Hochschule für angewandte Wissenschaften München
Fakultät für Geoinformation

Bachelorarbeit

Placeholder

Verfasser: Daniel Holzner

Matrikelnummer: 26576714

Studiengang: Geoinformatik und Navigation

Betreuer: Prof. Dr. Thomas Abmayr

Abgabedatum: 23. Juli 2023

Abstract

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

CH Contraction Hierarchie

OSM OpenStreetMap

PQ Vorrangwarteschlange (eng. Priority Queue)

API Application Programming Interface

1 Einleitung

1.1 Motivation und Kontext

Mit der fortschreitenden Entwicklung von Verkehrsmitteln und der dadurch zunehmenden Mobilität gewinnt die Routenplanung eine immer größere Bedeutung. Routenplanung ist ein faszinierendes und herausforderndes Gebiet, das eine wichtige Rolle in verschiedenen Bereichen spielt. Egal, ob es darum geht, den schnellsten Weg von einem Ort zum anderen zu finden, die effizienteste Route für die Zustellung von Waren zu bestimmen oder den Verkehr in einem komplexen Straßensystem zu simulieren, durch die Anwendung und Erforschung von Routenplanungsalgorithmen können effiziente und optimale Wege in komplexen Netzwerken gefunden werden, um Zeit, Ressourcen und Kosten zu sparen.

Das Problem, nach der Suche des kürzesten Weges, lässt sich auf eines der fundamentalsten Probleme aus der Graphentheorie, einem Teilgebiet der Mathematik und theoretischen Informatik zurückführen. Mit Graphen lassen sich eine Vielzahl von Problemen aus der echten Welt als mathematische Struktur, bestehend aus Knoten und Kanten, modellieren. So kann auch ein Straßennetzwerk durch Knoten, die Kreuzungen repräsentieren und Kanten, die als Straßensegmente Kreuzungen miteinander verbinden, dargestellt werden. Jeder Kante wird dabei ein Gewicht zugewiesen, das die mit dem Durchlaufen dieser Kante verbundenen Kosten widerspiegelt. Auf dem Graphen lassen sich anschließend Algorithmen ausführen, die den kürzesten Weg dist(s,t) zwischen einem Startknoten s und Zielknoten t bestimmen können, indem die Kosten des Weges minimiert werden. Einer der wohl bekanntesten Algorithmen, um diese Aufgabe zu lösen wurde von dem niederländischen Informatiker Edsger W. Dijkstra entwickelt und im Jahr 1959 veröffentlicht [6]. Der Dijkstra-Algorithmus funktioniert zwar gut auf kleinen Graphen und wird auch noch heutzutage häufig angewendet, skaliert jedoch schlecht mit immer größer werdenden Datenmengen, denn im schlechtesten Fall muss der gesamte Graph traversiert werden. Da ein Straßennetz aus mehreren Millionen Knoten und Kanten besteht, ist der Dijkstra-Algorithmus daher in Anwendungen, die in kurzer Zeit viele kürzeste Wege berechnen müssen, wie z. B. Navigationssysteme, die Routen in Echtzeit aktualisieren müssen, nicht mehr geeignet.
Um dieses Problem zu lösen, wurden im Laufe der Zeit neue Speed-Up Techniken entwickelt, die die Laufzeit der Suche verbessern. So wurde u. a. der A*-Algorithmus („A-Stern“) im Jahr 1968 von Peter Hart, Nils J. Nilsson und Bertram Raphael als eine Erweiterung des Dijkstra-Algorithmus veröffentlicht [14]. Der Algorithmus ist in der Lage durch das Einführen einer zusätzlichen Heuristik, orientierter Richtung Ziel zu suchen und damit den Suchraum deutlich

einzuschränken. Dadurch wurde die Laufzeit nochmals verbessert, war aber immer noch nicht ausreichend für sehr große Graphen.
Viele weitere Techniken basieren auf einer starken Vorverarbeitung des Graphen. So wurde 2008 von Geisberger, Sanders, Schultes, und Delling vorgestellt [12], die als Contraction Hierarchies (CHs) bezeichnet wird. Sie basiert auf einer Vorverarbeitung des Graphen, bei der ausgenutzt wird, dass Straßennetzwerke bereits eine natürliche Hierarchie besitzen. Während der Vorverarbeitung werden dem Graph zusätzliche Informationen hinzugefügt, die dann zur Laufzeit während der Suche ausgenutzt werden, was zu erheblich schnelleren Berechnung der Route führt. Da diese Technik als Sprungbrett für viele neue erweiterte Routenplanungstechniken gilt, soll sie im Rahmen dieser Arbeit genauer untersucht werden. Dazu soll ein Prototyp erstellt werden, der die Funktionsweise von CHs durch eine konkrete Implementierung demonstriert. Als Eingangsdaten werden die frei nutzbaren Geodaten des OpenStreetMap-Projekts (Open Data) verwendet, um die Ergebnisse an realen Daten zu analysieren und zu testen.

1.2 Stand der Forschung

Die Berechnung von kürzesten Wegen in dynamischen gewichteten Graphen ist in den letzten Jahrzehnten intensiv untersucht worden und es entstanden viele neue Techniken zur Lösung verschiedener Varianten des Problems. Das kürzeste-Wege-Problem ist so relevant, dass regelmäßig Wettbewerbe stattfinden, bei denen die aktuell besten Routenplanungsalgorithmen auf speziellen Eingabedaten ermittelt werden. So wurde z. B. 2006 die neunte DIMACS Implementation Challenge [1] ausgerichtet, in der die „State of the Art“-Techniken vorgestellt wurden. Eine ausführliche Übersicht über verschiedene Algorithmen zur Routenplanung in Straßennetzwerken wurde von Delling et al. [5] veröffentlicht, ist aber durch die signifikante Weiterentwicklungen der letzten Jahre nicht mehr topaktuell. Es sind neue Algorithmen entstanden, die Suchanfragen auf Straßennetzwerken in der Größenordnung von Kontinenten in wenigen hundert Nanosekunden beantworten können oder aktuelle Verkehrsinformationen mit in die Suche einfließen lassen [2]. Durch den aktuellen Stand der Technik können Methoden des maschinellen Lernens verwendet werden, um je nach Situation den Verkehrsfluss in Echtzeit vorherzusagen und mit in der Routenplanung zu berücksichtigen [17].
Diese Entwicklungen zeigen, dass die Routenplanung in Straßennetzwerken ein aktiver Forschungsbereich ist, der sich ständig weiterentwickelt, um den Bedürfnissen der Nutzer und modernen Anwendungen gerecht zu werden.

1.3 Beitrag der Arbeit

Im Rahmen dieser Arbeit wird die Implementierung der CHs-Technik in der Programmiersprache *Rust* untersucht. Rust ist eine moderne, systemsprachenorientierte Programmiersprache, die auf Performance, Sicherheit und Nebenläufigkeit abzielt [15]. Die Wahl von Rust als Implementierungssprache bietet die Möglichkeit, die Vorteile dieser Sprache in Bezug auf Geschwindigkeit, Speichersicherheit und Thread-Sicherheit in der Routenplanung zu erforschen.
Das Hauptziel dieser Arbeit ist, die Implementierung von CHs detailliert zu beschreiben und anschließend eine umfassende Leistungsanalyse durchzuführen, indem die Ergebnisse mit herkömmlichen Algorithmen wie dem Dijkstra-Algorithmus und dem A*-Algorithmus verglichen werden. Die Evaluierung erfolgt anhand verschiedener Metriken wie Laufzeit, Vorverarbeitungszeit und Speicherbedarf. Durch diesen Vergleich wird ein tieferes Verständnis