 4 näher erläutert wird.

3.4.2.2 Dynamische Auswahl Start und Endpunkte

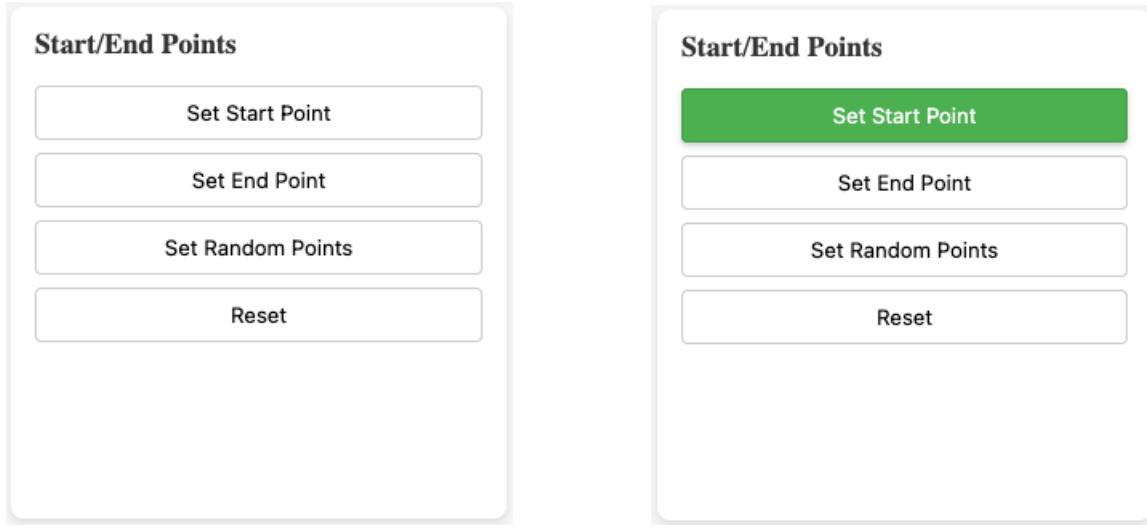


Abbildung 3.19: Start- und Endpunkt auswahl in der Weboberfläche [Eigene Darstellung]

Über das Start/End Points-Subpanel (siehe Abbildung 3.19a) können Start- und Endpunkte für die Routenberechnung dynamisch ausgewählt werden. Durch klicken auf den Button Set Start Point oder Set End Point wird der entsprechende Button aktiviert und es kann ein Punkt auf der Karte ausgewählt werden.

Die Auswahl aktiviert ebenfalls einen Hover-Effekt auf der Karte, welcher eine rote Linie von der Mausposition zu dem nächsten Knoten auf der Karte zeichnet. Dies ermöglicht eine genaue Auswahl des gewünschten Knotens. In Abbildung 3.20 ist die aktive Auswahl dargestellt. Die Bestimmung des nächsten Knotens erfolgt dabei über die Nutzung eines KD-Trees, welcher zu einem gegebenen Punkt (Mausposition) den nächsten Knoten im Graphen berechnet. Der Hover Effekt ist nur verfügbar, wenn das Zoomlevel der Karte ausreichend hoch ist, um die Knoten auf der Karte sinnvoll auszuwählen.

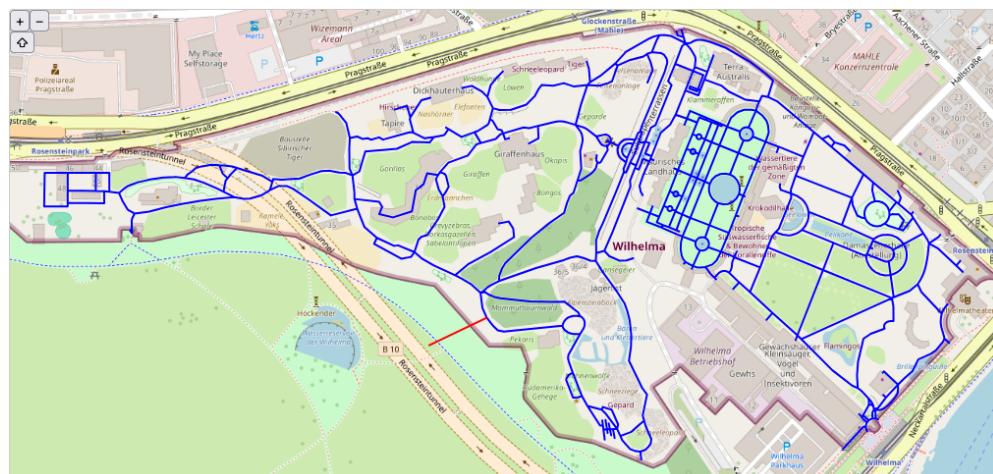


Abbildung 3.20: Hover Effekt bei der Auswahl von Start- und Endpunkten [Eigene Darstellung]

Die Auswahl der Start- und Endpunkte wird durch einen Klick auf die Karte bestätigt. Dabei wird der ausgewählte Punkt als Start- oder Endpunkt für die Routenberechnung übernommen. In Abbildung 3.21 ist die aktive Auswahl dargestellt.



Abbildung 3.21: Auswahl von Start- und Endpunkten auf der Karte [Eigene Darstellung]

Ebenfalls bietet die Weboberfläche die Möglichkeit zufällige Start- und Endknoten über die Button `Set Random Points` zu setzen. Dabei werden zufällige Knoten aus dem Graphen ausgewählt und als Start- beziehungsweise Endpunkt für die Routenberechnung gesetzt.

Weiterhin findet sich der Button `Reset` in dem Subpanel, welcher den Kartenzustand zurücksetzt.

3.4.2.3 Auswahl verschiedener Algorithmen

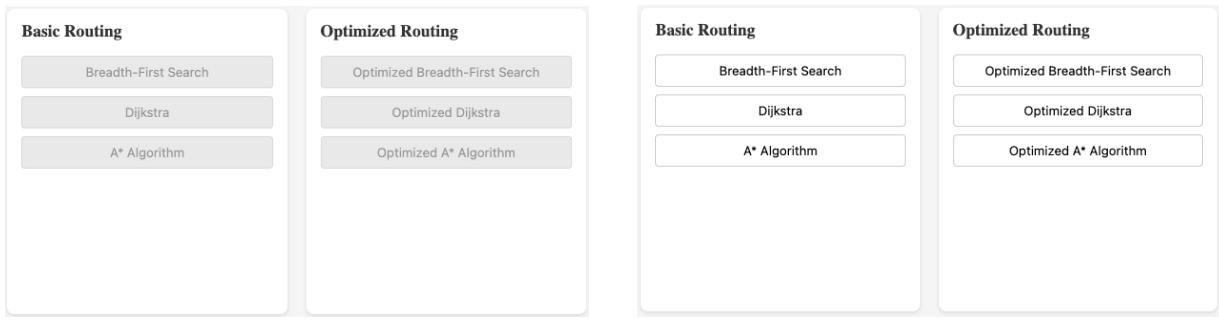


Abbildung 3.22: Start- und Endpunktauswahl in der Weboberfläche [Eigene Darstellung]

Über die `Basic Routing`- und `Optimized Routing`-Subpanels (siehe Abbildung 3.22a) kann der gewünschte Algorithmus für die Routenberechnung ausgewählt werden. Dabei stehen die Algorithmen `A*`, `Dijkstra` und `Breadth First Search` sowohl als Grund-, als auch Optimierte-Varianten zur Auswahl. Durch klicken auf den entsprechenden Button wird der Algorithmus ausgewählt und die Routenberechnung mit dem ausgewählten Algorithmus durchgeführt.

Die Grundvarianten verwenden dabei die Implementierung der Algorithmen ohne Optimierungen und nutzen alle Knoten im Gesamtgraphen ohne Kontraktion. Die Optimierten Varianten verwenden hingegen die implementierten Optimierungen, um die Laufzeit der Routenberechnung zu reduzieren. Dabei wird der lediglich die kontrahierte Variante des Graphen verwendet und die Funktion `CalculateOptimizedPath` (siehe Algorithmus 13) entsprechend der Auswahl aufgerufen, um die Routenberechnung durchzuführen.

Solange noch kein Start- und Endknoten auf der Karte ausgewählt ist, sind die Knöpfe dieser Subpanels deaktiviert. Sobald jedoch Start- und Endknoten ausgewählt sind, werden die Knöpfe aktiviert und die Auswahl des Algorithmus kann vorgenommen werden.

3.4.2.4 Karten Optionen

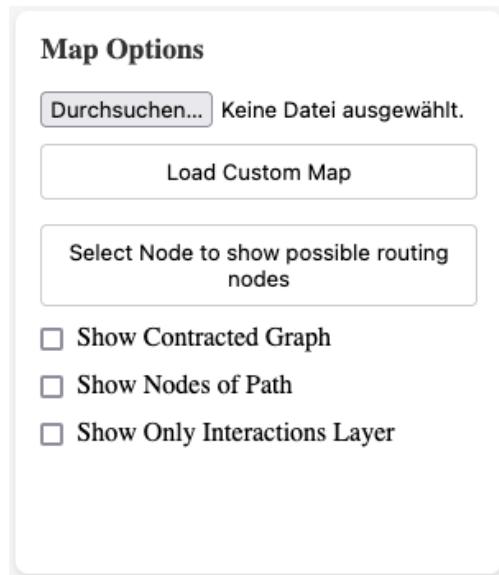


Abbildung 3.23: Karten Optionen in der Weboberfläche [Eigene Darstellung]

Über das `Map Options`-Subpanel (siehe Abbildung 3.23) können verschiedene Optionen für die Karte ausgewählt werden.

Kontrahierten Graphen anzeigen Über die Checkbox `Show Contracted Graph` lässt sich der angezeigte Graph zwischen dem kompletten und dem kontrahierten Graphen umschalten. Dabei wird der kontrahierte Graph auf der Karte dargestellt, um die Kontraktion der Knoten und Kanten zu visualisieren. Dies ermöglicht es, die Kontraktion des Graphen zu betrachten und die Auswirkungen auf die Routenberechnung zu verstehen. In Abbildung 3.24 ist der kontrahierte Graph auf der Karte dargestellt.

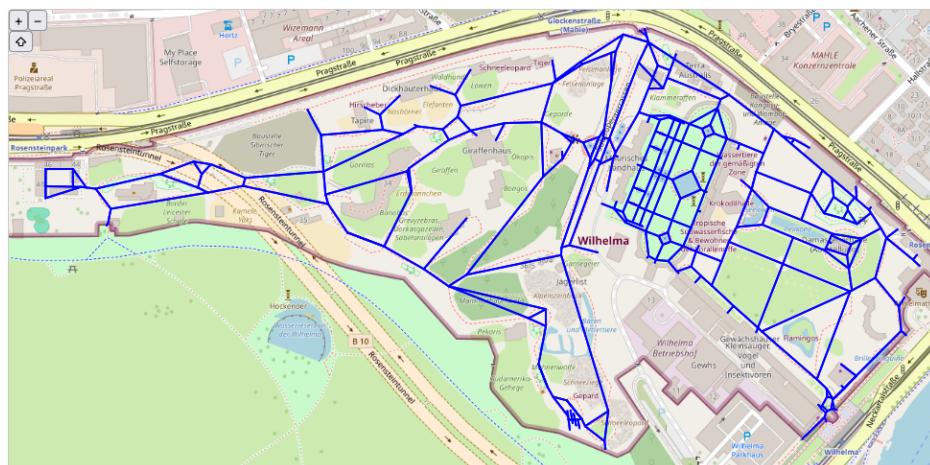


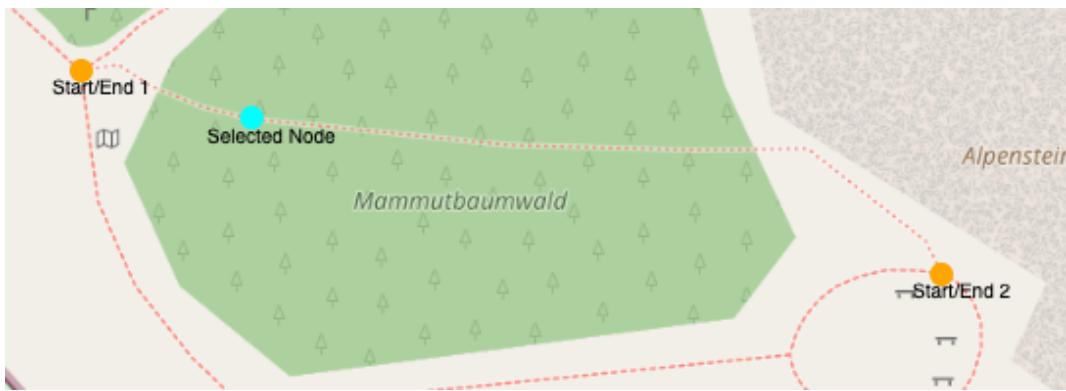
Abbildung 3.24: Kontrahierter Graph auf der Karte [Eigene Darstellung]

Nur Interaktions Layer anzeigen Mit der Checkbox Show Only Interaction Layer kann die Anzeige zwischen allen Layers und nur dem Interaktions-Layer umgeschaltet werden. Bei aktiverter Checkbox wird nur der Interaktions-Layer auf der Karte dargestellt. Dadurch wird die Übersichtlichkeit auf der Karte erhöht und nur die relevanten Informationen zu einer Route sind ersichtlich. Genauere Details zum Interaktions-Layer werden in Abschnitt 3.4.1.2 behandelt. Durch das aktivieren der Checkbox werden die anderen Layer ausgeblendet, wodurch der Graph nicht länger sichtbar ist. In Abbildung 3.25 ist zu sehen, wie sich das Verhalten der Karte ändert, wenn die Checkbox aktiviert ist.



Abbildung 3.25: Interaktions-Layer mit Start-, Endknoten und Pfad [Eigene Darstellung]

Mögliche Start und Endknoten anzeigen Der Button Select Node to show possible routing nodes ermöglicht es über die Karte einen Knoten auszuwählen und darauf basierend die möglichen Start- und Endknoten im kontrahierten Graphen anzuzeigen. In Abbildung 3.26 ist die Auswahl eines Knotens und die Anzeige der möglichen Start- und Endknoten dargestellt.



(a) Knoten auf einer ungerichteten Kante



(b) Knoten auf einer gerichtet Kante

Abbildung 3.26: Auswahl eines Knotens und Anzeige der möglichen Start- und Endknoten [Eigene Darstellung]

Knoten des gefundenen Pfades anzeigen Wenn bereits ein Pfad berechnet wurde, kann über den Button *Show Nodes of Path* die Knoten des gefundenen Pfades auf der Karte angezeigt werden. Dies ermöglicht es, den Pfad auf der Karte zu visualisieren und die einzelnen Knoten des Pfades zu betrachten. Dabei werden die Knoten des Pfades entsprechend ihrer Reihenfolge nach aufsteigend benannt. Ein Beispiel ist in Abbildung 3.27 dargestellt.

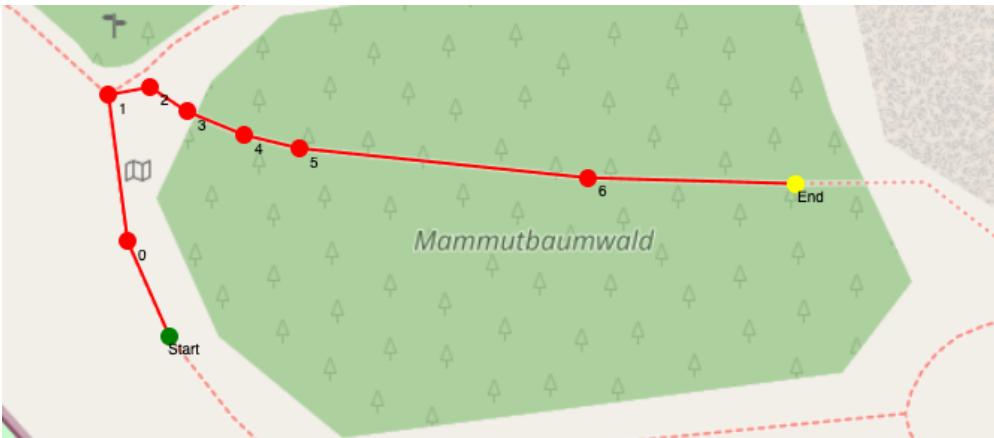


Abbildung 3.27: Knoten des gefundenen Pfades auf der Karte [Eigene Darstellung]

Laden eigener Daten Über den Datei-Input können eigene JSON-Dateien geladen werden, welche dann mit dem Knopf *Load Custom Data* in die Karte geladen werden. Dabei wird die JSON-Datei eingelesen und die darin enthaltenen Daten auf der Karte dargestellt. Dies ermöglicht es, eigene Daten in die Karte zu laden und mit dem implementierten Algorithmus zu interagieren. Ebenfalls wurde ein vollständiges Fehlerhandling integriert, wodurch sicher gestellt wird, dass nur valide JSON-Dateien geladen und verarbeitet werden. Sollte die geforderte Datenstruktur nicht gegeben sein, so wird eine entsprechende Fehlermeldung ausgegeben und die Daten nicht geladen. Ebenfalls wird eine Fehlermeldung ausgegeben, wenn die Datei nicht gelesen werden kann. Wurde die Datei erfolgreich geladen, so wird eine entsprechende Benachrichtigung ausgegeben und die neuen Daten auf der Karte dargestellt. Die Karte zentriert sich ebenfalls automatisch um die neuen Daten, sodass diese direkt sichtbar sind.

3.4.2.5 Simulation

Über das Subpanel *Simulation Controls* können die Algorithmen simuliert werden. Dabei wird die Routenberechnung schrittweise durchgeführt und die einzelnen Schritte auf der Karte dargestellt. Die Simulation ermöglicht es, die Funktionsweise der Algorithmen zu verstehen und die Berechnung der Route nachzuvollziehen. Dabei können die Simulationen gestartet, pausiert und angehalten werden. Ebenfalls kann die Geschwindigkeit der Simulation angepasst werden, um die Simulation zu verlangsamen oder zu beschleunigen. In Abbildung 3.28 sind die verschiedenen Steuerelemente der Simulation dargestellt.

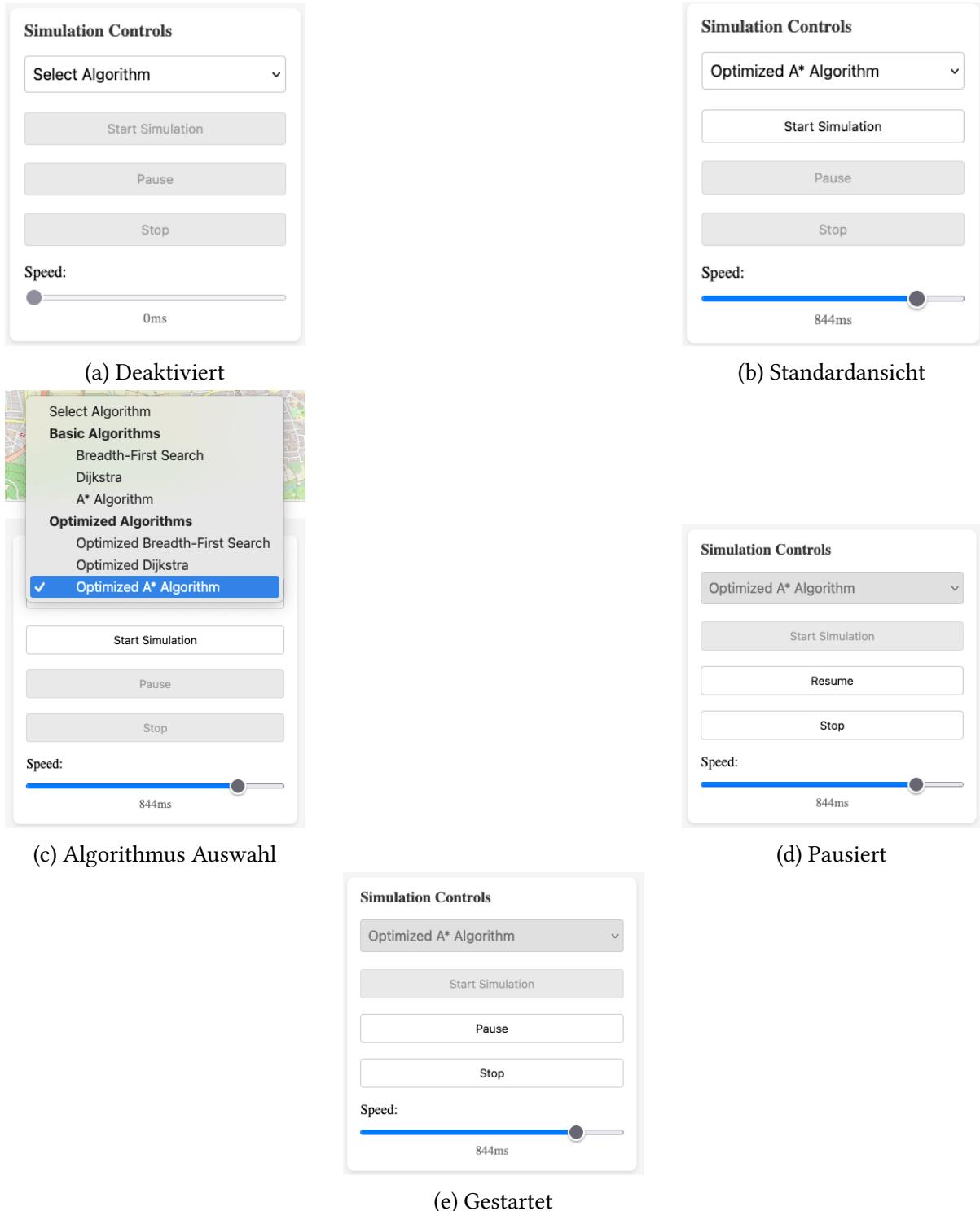


Abbildung 3.28: Simulations Steuerelemente in der Weboberfläche [Eigene Darstellung]

Wie in Abbildung 3.28a zu sehen, werden die Simulationselemente deaktiviert, sollte kein Start- und Endknoten ausgewählt sein. Sobald jedoch Start- und Endknoten ausgewählt sind, werden die Simulationselemente aktiviert und die Simulation kann gestartet werden. In Abbildung 3.28b ist die Standardansicht der Simulationselemente dargestellt. Über das Dropdown-Menü kann der gewünschte Algorithmus für die Simulation ausgewählt werden. In Abbildung 3.28c ist die Auswahl des Algorithmus dargestellt. Durch klicken auf den Button Start wird die Simulation gestartet und die Routenberechnung schrittweise durchgeführt. In Abbildung 3.28e

ist die Simulation gestartet. Über den Button Pause kann die Simulation pausiert und über den Button Stop angehalten werden. In Abbildung 3.28d ist die Simulation pausiert.

Simulationsbeispiel In diesem Beispiel wird die Simulation eines Pfades mit dem Grund- und Optimierten-A*-Algorithmus aufgezeigt. Dabei wird jeweils die Darstellung eines Zwischenstandes und der Endzustand nach erfolgreichem Finden des Pfades erläutert.

Grund-A*-Algorithmus In Abbildung 3.29 ist der Zwischenstand der Simulation mit dem Grund-A*-Algorithmus dargestellt. Hierbei stellen die dunkel blauen Kanten den gesamten Graphen dar. Türkisfarbene Knoten und Kanten wurden bereits besucht. Der aktuelle Knoten ist in orange dargestellt, der Startknoten in grün und der Endknoten in gelb. Weiterhin ist der aktuelle Pfad ebenfalls in orange dargestellt. Von dem aktuellen Knoten aus werden die nächsten Knoten betrachtet, diese nächsten Knoten werden jeweils der Reihe nach besucht und in lila dargestellt. Ebenfalls ist die verbindende Kante zu dem nächsten Knoten in lila eingefärbt. Dieser Vorgang wird solange wiederholt, bis der Endknoten erreicht wurde.

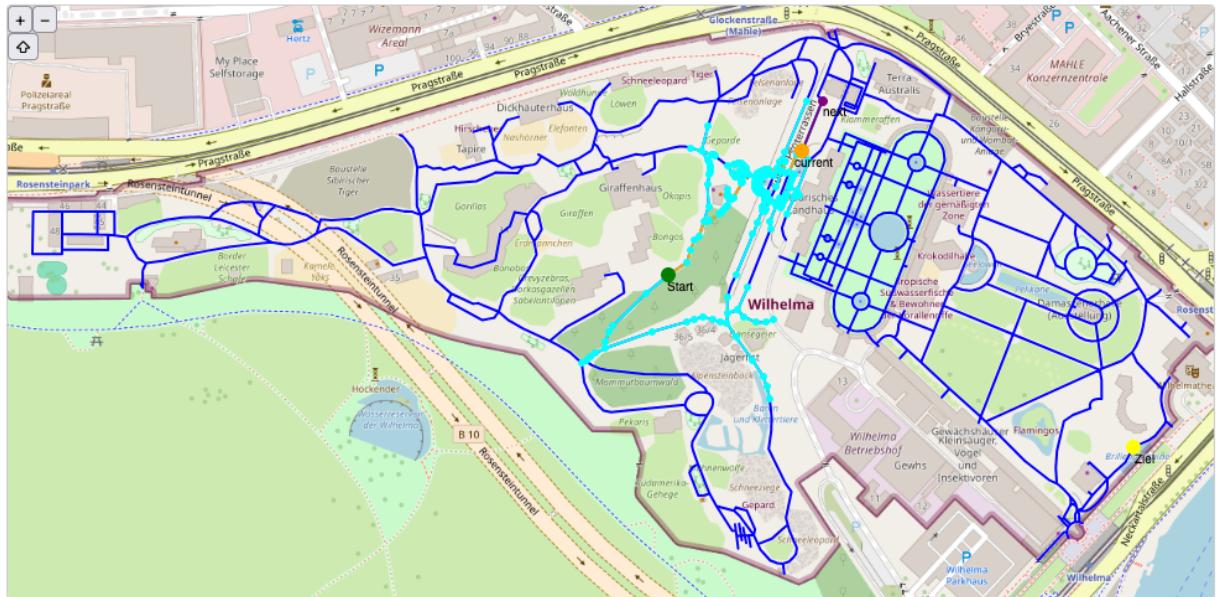


Abbildung 3.29: Zwischenstand der Simulation mit dem Optimierten-A*-Algorithmus [Eigene Darstellung]

In Abbildung 3.30 ist der Endstand der Simulation mit dem Grund-A*-Algorithmus dargestellt. Hierbei ist der gefundene Pfad in rot dargestellt. Der Pfad wurde erfolgreich gefunden und die Simulation wurde beendet. Alle türkisfarbenen Knoten und Knoten wurden dabei in Betracht gezogen, um den kürzesten Pfad zu finden.

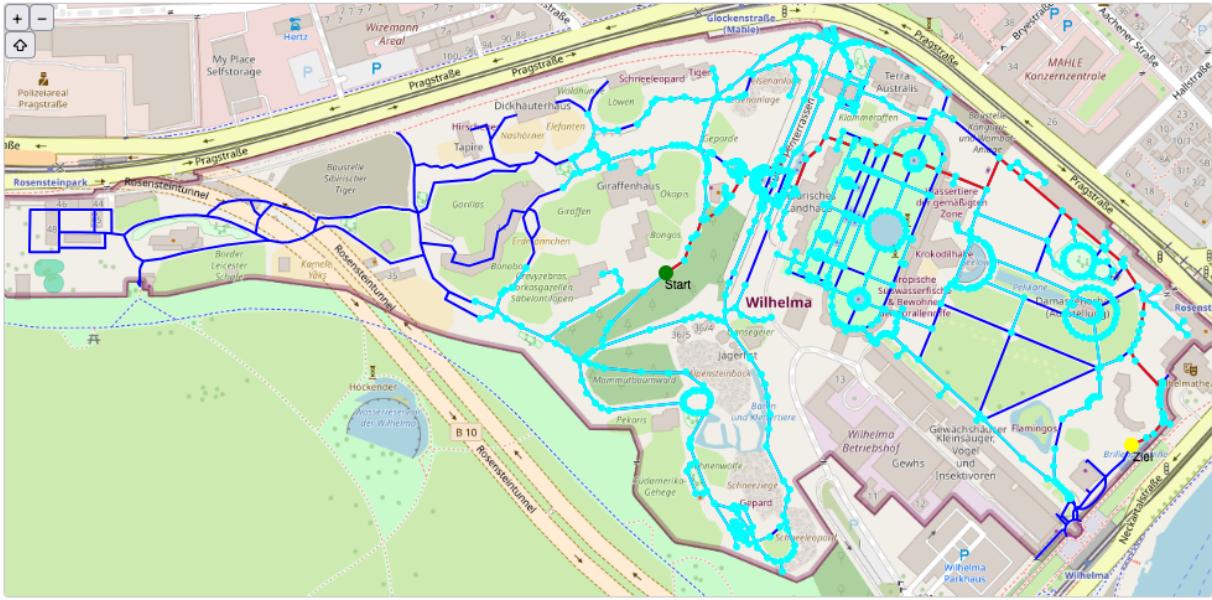


Abbildung 3.30: Endstand der Simulation mit dem Optimierten-A*-Algorithmus [Eigene Darstellung]

Optimierter-A*-Algorithmus In Abbildung 3.31 ist der Zwischenstand der Simulation mit dem Optimierten-A*-Algorithmus dargestellt. Die Farben stellen dabei die gleichen Informationen wie in Abbildung 3.29 dar. Der Unterschied liegt jedoch in der Erweiterung von Informationen, welche zuvor nicht dargestellt wurden. Dabei wird der Weg von dem ausgewählten Startknoten zu dem Startknoten in dem kontrahierten Graphen ebenfalls in grün dargestellt. Weiterhin wird der Weg von dem Endknoten in dem kontrahierten Graphen zu dem ausgewählten Endknoten in gelb dargestellt. Dies ermöglicht es, besser zu verstehen, wie die Routenberechnung über den optimierten Algorithmus abläuft.

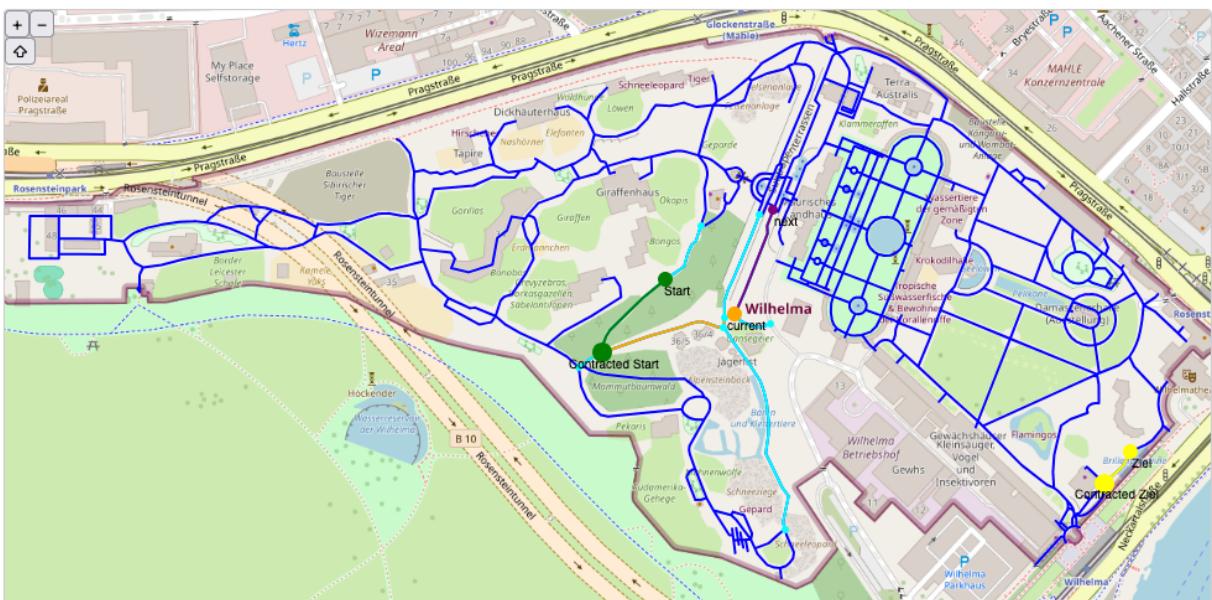


Abbildung 3.31: Zwischenstand der Simulation mit dem Optimierten-A*-Algorithmus [Eigene Darstellung]

In Abbildung 3.32 ist der Endstand der Simulation mit dem Optimierten-A*-Algorithmus dargestellt. Hierbei unterscheidet sich Darstellung in ihrer Farbgebung nicht von der des in Abbildung 3.30 gezeigten Grund-A*-Algorithmus. Das besondere an dem Endzustand der Simulation von den optimierten Algorithmen ist jedoch, dass alle betrachteten Kanten und Knoten aus jedem Durchlauf in türkisfarben dargestellt werden. So ist auch in diesem Endzustand ersichtlich, welche Kanten und Knoten in Summe durch den Algorithmus berücksichtigt wurden, um den kürzesten Pfad zu finden.

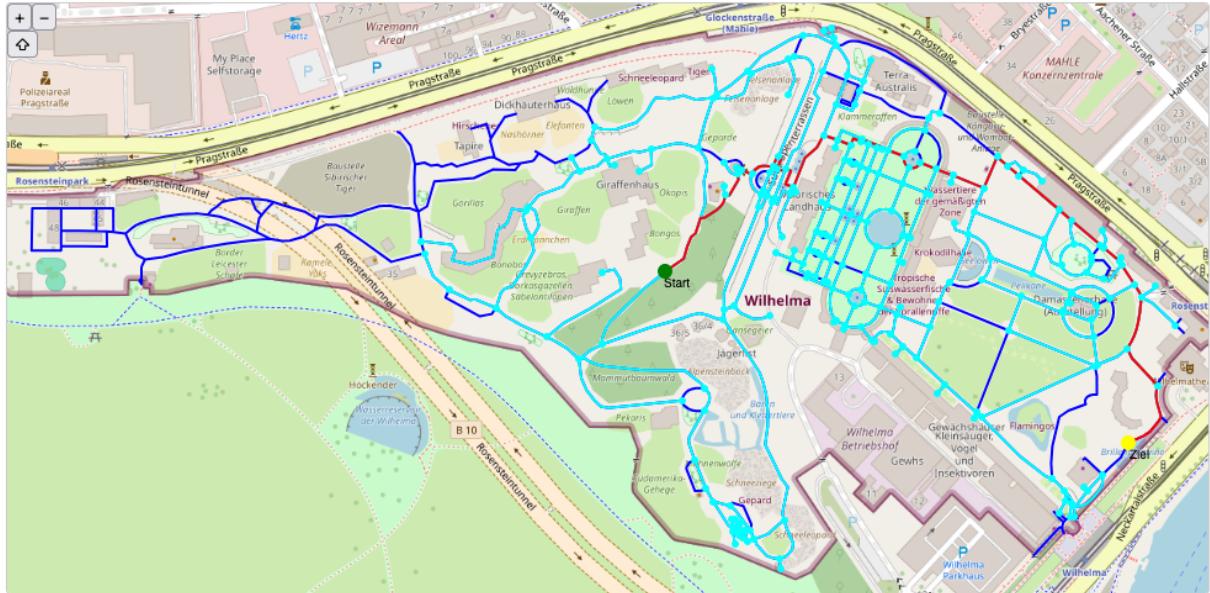


Abbildung 3.32: Endstand der Simulation mit dem Optimierten-A*-Algorithmus [Eigene Darstellung]

Generatoren Die Simulation der Routenberechnung erfolgt über die Verwendung von Generatoren. Diese erlauben es, die Routenberechnung schrittweise durchzuführen und die einzelnen Schritte auf der Karte darzustellen. Dabei wird die Berechnung der Route in einzelne Schritte unterteilt und die Karte nach jedem Schritt aktualisiert. Durch die Verwendung von Generatoren können Zwischenergebnisse ausgegeben werden, ohne die Berechnung zu unterbrechen. Dabei wird anstelle von `return` das Schlüsselwort `yield` verwendet, um den aktuellen Wert auszugeben. Der Generator wird dabei durch das Schlüsselwort `function*` definiert. Durch das Schlüsselwort `next` wird der nächste Wert des Generators abgerufen und die Berechnung fortgesetzt. [31]

3.4.2.6 Benachrichtigungen

Zur transparenten Kommunikation mit dem Benutzer werden Benachrichtigungen verwendet. Diese informieren den Benutzer über relevante Informationen zu Ihren getätigten Aktionen und geben beispielsweise Feedback über den Erfolg einer Routenberechnung. Dabei werden verschiedene Benachrichtigungsarten verwendet, welche in Abbildung 3.33 dargestellt sind.

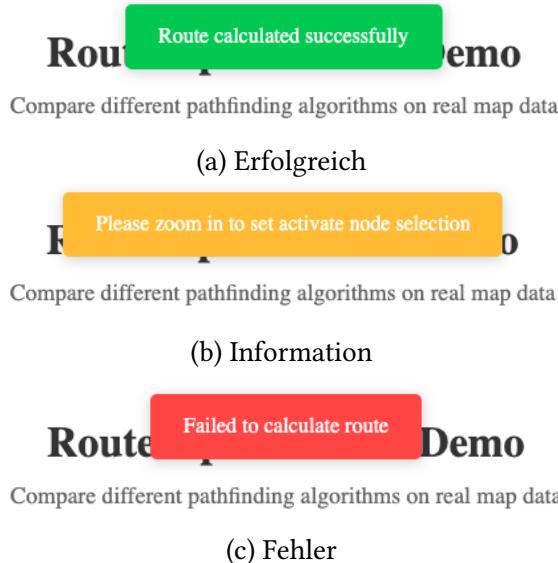


Abbildung 3.33: Verschiedene Benachrichtigungen in der Weboberfläche [Eigene Darstellung]

4 Evaluation

In diesem Kapitel wird aufgezeigt, wie die Implementierung der Optimierungen getestet und welche Ergebnisse dabei erzielt wurden. Es wird auch auf Limitationen und Probleme eingegangen, die während der Evaluation aufgetreten sind.

4.1 Vorgehensweise

Für das Testen der Implementierung wurden verschiedene Testdaten verwendet (siehe Abschnitt 3.1). Diese Testdaten repräsentieren verschiedenen Szenarien, um die Auswirkungen der Optimierungen in diesen unterschiedlichen Szenarien zu testen. Zu den Testdaten gehören kleine veranschaulichende Datensätze, aber auch städtische und ländliche Gebiete, Autobahnen und ganze Bundesländer. Dabei existieren die folgenden Datensätze, die für die Evaluation verwendet wurden:

- **Wilhema (Stuttgart):** Ein kleiner Datensatz, der das Wegenetz im Wilhelma-Gelände in Stuttgart repräsentiert. Hierbei handelt es sich um den Datensatz, der auch in Abschnitt 3.4 zu sehen ist.
- **Stuttgart:** Ein Datensatz, der das Wege- und Straßennetz in Stuttgart abbildet. Dieser Datensatz wurde verwendet, um die Optimierungen in einem städtischen Gebiet zu testen.
- **Backnang:** Dieser Datensatz bildet das Wege- und Straßennetz um die Stadt Backnang ab. Er wurde verwendet, um die Optimierungen in einem ländlichen Gebiet zu testen.
- **Autobahnen:** Es wurden zwei Datensätze verwendet, die Autobahnen in Deutschland abbilden. Dabei handelt es sich um folgende Datensätze:
 - **Deutschland:** Ein Datensatz, der ausschließlich Autobahnen mit Auf- und Abfahrten in Deutschland erfasst.
 - **Sachsen:** Ein Datensatz, der Autobahnen mit Auf- und Abfahrten in Sachsen erfasst. Zusätzlich enthält er alle Wege innerhalb eines 200-Meter-Radius um diese Zufahrten, um ein Umkehren über andere Straßentypen zu ermöglichen.
- **Baden-Württemberg:** Ein Datensatz, der das gesamte Straßennetz in Baden-Württemberg abbildet. Dieser Datensatz dient zum testen eines gemischten Szenarios, das sowohl städtische als auch ländliche Gebiete umfasst. Dieser Datensatz kommt einer realen Anwendung am nächsten.

Jeder Datensatz wurde mit allen Algorithmen getestet. Dabei wurden 1000 zufälligen Start- und Endpunkten ausgewählt, zu welchen die Algorithmen ihre jeweilige Lösung berechneten.

4.1.1 Vergleichsmetriken

Um die Ergebnisse der Algorithmen zu vergleichen, wurden die folgenden Metriken erhoben:

- **Erfolgsrate:** Die Erfolgsrate gibt an, wie viele der 1000 zufälligen Start- und Endpunkte von einem Algorithmus erfolgreich berechnet wurden. Ein Start- und Endpunkt wird als erfolgreich berechnet angesehen, wenn eine Route zwischen den beiden Punkten gefunden wurde.

- **Erfolgreiche Berechnungen**

- **Distanz:** Die durchschnittliche Distanz der berechneten Route in Metern.
 - **Zeit:** Die durchschnittliche Zeit, die benötigt wird, um die Route zu berechnen.
 - **Anzahl der Knoten:** Die durchschnittliche Anzahl der Knoten, die in der berechneten Route enthalten sind.

- **Gescheiterte Berechnungen**

- **Zeit:** Die durchschnittliche Zeit, die benötigt wird, um festzustellen, dass keine Route zwischen den beiden Punkten gefunden werden kann.

- **Gesamtdurchschnitt**

- **Zeit:** Die durchschnittliche Zeit, die benötigt wird, um eine Route zu berechnen oder festzustellen, dass keine Route gefunden werden kann.

Primärer Fokus liegt dabei auf der Berechnung erfolgreicher Wege. Die Metriken für gescheiterte Berechnungen sind dabei als zusätzliche Information zu verstehen und dienen dazu, die Auswirkungen der Optimierungen auf die Laufzeit zu verdeutlichen, um die bestimmen, dass keine Route gefunden werden kann.

4.2 Ergebnisse

In diesem Abschnitt werden die Ergebnisse der Evaluation aufgezeigt. Dabei wird auf die verschiedenen Datensätze eingegangen und die Ergebnisse der Optimierungen in diesen Szenarien dargestellt.

4.2.1 Wilhelma - Stuttgart

Der Wilhelma-Datensatz besteht dabei aus insgesamt 1099 Knoten im vollständigen Graphen und 292 Knoten im kontrahierten Graphen. Das entspricht einer Reduktion um 73,4%. Die Erfolgsrate der Berechnungen beträgt 99,6%. Dies ist beispielsweise darauf zurückzuführen, dass es gerichtete Wege gibt, welche aus der Wilhelma herausführen (Ausgänge). Sollte auf solch einem Weg gestartet werden, so ist keine Route möglich, da der Datensatz an der Stelle keine weiteren Knoten enthält. Die Ergebnisse der 1000 Testdurchläufe sind in Tabelle 4.1 dargestellt.

Die Laufzeit aller Algorithmen ist sowohl auf dem vollständigen als auch auf dem kontrahierten Graphen äußerst gering und liegt durchweg unter einer Millisekunde. Dennoch zeigt sich, dass die optimierten Algorithmen auf dem kontrahierten Graphen schneller sind als die Basisalgorithmen auf dem vollständigen Graphen. Dies wird durch die negativen Differenzwerte verdeutlicht. Die Laufzeit hat sich dabei bei der Breitensuche um 0,08ms, bei Dijkstra um 0,12ms und bei A* um 0,06ms verbessert. Die Verbesserung der Laufzeit beträgt bei der Breitensuche 12,1%, bei Dijkstra 16,2% und bei A* 10,3%. Bei diesem Datensatz erfährt der Dijkstra-Algorithmus die größte Verbesserung und A* die geringste. Dies ist darauf zurückzuführen, dass der Dijkstra-Algorithmus auf dem vollständigen Graphen die meisten Knoten besucht, um den kürzesten Pfad zu finden.

Graph Typ	Erfolgreiche Pfade			Gescheitert	Gesamt
	Laufzeit [ms]	Länge [m]	Knoten	Laufzeit [ms]	Laufzeit [ms]
vollständig	breadth	0,66	-	39	0,00
	dijkstra	0,74	415,41	53	0,00
	aStar	0,58	415,41	53	0,00
kontrahiert	breadth	0,59	-	47	0,01
	dijkstra	0,62	415,41	53	0,00
	aStar	0,52	415,41	53	0,00
Differenz	breadth	-0,08	-	8	+0,00
	dijkstra	-0,12	+0,00	0	+0,00
	aStar	-0,06	+0,00	0	+0,00

Tabelle 4.1: Leistungsvergleich verschiedener Algorithmen auf vollständigen und kontrahierten Graphen des Wilhelma-Datensatzes

Durch die Kontraktion des Graphen werden viele dieser Knoten entfernt, wodurch der DijkstrAlgorithmus auf dem kontrahierten Graphen schneller ist. Der A*-Algorithmus hingegen besucht bereits im gesamten Graphen deutlich weniger Knoten, wodurch sich die Verbesserung auf dem kontrahierten Graphen geringer auswirkt.

Zudem fällt auf, dass die Anzahl der Knoten im ermittelten Pfad bei allen Algorithmen identisch ist – mit Ausnahme der Breitensuche. Diese enthält die wenigsten Knoten, da sie den kürzesten Pfad anhand der Knotenzahl bestimmt, nicht jedoch den schnellsten. Interessanterweise ist die Anzahl der Knoten im Pfad bei der Breitensuche auf dem kontrahierten Graphen höher als auf dem vollständigen Graphen. Der Grund hierfür liegt darin, dass die Breitensuche auf dem kontrahierten Graphen nicht zwingend den kürzesten Pfad findet, sondern den Pfad mit den wenigsten Verzweigungen. Dadurch enthält der Gesamtpfad auf dem kontrahierten Graphen mehr Knoten als auf dem vollständigen Graphen.

4.2.2 Städtische Gebiete - Stuttgart

Der Stuttgart-Datensatz besteht aus insgesamt 416759 Knoten im vollständigen Graphen und 116733 Knoten im kontrahierten Graphen. Das entspricht einer Reduktion um 71,9%. Die Erfolgsrate der Berechnungen beträgt 98,2%. Dies ist auf Randstellen des Datensatzes zurückzuführen, ähnlich wie dem Wilhelma-Datensatz. Hierbei kann es auch vorkommen, dass Straßenteile im Datensatz enthalten sind, die nicht an das restliche Straßennetz angeschlossen sind. Dies tritt ebenfalls in dem Randbereich des Datensatzes auf. Von diesen Straßenteilen aus, ist entsprechend keine Route auffindbar. Wie zu sehen ist, sind diese Fälle jedoch sehr unwahrscheinlich, da zu 97,6% eine Route gefunden werden kann. Die Ergebnisse der 1000 Testdurchläufe sind in Tabelle 4.2 dargestellt.

Wie aus den Daten hervorgeht, ist die Laufzeit der Algorithmen bereits deutlich höher als bei dem Wilhelma-Datensatz. Dies ist auf die Größe des Datensatzes zurückzuführen. Die Laufzeit der Algorithmen auf dem kontrahierten Graphen ist jedoch durchweg geringer als auf dem vollständigen Graphen. Dies zeigt, dass die Optimierungen auch bei größeren Datensätzen eine Verbesserung der Laufzeit bewirken. Die Laufzeit hat sich dabei bei der Breitensuche um 112,50ms, bei Dijkstra um 118,58ms und bei A* um 16,17ms verbessert. Die Verbesserung der Laufzeit beträgt

Graph Typ	Erfolgreiche Pfade			Gescheitert	Gesamt
	Laufzeit [ms]	Länge [km]	Knoten	Laufzeit [ms]	Laufzeit [ms]
vollständig	breadth	490,40	-	362	542,30
	dijkstra	570,83	~11,88	546	683,54
	aStar	157,89	~11,88	546	907,88
kontrahiert	breadth	377,90	-	620	448,07
	dijkstra	452,25	~11,88	546	504,75
	aStar	141,72	~11,88	546	795,45
Differenz	breadth	-112,50	-	258	-94,23
	dijkstra	-118,58	+0,00	0	-178,79
	aStar	-16,17	+0,00	0	-112,43

Tabelle 4.2: Leistungsvergleich verschiedener Algorithmen auf vollständigen und kontrahierten Graphen des Stuttgart-Datensatzes

bei der Breitensuche 23,0%, bei Dijkstra 20,8% und bei A* 10,2%. Bei diesem Datensatz erfährt die Breitensuche die größte Verbesserung und A* die geringste. Weiterhin ist das gleiche Verhalten der Breitensuche wie schon bei dem Wilhelma-Datensatz zu beobachten. Die Anzahl der Knoten im Pfad auf dem kontrahierten Graphen ist höher als auf dem vollständigen Graphen.

4.2.3 Autobahnen

In diesem Abschnitt werden nachfolgend die Ergebnisse für die beiden Autobahndatensätze dargestellt.

4.2.3.1 Deutschland

Der Autobahn-Deutschland-Datensatz besteht aus insgesamt 574793 Knoten im vollständigen Graphen und 38537 Knoten im kontrahierten Graphen. Das entspricht einer Reduktion um 93,3%. Dieses hohe Reduktionspotential entsteht durch die Eigenschaften von Autobahnen. Sie weisen wenige Verzweigungen auf und können zwischen diesen Verzweigungen lange und kurvige Strecken zurücklegen. Diese werden durch die Reduktion des Graphen stark vereinfacht. Die Erfolgsrate der Berechnungen beträgt jedoch lediglich 67,8%. Diese niedrige Erfolgsrate ist dabei auf mehrere Gründe zurückzuführen. So kann beispielsweise ein Startknoten direkt vor einem Ende des Datensatzes liegen, wodurch keine Route gefunden werden kann. Oder der Endknoten liegt auf einem Stück, welches nur von außerhalb des Datensatzes erreicht werden kann. Ebenfalls ist durch die Verwendung von ausschließlich Autobahndaten, ohne die Möglichkeit über andere Straßenarten zu wenden, die Anzahl der möglichen Routen stark eingeschränkt. Dies hat den Hintergrund, dass Autobahnen in der Regel keine direkten Wendemöglichkeiten außer an größeren Autobahnkreuzen. Dennoch existieren viele Fälle, in denen kein Wenden möglich ist. Die Ergebnisse der 1000 Testdurchläufe sind in Tabelle 4.3 dargestellt.

Wie aus den Daten zu entnehmen ist, hat sich die Laufzeit bei der Breitensuche um 417,75ms, bei Dijkstra um 477,95ms und bei A* um 157,21ms verbessert. Die Verbesserung der Laufzeit beträgt bei der Breitensuche 94,6%, bei Dijkstra 94,7% und bei A* 93,9%. Die Verbesserung der Laufzeit ist bei diesem Datensatz am größten. Dies ist darauf zurückzuführen, dass der Datensatz

Graph Typ	Erfolgreiche Pfade			Gescheitert	Gesamt
	Laufzeit [ms]	Länge [km]	Knoten	Laufzeit [ms]	Laufzeit [ms]
vollständig	breadth	441,43	-	4940	396,65
	dijkstra	503,90	~440,62	5086	448,53
	aStar	167,72	~440,62	5086	518,20
kontrahiert	breadth	23,68	-	5114	19,63
	dijkstra	25,95	~440,62	5086	22,24
	aStar	10,51	~440,62	5086	28,03
Differenz	breadth	-417,75	-	174	-377,02
	dijkstra	-477,95	+0,00	0	-426,28
	aStar	-157,21	+0,00	0	-490,17

Tabelle 4.3: Leistungsvergleich verschiedener Algorithmen auf vollständigen und kontrahierten Graphen des Autobahn-Deutschland-Datensatzes

ausschließlich Autobahnen enthält, die nur wenige Verzweigungen aufweisen. Dadurch kann der Graph stark kontrahiert werden, was zu einer großen Verbesserung der Laufzeit führt.

4.2.3.2 Sachsen

Der Autobahn-Sachsen-Datensatz besteht aus insgesamt 29542 Knoten im vollständigen Graphen und 1955 Knoten im kontrahierten Graphen. Das entspricht einer Reduktion um 93,4%. Dieses hohe Reduktionspotential hat denselben Hintergrund, wie beim Autobahn-Deutschland-Datensatz. Autobahnen weisen wenige Verzweigungen auf und können zwischen diesen Verzweigungen lange und kurvige Strecken zurücklegen. Die Erfolgsrate der Berechnungen ist mit 80,5% höher als beim Autobahn-Deutschland-Datensatz, jedoch immer noch niedrig. Die Verbesserung gegen über dem Autobahn-Deutschland-Datensatz ist durch die Erweiterung um Wendemöglichkeiten über andere Straßentypen zu erklären. Allerdings sind durch die Verwendung von *around* bei der Abfrage der Daten, weitere Straßen enthalten, die nicht an das restliche Straßennetz angeschlossen sind. Von diesen Straßenteilen aus, ist entsprechend keine Route auffindbar, was dazu führt, dass die Erfolgsrate nicht noch besser ist. Die Ergebnisse der 1000 Testdurchläufe sind in Tabelle 4.4 dargestellt.

Auch bei dem Autobahn-Sachsen-Datensatz ist die Laufzeit der Algorithmen auf dem kontrahierten Graphen durchweg geringer als auf dem vollständigen Graphen. Die Laufzeit hat sich dabei bei der Breitensuche um 16,22ms, bei Dijkstra um 17,91ms und bei A* um 8,24ms verbessert. Die Verbesserung der Laufzeit beträgt bei der Breitensuche 91,5%, bei Dijkstra 92,7% und bei A* 90,6%. Die Verbesserung der Laufzeit ist bei diesem Datensatz knapp unterhalb des Autobahn-Deutschland-Datensatzes. Dies ist darauf zurückzuführen, dass der Datensatz ebenfalls Autobahndaten beinhaltet, allerdings noch weitere Wege innerhalb eines 200-Meter-Radius um die Zufahrten herum. Dadurch kann der Graph stark kontrahiert werden, was zu einer deutlichen Verbesserung der Laufzeit führt, allerdings nicht so stark, wie bei dem reinen Autobahn-Datensatz.

Graph Typ	Erfolgreiche Pfade			Gescheitert	Gesamt
	Laufzeit [ms]	Länge [km]	Knoten	Laufzeit [ms]	Laufzeit [ms]
vollständig	breadth	17,70	-	1218	18,11
	dijkstra	19,35	~89,24	1228	20,58
	aStar	9,10	~89,24	1228	23,73
kontrahiert	breadth	1,48	-	1222	1,55
	dijkstra	1,44	~89,24	1228	1,57
	aStar	0,86	~89,24	1228	2,09
Differenz	breadth	-16,22	-	4	-16,56
	dijkstra	-17,91	+0,00	0	-19,01
	aStar	-8,24	+0,00	0	-21,64

Tabelle 4.4: Leistungsvergleich verschiedener Algorithmen auf vollständigen und kontrahierten Graphen des Autobahn-Sachsen-Datensatzes

4.2.4 Ländliche Gebiete - Backnang

Der Backnang-Datensatz besteht aus insgesamt 113525 Knoten im vollständigen Graphen und 23494 Knoten im kontrahierten Graphen. Das entspricht einer Reduktion um 79,3%. Die Erfolgsrate der Berechnungen beträgt dabei 98,4%. Die Ergebnisse der 1000 Testdurchläufe sind in Tabelle 4.4 dargestellt.

Graph Typ	Erfolgreiche Pfade			Gescheitert	Gesamt
	Laufzeit [ms]	Länge [km]	Knoten	Laufzeit [ms]	Laufzeit [ms]
vollständig	breadth	117,79	-	332	86,28
	dijkstra	137,38	~10,31	429	101,14
	aStar	51,26	~10,31	429	144,26
kontrahiert	breadth	74,49	-	472	58,75
	dijkstra	87,73	~10,31	429	72,70
	aStar	36,98	~10,31	429	121,55
Differenz	breadth	-43,30	-	140	-27,54
	dijkstra	-49,65	+0,00	0	-28,44
	aStar	-14,28	+0,00	0	-22,72

Tabelle 4.5: Leistungsvergleich verschiedener Algorithmen auf vollständigen und kontrahierten Graphen des Backnang-Datensatzes

Die Laufzeit der Algorithmen auf dem kontrahierten Graphen ist durchweg geringer als auf dem vollständigen Graphen. Die Laufzeit hat sich dabei bei der Breitensuche um 43,30ms, bei Dijkstra um 49,65ms und bei A* um 14,28ms verbessert. Die Verbesserung der Laufzeit beträgt bei der Breitensuche 36,7%, bei Dijkstra 36,1% und bei A* 27,9%. Die Verbesserung der Laufzeit ist bei diesem Datensatz deutlich geringer als bei den Autobahndaten, jedoch ebenfalls besser als bei dem städtischen Beispiel. Dies ist darauf zurückzuführen, dass der Datensatz zwar mehr Verzweigungen und kürzere Wege enthält, als die Autobahndaten, allerdings deutlich weitläufiger ist als die städtischen Daten. Dadurch enthält der kontrahierte Graph weniger Knoten, was zu einer Verbesserung der Laufzeit führt.

4.2.5 Bundesländerweit - Baden-Württemberg

Der Baden-Württemberg-Datensatz besteht aus insgesamt 2703440 Knoten im vollständigen Graphen und 377984 Knoten im kontrahierten Graphen. Das entspricht einer Reduktion um 86,0%. Die Erfolgsrate der Berechnungen beträgt dabei 95,3%. Dies lässt sich ebenfalls durch Probleme im Randbereich des Datensatzes erklären. Die Ergebnisse der 1000 Testdurchläufe sind in Tabelle 4.6 dargestellt.

Graph Typ	Erfolgreiche Pfade			Gescheitert	Gesamt
	Laufzeit [s]	Länge [km]	Knoten	Laufzeit [s]	Laufzeit [s]
vollständig	breadth	~5,15	-	2764	~3,66
	dijkstra	~6,08	~124,10	3826	~4,54
	aStar	~2,33	~124,10	3826	~7,20
kontrahiert	breadth	~3,01	-	3355	~1,42
	dijkstra	~2,17	~124,10	3826	~1,81
	aStar	~1,15	~124,10	3826	~3,51
Differenz	breadth	-2,14	-	590	-2,24
	dijkstra	-3,91	+0,00	0	-2,73
	aStar	-1,18	+0,00	0	-3,69

Tabelle 4.6: Leistungsvergleich verschiedener Algorithmen auf vollständigen und kontrahierten Graphen des Baden-Württemberg-Datensatzes

Die Laufzeit der Algorithmen auf dem kontrahierten Graphen ist durchweg geringer als auf dem vollständigen Graphen. Die Laufzeit hat sich dabei bei der Breitensuche um 2,14s, bei Dijkstra um 3,91s und bei A* um 1,18s verbessert. Die Verbesserung der Laufzeit beträgt bei der Breitensuche 41,6%, bei Dijkstra 64,3% und bei A* 50,6%. Die Verbesserung der Laufzeit ist bei diesem Datensatz exakt in der Mitte aller anderer verglichener Datensätze. Dies ist darauf zurückzuführen, dass der Datensatz sowohl städtische als auch ländliche Gebiete umfasst. Dadurch enthält der kontrahierte Graph weniger Knoten, was zu einer Verbesserung der Laufzeit führt. Allerdings sind die Verbesserungen geringer als bei reinen Autobahndaten.

4.3 Limitationen

In diesem Abschnitt werden Limitation und Probleme während der Evaluation beschrieben. Dazu gehören Limitationen der Overpass API, sowie Limitationen bei der Erstellung des Graphen und der Berechnung der Route.

4.3.1 Overpass API Limitierung

Bei der Datenabfrage über Overpass, ist die Größe der zurückgegebenen Datenmenge limitiert. Diese Limitierung beträgt 2GB. Sollte die Datenmenge größer als 2GB sein, so wird die Anfrage abgebrochen und es werden keine Daten zurückgegeben. Dies ist insbesondere bei der Abfrage von großen Datensätzen, wie beispielsweise der Abfrage aller Straßen und Wege von Deutschland, ein Problem.

4.3.2 Erstellung des Graphen

Neben den Limitierungen der Overpass API, gibt es auch Limitierungen bei der Erstellung des Graphen. Diese hängen dabei mit dem Speicher, der einer NodeJS-Anwendung zur Verfügung steht, zusammen. Sollte der Graph zu groß werden, so kann es passieren, dass die Anwendung aufgrund von Speichermangel abstürzt. Dies ist insbesondere bei der Erstellung des Graphen für große Datensätze (> 1GB) ein Problem. Über den Parameter `-max-old-space-size=X` kann dabei der Speicher, der einer NodeJS-Anwendung zur Verfügung steht, erhöht werden. Dieser Parameter gibt an, wie viel Speicher in Megabyte der Anwendung zur Verfügung steht. Getestet wurde in diesem Zusammenhang mit 32768MB. Trotz dieser Erhöhung des Speichers, kann es passieren, dass die Anwendung abstürzt, da die Berechnungen zu viele Daten im Speicher halten.

4.4 Vergleich / Diskussion

Die Evaluation der verschiedenen Datensätze und Algorithmen liefert wertvolle Erkenntnisse über die Effektivität der implementierten Optimierungen. In diesem Abschnitt werden die Ergebnisse vergleichend analysiert und die zugrundeliegenden Faktoren diskutiert, die zu den beobachteten Leistungsunterschieden führen.

4.4.1 Einfluss der Netzwerkstruktur auf die Optimierungseffizienz

Die durchgeführten Tests zeigen deutlich, dass die Struktur des Straßennetzwerks einen erheblichen Einfluss auf das Potenzial und die Wirksamkeit der Graphenkontraktion hat. Abbildung 4.1 visualisiert die prozentuale Verbesserung der Laufzeit durch die Kontraktion für die verschiedenen Datensätze.

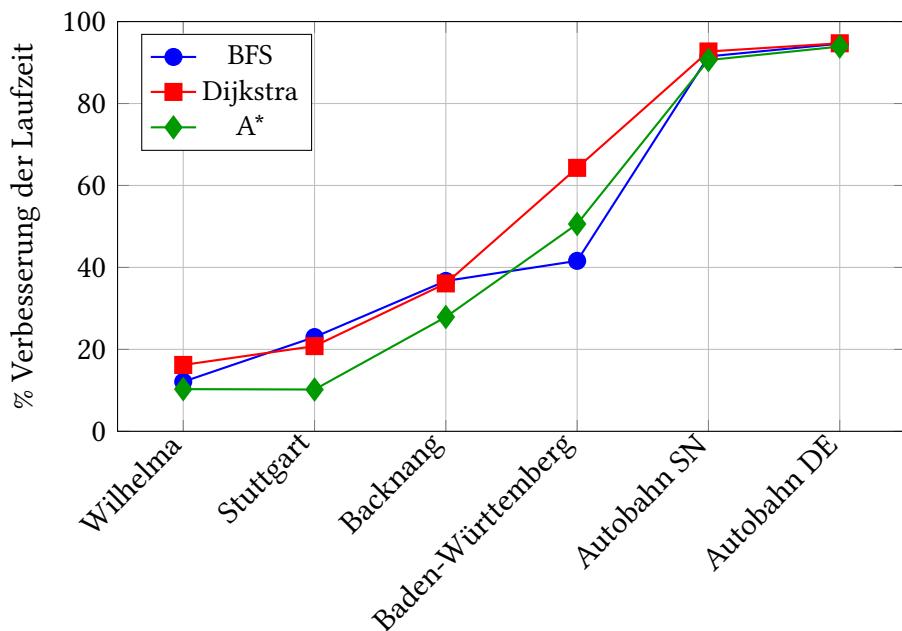


Abbildung 4.1: Prozentuale Verbesserung der Laufzeit durch Graphenkontraktion nach Datensatztyp und Algorithmus [Eigene Darstellung]

Aus den Testergebnissen lassen sich folgende Schlüsselerkenntnisse ableiten:

4.4.1.1 Autobahnnetzwerke: Maximales Optimierungspotenzial

Die mit Abstand größten Leistungsverbesserungen werden bei den Autobahndatensätzen erzielt, mit einer Laufzeitreduktion von über 90% bei allen getesteten Algorithmen. Dies ist auf mehrere strukturelle Eigenschaften von Autobahnen zurückzuführen:

- **Geringe Verzweigungsdichte:** Autobahnen weisen vergleichsweise wenige Auf- und Abfahrten (Verzweigungen) auf, wodurch lange Streckenabschnitte ohne Entscheidungspunkte entstehen.
- **Hohe Kurvigkeit:** Zwischen den Verzweigungen enthalten Autobahnen typischerweise zahlreiche Zwischenpunkte, die den geografischen Verlauf definieren, aber keine navigationstechnische Bedeutung haben.

Der Reduktionsgrad von 93,3% bzw. 93,4% bei den Autobahndatensätzen belegt die hohe Effizienz der Kontraktion für diese Netzwerktypen. Dadurch verringert sich die zu durchsuchende Knotenmenge drastisch, was direkt zu der beobachteten erheblichen Beschleunigung führt.

4.4.1.2 Städtische Gebiete: Moderates Optimierungspotenzial

Bei städtischen Datensätzen wie Stuttgart ist die erzielte Verbesserung deutlich geringer, mit Werten zwischen 10,2% und 23,0% je nach Algorithmus. Diese moderateren Gewinne erklären sich durch:

- **Hohe Verzweigungsdichte:** Städtische Straßennetze weisen eine Vielzahl von Kreuzungen, Einmündungen und komplexen Verkehrsknotenpunkten auf, die als nicht kontrahierbare Verzweigungsknoten erhalten bleiben müssen.
- **Kurze Distanzen zwischen Verzweigungen:** Die durchschnittliche Straßenlänge zwischen zwei Kreuzungen ist in städtischen Bereichen deutlich geringer, wodurch weniger Zwischenknoten kontrahiert werden können.
- **Komplexe Verkehrsregeln:** Einbahnstraßen und andere verkehrstechnische Besonderheiten führen zu einem komplexeren Graphen mit mehr nicht kontrahierbaren Elementen.

Dennoch zeigt der Reduktionsgrad von 71,9% beim Stuttgart-Datensatz, dass auch in städtischen Gebieten ein erhebliches Potenzial zur Graphenvereinfachung besteht, selbst wenn sich dies nicht im gleichen Maße in der Laufzeitverbesserung widerspiegelt.

4.4.1.3 Ländliche Gebiete: Erhöhtes Optimierungspotenzial

Die Ergebnisse für den Backnang-Datensatz zeigen mit Verbesserungen zwischen 27,9% und 36,7% ein deutlich höheres Optimierungspotenzial als bei städtischen Gebieten. Der Reduktionsgrad von 79,3% unterstreicht die Effektivität der Kontraktion in ländlichen Regionen. Dies lässt sich auf folgende Faktoren zurückführen:

- **Mittlere Verzweigungsdichte:** Ländliche Straßennetze weisen weniger Verzweigungen pro Flächeneinheit auf als städtische Gebiete.

- **Kurvige Landstraßen:** Ländliche Straßen folgen oft natürlichen Geländeformen und weisen daher viele Kurven auf, die durch zahlreiche kontrahierbare Zwischenknoten repräsentiert werden.
- **Längere Verbindungsstrecken:** Zwischen Ortschaften und Kreuzungspunkten liegen oft längere, unverzweigte Straßenabschnitte, die sich ideal für die Kontraktion eignen.

4.4.1.4 Gemischte Datensätze: Ausgewogene Optimierung

Der Baden-Württemberg-Datensatz als größter getester Datensatz mit gemischter Struktur zeigt mit Verbesserungen zwischen 41,6% und 64,3% eine ausgewogene Optimierung. Der Reduktionsgrad von 86,0% verdeutlicht das hohe Optimierungspotenzial auch bei großen, heterogenen Datensätzen. Hierbei zeigt sich ein interessantes Phänomen: Die Laufzeitverbesserung liegt deutlich über der von rein städtischen oder ländlichen Gebieten, was auf folgende Faktoren zurückzuführen sein könnte:

- **Ausgewogene Mischung:** Der Datensatz enthält sowohl städtische als auch ländliche Bereiche sowie Autobahnen, wodurch eine günstige Kombination der jeweiligen Optimierungspotenziale entsteht.
- **Skaleneffekte:** Bei größeren Datensätzen wirkt sich die Reduktion der zu durchsuchenden Knotenmenge stärker aus, da die absolute Anzahl der eingesparten Berechnungsschritte höher ist.

4.4.2 Algorithmenspezifische Unterschiede

Die Evaluation zeigt deutliche Unterschiede in der Optimierbarkeit der verschiedenen Routingalgorithmen durch die Graphenkontraktion:

4.4.2.1 Dijkstra-Algorithmus: Höchster relativer Gewinn

Der Dijkstra-Algorithmus profitiert in den meisten Fällen am stärksten von der Kontraktion, insbesondere bei großen Datensätzen. Bei Baden-Württemberg beträgt die Verbesserung stolze 64,3%. Dies ist zu erwarten, da:

- Der Algorithmus ohne Heuristik eine breitere Suche durchführt und daher stärker von der Reduzierung der zu untersuchenden Knotenmenge profitiert.
- Die Laufzeitkomplexität von Dijkstra direkt von der Anzahl der Kanten und Knoten abhängt.

4.4.2.2 Breitensuche: Konsistente Verbesserung

Die Breitensuche zeigt über alle Datensätze hinweg eine solide Verbesserung, die besonders bei den Autobahndaten mit 94,6% hervorsticht. Dies lässt sich erklären durch:

- Die Vereinfachung des Suchraums, die besonders bei der nicht-zielgerichteten Breitensuche zu Buche schlägt.
- Die Reduktion der zu traversierenden Kanten, was die Anzahl der Schritte bis zum Erreichen des Ziels signifikant verringert.

- Die Laufzeitkomplexität von $O(|V| + |E|)$, die direkt von der Reduktion der Knoten- und Kantenmenge profitiert.

Bemerkenswert ist jedoch die Beobachtung, dass die durch Breitensuche gefundenen Pfade im kontrahierten Graphen tendenziell mehr Knoten enthalten als im vollständigen Graphen. Dies ist darauf zurückzuführen, dass die Breitensuche den Pfad mit den wenigsten Knoten findet, was im kontrahierten Graphen nicht mehr zwangsläufig zum global kürzesten Weg führt.

4.4.2.3 A*-Algorithmus: Geringster relativer Gewinn

Der A*-Algorithmus zeigt in den meisten Testfällen die geringste relative Verbesserung durch die Kontraktion. Dies ist konsistent mit den theoretischen Erwartungen, da:

- A* durch seine heuristische Komponente bereits im unkontrahierten Graphen eine zielgerichtete Suche durchführt.
- Die Luftlinienheuristik bereits viele Knoten ausschließt, die nicht in Richtung des Ziels liegen, wodurch der zusätzliche Nutzen der Kontraktion geringer ausfällt.
- Der Algorithmus bereits eine optimierte Suchstrategie implementiert, wodurch der Raum für weitere Verbesserungen kleiner ist.

Dennoch sind die absoluten Verbesserungen bei A* beachtlich, insbesondere bei den Autobahndatensätzen mit über 90% und bei Baden-Württemberg mit 50,6%. Dies unterstreicht die gegenseitige Ergänzung der algorithmischen Optimierung durch A* und der strukturellen Optimierung durch Graphenkontraktion.

4.4.3 Skalierbarkeit und praktische Anwendbarkeit

Die Ergebnisse demonstrieren die Skalierbarkeit der implementierten Optimierungen für große Datensätze. Besonders deutlich wird dies beim Baden-Württemberg-Datensatz:

- Die Reduktion der durchschnittlichen Laufzeit von 5,08 auf 2,94 Sekunden bei der Breitensuche, von 6,01 auf 2,15 Sekunden bei Dijkstra und von 2,56 auf 1,26 Sekunden bei A* stellt einen signifikanten Gewinn für praktische Anwendungen dar.
- Die Verringerung der zu verarbeitenden Datenmenge von 2,7 Millionen auf 378.000 Knoten (86,0% Reduktion) ermöglicht den Einsatz auf Geräten mit begrenzten Ressourcen wie mobilen Endgeräten oder Navigationsgeräten.

Dabei ist hervorzuheben, dass die Optimierungen keine Kompromisse bei der Qualität der berechneten Routen erfordern. Die gefundenen Pfade sind identisch zu denen im unkontrahierten Graphen, wie die konsistenten Längenangaben in den Ergebnistabellen belegen. Ein Ausnahmefall ist die Breitensuche, die aufgrund ihrer spezifischen Funktionsweise im kontrahierten Graphen teilweise andere (aber nicht schlechtere) Pfade findet.

4.4.4 Charakteristische Muster der Optimierung

Aus den Evaluationsdaten lassen sich charakteristische Muster der Optimierungseffizienz in Abhängigkeit von der Straßennetzstruktur ableiten. Abbildung 4.2 visualisiert den Zusammenhang zwischen der Netzwerkstruktur und dem Optimierungspotenzial.

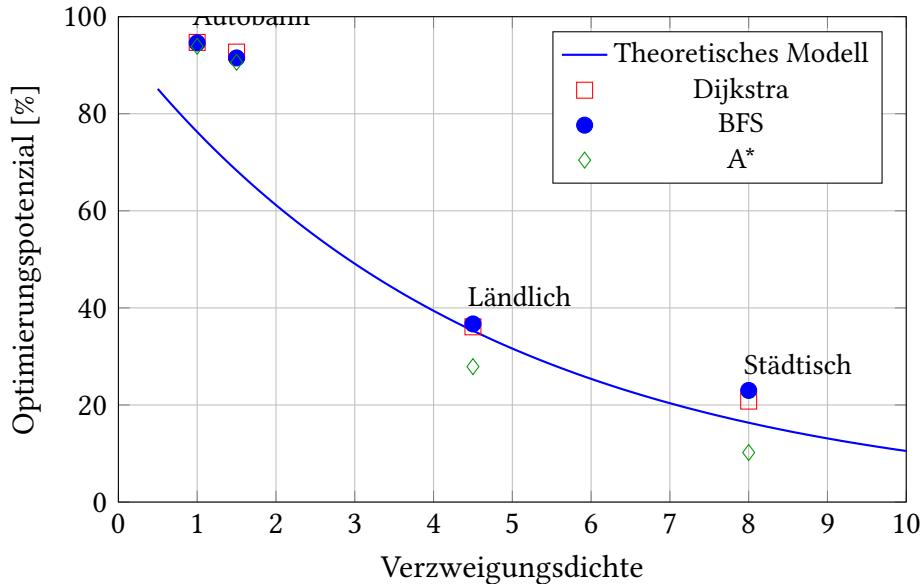


Abbildung 4.2: Zusammenhang zwischen Verzweigungsichte und Optimierungspotenzial durch Graphenkontraktion für verschiedene Algorithmen [Eigene Darstellung]

Das identifizierte Muster zeigt einen klaren Zusammenhang zwischen der Verzweigungsichte und dem Optimierungspotenzial: Je niedriger die Verzweigungsichte, desto höher die mögliche Laufzeitverbesserung durch Graphenkontraktion. Dieses Prinzip erklärt die beobachtete Reihenfolge der Optimierungseffizienz:

1. **Autobahnen** (niedrigste Verzweigungsichte): 90-95% Laufzeitverbesserung
2. **Ländliche Gebiete** (mittlere Verzweigungsichte): 27-37% Laufzeitverbesserung
3. **Städtische Gebiete** (hohe Verzweigungsichte): 10-23% Laufzeitverbesserung

Der Baden-Württemberg-Datensatz fällt aus diesem Muster etwas heraus, was auf die bereits diskutierten Skaleneffekte und die gemischte Struktur zurückzuführen ist.

4.4.5 Fazit der Evaluation

Die durchgeführte Evaluation bestätigt die Effektivität der implementierten Optimierungen und zeigt deren unterschiedliche Auswirkungen auf verschiedene Straßennetzwerktypen und Routingalgorithmen. Zusammenfassend lässt sich feststellen:

- **Graphenkontraktion ist hocheffektiv:** Die Reduktion der Knotenmenge um 72-93% führt zu Laufzeitverbesserungen von 10-95%, je nach Datensatz und Algorithmus.
- **Netzwerkstruktur ist entscheidend:** Das Optimierungspotenzial korreliert stark mit der Verzweigungsichte und Charakteristik des Straßennetzes.
- **Algorithmenspezifische Unterschiede:** Dijkstra profitiert tendenziell am stärksten, A* am wenigsten von der Kontraktion, während die Breitensuche eine Mittelposition einnimmt.
- **Skalierbarkeit:** Die Optimierungen skalieren gut mit der Datensatzgröße und zeigen bei großen, landesweiten Datensätzen starke Verbesserungen.
- **Praxistauglichkeit:** Die hohen Erfolgsraten und die Beibehaltung der Routenqualität belegen die Eignung für reale Anwendungsszenarien.

Die implementierte Lösung liefert damit einen wertvollen Beitrag zur effizienten Routenberechnung auf OpenStreetMap-Daten und demonstriert das große Potenzial struktureller Optimierungen in der Graphenverarbeitung.

5 Zusammenfassung und Ausblick

In diesem Kapitel eine Zusammenfassung der Ergebnisse und Erkenntnisse der Projektarbeit sowie ein Ausblick auf mögliche Erweiterungen und Verbesserungen der entwickelten Routinganwendung gegeben. Dabei werden verschiedene Optimierungsmöglichkeiten und weitere Funktionalitäten vorgestellt, die die Effizienz und Benutzerfreundlichkeit der Anwendung weiter steigern können.

5.1 Zusammenfassung

Im Rahmen der vorliegenden Projektarbeit wurden Routingalgorithmen auf OpenStreetMap-Daten implementiert, optimiert und visualisiert. Ein zentraler Bestandteil war die Entwicklung einer effizienten Graphenkontraktion zur Beschleunigung der Routenberechnung sowie die Erstellung einer interaktiven Webanwendung zur Veranschaulichung der Algorithmen.

Die durchgeführte Evaluation bestätigt die Effektivität der Optimierungen und zeigt, dass die Graphenkontraktion erheblich zur Beschleunigung von Routingalgorithmen beitragen kann. Laufzeitverbesserungen von bis zu 95% bei gleichbleibender Routenqualität belegen die Effizienz des gewählten Ansatzes. Zudem hebt die systematische Analyse den Zusammenhang zwischen Netzwerkstruktur und Optimierungspotenzial hervor und liefert wertvolle Erkenntnisse für die gezielte Anwendung dieser Techniken.

Die entwickelte Webanwendung macht die optimierten Routingalgorithmen nicht nur praktisch nutzbar, sondern ermöglicht durch ihre Visualisierungen auch ein besseres Verständnis der zugrunde liegenden Prozesse.

Die erzielten Ergebnisse und Methoden bieten eine fundierte Grundlage für weiterführende Forschung und Entwicklung im Bereich der effizienten Routenberechnung und sind sowohl für wissenschaftliche als auch für praxisnahe Anwendungen von Relevanz.

5.2 Optimierungspotenzial

Allerdings bleiben auch nach Abschluss der Projektarbeit noch verschiedene Optimierungsmöglichkeiten und Funktionalitäten offen, die in zukünftigen Arbeiten weiter untersucht und umgesetzt werden können.

In diesem Abschnitt werden weitere dieser Optimierungsmöglichkeiten für die entwickelte Routinganwendung vorgestellt. Diese können dazu beitragen, die Effizienz und Benutzerfreundlichkeit der Anwendung weiter zu steigern.

5.2.1 Speicheroptimierungen der Graphenkontraktion

Im Bereich des Speicherbedarfs gibt es weiteres Optimierungspotenzial. Hierbei kann der Algorithmus der Graphenkontraktion hinsichtlich des Speicherbedarfs verbessert werden. Es führt bereits bei einer Datenmenge von einem Gigabyte (1GB) zu einem Überlauf des Speichers mit einer Größe von 32GB. Dies ist problematisch, da die Erstellung eines Graphen mit einer Größe von 1GB ungefähr dem Straßennetz von Baden-Württemberg entspricht. Sollte ein noch größeres Straßennetz betrachtet werden, so ist eine Optimierung des Speicherverbrauchs zwingend erforderlich.

5.2.2 Andere Optimierungsverfahren

Ebenfalls ist es möglich andere Optimierungsverfahren zu implementieren, um die Laufzeit der Anwendung zu verbessern. Hierbei werden nachfolgend exemplarisch zwei Ansätze näher betrachtet und erläutert. Beide Ansätze versuchen dabei das Problem der häufigen Routenberechnung, wie sie in der Anwendung vorkommt, zu lösen, sodass lediglich eine Route berechnet werden muss anstelle der vier Routen, die aktuell im schlimmsten Fall berechnet werden müssen.

5.2.2.1 Abkürzungen einfügen

Ein erster Ansatz besteht darin die kontrahierten Kanten mit ihren Gewichten den entsprechenden Knoten mitanzuhängen. Dies ermöglicht es, dass die kontrahierten Kanten, also die gesamten Wegstücke bis zum nächsten Verzweigungsknoten, von den Algorithmen in der Routenberechnung direkt berücksichtigt werden können. Der Vorteil besteht darin, dass die Algorithmen nicht mehr auf einen der beiden Graphen limitiert sind, sondern die Vorteile aus beiden miteinander kombinieren, um somit die Laufzeit zu verbessern.

Ein Beispiel für eine Graphen mit eingefügten kontrahierten Kanten ist in Abbildung 5.1 dargestellt.

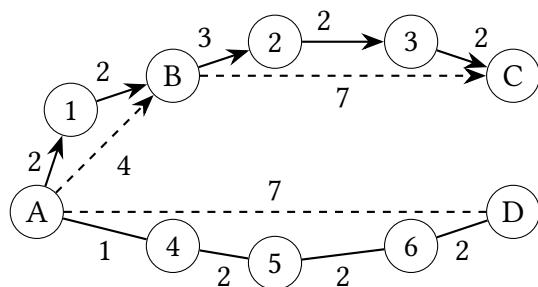


Abbildung 5.1: Beispiel für das Einfügen von Abkürzungen [Eigene Darstellung]

Die kontrahierten Kanten können dabei als Routingabkürzungen betrachtet werden, da sie die Möglichkeit bieten, direkt das gesamte Wegstück in einem einzigen Schritt zu betrachten. So könnte beispielsweise der Weg von A nach B über die direkt Kante A-B abgebildet werden, ohne den Zwischenschritt über den Knoten 1 zu gehen. Dies ermöglicht es, in größeren Schritten dem Zielknoten näher zu kommen, ohne dabei alle Knoten des Wegstückes einzeln betrachten zu müssen. Die einzelnen Kanten der Zwischenknoten werden jedoch weiterhin aufgrund der Funktionsweise der Algorithmen berücksichtigt. So ist es immer noch möglich, dass ein Zielknoten über eine nicht kontrahierte Kanten erreicht werden kann.

Dieser Ansatz ermöglicht es, die Laufzeit der Routingberechnungen zu verbessern, durch den Vorteil, dass nur eine Route berechnet werden muss. Allerdings ist es weiterhin notwendig, die kontrahierten Kanten zu erstellen, was einen zusätzlichen Rechenaufwand beim Erstellen des Graphen bedeutet. Dieser Rechenaufwand entsteht dabei einmalig.

5.2.2.2 Effizienzsteigerung durch selektive Knotenauswahl

Ein weiterer Ansatz besteht darin nur Knoten mit einem Grad ungleich zwei (2) oder den Zielknoten in die Queue der zu betrachtenden Knoten beim Routen aufzunehmen. Dies ermöglicht es, dass die Algorithmen nur noch Knoten betrachten, die entweder eine Verzweigung darstellen oder der Zielknoten sind. Dadurch wird die Anzahl der Knoten, die betrachtet werden deutlich reduziert, was dazu führt, dass die Algorithmen noch schneller das Ziel erreichen können, da weniger Knoten betrachtet werden müssen.

In Algorithmus 14 wird dieser Ansatz zur Optimierung des Hinzufügens von Knoten zur Queue dargestellt. Dabei wird zu Beginn mit einer leeren Queue gestartet und der Startknoten hinzugefügt. Anschließend wird solange ein Knoten aus der Queue genommen, bis die Queue leer ist. Dabei wird überprüft, ob der Knoten ein Zielknoten ist. Ist dies der Fall, so wird die Route zurückverfolgt und die Funktion beendet. Ansonsten werden alle ausgehenden Kanten des Knotens betrachtet. Sollte der Knoten einer ausgehenden Kante einen Grad von zwei (2) haben und nicht das Ziel sein, so wird entlang der ausgehenden Kante dieses Knotens weitergegangen, bis ein Knoten gefunden wird, der entweder einen Grad ungleich zwei (2) hat oder das Ziel ist. Anschließend wird der Knoten mit seinen vorangegangenen Knoten in die Queue hinzugefügt. Dieser Vorgang wird solange wiederholt, bis entweder das Ziel erreicht wurde oder die Queue leer ist.

Algorithmus 14 Effizienzsteigerung durch selektive Knotenauswahl

- 1: Starte mit einer leeren Queue
 - 2: Füge den Startknoten zur Queue hinzu
 - 3: **while** Queue nicht leer **do**
 - 4: Nimm den nächsten Knoten aus der Queue
 - 5: **if** Knoten ist ein Zielknoten **then**
 - 6: Route zurückverfolgen und beenden
 - 7: **end if**
 - 8: Betrachte alle ausgehenden Kanten des Knotens
 - 9: **if** Knoten der ausgehenden Kante hat Grad 2 und ist nicht das Ziel **then**
 - 10: Gehe entlang der ausgehenden Kante weiter
 - 11: **while** nächster Knoten Grad = 2 und nicht das Ziel **do**
 - 12: Bewege weiter entlang der ausgehenden Kante dieses Knotens
 - 13: **end while**
 - 14: **end if**
 - 15: Füge den aktuellen Knoten mit seinen vorangegangenen Knoten in die Queue hinzu
 - 16: **end while**
 - 17: Beenden, keine Route gefunden
-

Durch die Verwendung dieses Ansatzes würde auch der Bedarf der Graphenkontraktion entfallen, da die Algorithmen bereits die Möglichkeit haben, entlang der Kanten weiter zu gehen. Dies ermöglicht es, dass die Laufzeit der Algorithmen weiter verbessert wird, da weniger Knoten

betrachtet werden müssen. Allerdings ist es notwendig, die Reihenfolge der Knoten zu aktualisieren, sodass eine Rückverfolgung der Route möglich ist. Weiterhin muss bei diesem Ansatz nur eine Route berechnet werden, was ebenfalls die Laufzeit verbessert.

5.3 Funktionalitäten

Neben den Optimierungsmöglichkeiten gibt es auch weitere Funktionalitäten, die die Anwendung verbessern können. Diese werden nachfolgend vorgestellt und erläutert.

5.3.1 Höchstgeschwindigkeit als Metrik

Neben der Optimierung der Routing-Algorithmen besteht auch die Möglichkeit, die Kostenfunktion um zusätzliche Metriken zu erweitern, wodurch realistischere und genauere Routen berechnet werden können. Eine sinnvolle Erweiterung der Kostenfunktion besteht in der Einbeziehung der zulässigen Höchstgeschwindigkeit der Wege, wodurch die berechneten Routen realistischer werden, da der Algorithmus bevorzugt Wege mit höheren Geschwindigkeiten wählt. So ermöglichen beispielsweise Autobahnen in der Regel eine schnellere Fahrt als Landstraßen oder Wohngebiete, sodass der Algorithmus bevorzugt Autobahnen nutzt, um die Reisezeit zu minimieren. Die Informationen zur Höchstgeschwindigkeit können dabei aus den OSM-Daten extrahiert und in die Kostenfunktion integriert werden. Allerdings erschwert die Berücksichtigung weiterer Metriken die Erstellung der in dieser Projektarbeit vorgestellten kontrahierten Kanten. Ein entscheidender Faktor ist dabei das Zusammenführen von Kanten mit unterschiedlichen Metrikwerten. So lässt sich beispielsweise für ein kontrahiertes Teilstück nicht einfach eine einheitliche Geschwindigkeit festlegen, wenn es aus Segmenten mit unterschiedlichen Höchstgeschwindigkeiten besteht. In einem solchen Fall müsste eine gewichtete Durchschnittsgeschwindigkeit berechnet werden, die die Länge der einzelnen Segmente berücksichtigt.

5.3.2 Filtern von Wegtypen

Eine weitere Funktionalität besteht darin, die Wegtypen zu filtern, die in der Berechnung berücksichtigt werden sollen. So kann es beispielsweise sinnvoll sein, nur Autobahnen oder Bundesstraßen in die Berechnung einzubeziehen, um ein anderes Verhalten der Algorithmen zu erreichen. Weiterhin können auch spezielle Wegtypen wie Brücken oder Tunnel ausgeschlossen werden, um die Berechnung zu beschränken und andere Routen zu erhalten. Dies ermöglicht es, die Anwendung an spezielle Anforderungen anzupassen und die Routenberechnung zu beeinflussen. Durch die Anpassung der Wegtypen muss jedoch auch die Erstellung der kontrahierten Kanten angepasst werden, um die Filterung der Wegtypen zu berücksichtigen. So darf beispielsweise ein kontrahiertes Teilstück nur aus einem Wegtyp bestehen, der in der Berechnung berücksichtigt wird. Ebenfalls ist es notwendig, dass die Filterung der Wegtypen in die Routing-Algorithmen integriert wird, um die Berechnung der Routen entsprechend anzupassen.

5.3.3 Unterschiedliche Detailtiefe

Eine weitere Anpassungsmöglichkeit besteht darin, die Detailtiefe der Kartenansicht auf das Zoom-Level abzustimmen. So können beispielsweise bei einem geringen Zoom-Level nur Autobahnen

und Bundesstraßen angezeigt werden, während bei einem hohen Zoom-Level auch kleinere Straßen und Wege sichtbar sind. Dies ermöglicht es, die Kartenansicht an die Anforderungen des Benutzers anzupassen und die Darstellung der Karte zu verbessern. Durch die Anpassung der Detailtiefe ist es jedoch notwendig, dass die Daten entsprechend aufbereitet und gefiltert werden, um die Anzeige der Karte zu optimieren.

Literaturverzeichnis

- [1] F Benjamin Zhan und Charles E Noon. „Shortest path algorithms: an evaluation using real road networks“. In: *Transportation science* 32.1 (1998), S. 65–73.
- [2] Sirine Ben Abbes, Lilia Rejeb und Lasaad Baati. „Route planning for electric vehicles“. In: *IET Intelligent Transport Systems* 16.7 (2022), S. 875–889. DOI: <https://doi.org/10.1049/itr2.12182>. URL: <https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/itr2.12182>.
- [3] Mordechai Haklay und Patrick Weber. „OpenStreetMap: User-Generated Street Maps“. In: *IEEE Pervasive Computing* 7.4 (2008), S. 12–18. DOI: [10.1109/MPRV.2008.80](https://doi.org/10.1109/MPRV.2008.80).
- [4] Peter Mooney und Marco Minghini. „A review of OpenStreetMap data“. In: London: Ubiquity Press, Sep. 2017, S. 37–59. DOI: [10.5334/bbf](https://doi.org/10.5334/bbf).c.
- [5] Daniel Delling u. a. „Engineering Route Planning Algorithms“. In: *Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, S. 117–139. ISBN: 978-3-642-02094-0. DOI: [10.1007/978-3-642-02094-0_7](https://doi.org/10.1007/978-3-642-02094-0_7). URL: https://doi.org/10.1007/978-3-642-02094-0_7.
- [6] Wolfgang Torge und Jürgen Müller. *Geodesy*. 4. Aufl. Berlin: Walter de Gruyter, 2012. ISBN: 978-3-11-025000-8.
- [7] John Parr Snyder. „Map projections–A working manual“. In: *US Geological survey professional paper* 1395 (1987).
- [8] B. Louis Decker. „World geodetic system 1984“. In: *Defense Mapping Agency Aerospace Center St Louis Afs Mo* (1986).
- [9] NIMA Technical Report TR8350.2. *Department of Defense World Geodetic System 1984: its definition and relationships with local geodetic systems*. Technical Report TR8350.2. National Imagery und Mapping Agency, 2000.
- [10] OpenStreetMap Contributors. *OpenStreetMap Wiki: Map Features*. https://wiki.openstreetmap.org/wiki/Map_Features. [Online; abgerufen am 14. Februar 2025]. 2025.
- [11] OpenStreetMap Contributors. *OpenStreetMap Wiki: Converting to WGS84*. https://wiki.openstreetmap.org/wiki/Converting_to_WGS84. [Online; abgerufen am 14. Februar 2025]. 2024.
- [12] Fossgis e.V. *Projections/Spatial reference systems*. <https://osmdata.openstreetmap.de/info/projections.html>. [Online; abgerufen am 14. Februar 2025]. 2025.
- [13] Christopher BARRON, Pascal NEIS und Alexander ZIPF. „iOSMAnalyzer- ein Werkzeug für intrinsische OSM Qualitätsuntersuchungen“. In: *Strobel, J. et al.(Hg.)(2013): Angewandte Geoinformatik* (2013).
- [14] Roland Olbricht u. a. „Overpass API“. In: *Anwenderkonferenz für Freie und Open Source Software für Geoinformationssysteme* (2011).

- [15] OpenLayers Contributors. *OpenLayers - A high-performance, feature-packed library for all your mapping needs*. <https://openlayers.org/>. [Online; abgerufen am 28. Februar 2025]. 2025.
- [16] T. Mitchell, A. Emde und A. Christl. *Web-Mapping mit Open Source-GIS-Tools*. O'Reilly, 2008. ISBN: 9783897217232. URL: <https://books.google.de/books?id=tgwcufnGX88C>.
- [17] Erik Hazzard. *Openlayers 2.10 beginner's guide*. Packt Publishing Ltd, 2011.
- [18] J. A. Bondy und U. S. R. Murty. *Graph Theory With Applications*. Springer, 2008. ISBN: 978-1-84628-969-9. URL: <https://www.zib.de/userpage/groetschel/teaching/WS1314/BondyMurtyGTWA.pdf>.
- [19] Mary Attenborough. „19 - Graph theory“. In: *Mathematics for Electrical Engineering and Computing*. Hrsg. von Mary Attenborough. Oxford: Newnes, 2003, S. 461–478. ISBN: 978-0-7506-5855-3. DOI: <https://doi.org/10.1016/B978-075065855-3/50045-0>. URL: <https://www.sciencedirect.com/science/article/pii/B9780750658553500450>.
- [20] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. [Online; abgerufen am 20. Februar 2025]. Addison-Wesley, 1984. URL: https://mat.uab.cat/~alseda/MasterOpt/Judea_Pearl-Heuristics_Intelligent_Search_Strategies_for_Computer_Problem_Solving.pdf.
- [21] Ping Zhang und Gary Chartrand. *Introduction to graph theory*. Bd. 2. 2.1. Tata McGraw-Hill New York, 2006.
- [22] Thomas Wolle. „A Note on Edge Contraction“. In: *Technical Report, Utrecht University* (2004). URL: https://dspace.library.uu.nl/bitstream/handle/1874/18000/wolle_04_a_note_on_edge.pdf.
- [23] Scott Beamer, Krste Asanović und David Patterson. „Direction-optimizing breadth-first search“. In: *Scientific Programming* 21.3-4 (2013), S. 137–148.
- [24] Edsger W. Dijkstra. „A Note on Two Problems in Connexion with Graphs“. In: *Numerische Mathematik* 1 (1959), S. 269–271.
- [25] Peter E. Hart, Nils J. Nilsson und Bertram Raphael. „A Formal Basis for the Heuristic Determination of Minimum Cost Paths“. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), S. 100–107.
- [26] Reinhard Diestel. *Graph Theory*. 6. Aufl. Berlin: Springer, 2025. ISBN: 978-3-662-70107-2.
- [27] Per-Erik Danielsson. „Euclidean distance mapping“. In: *Computer Graphics and image processing* 14.3 (1980), S. 227–248.
- [28] Kenneth Gade. „A non-singular horizontal position representation“. In: *The journal of navigation* 63.3 (2010), S. 395–417.
- [29] Wikipedia-Autoren. *Haversine-Formel — Wikipedia, die freie Enzyklopädie*. [Online; abgerufen am 20. Februar 2025]. 2025. URL: https://en.wikipedia.org/wiki/Haversine_formula.
- [30] MDN Web Docs. *Performance.now() - Web APIs / MDN*. [Online; abgerufen am 26. Februar 2025]. 2025. URL: <https://developer.mozilla.org/en-US/docs/Web/API/Performance/now>.
- [31] MDN Web Docs. *Generator - JavaScript / MDN*. [Online; abgerufen am 20. Februar 2025]. 2025. URL: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Generator.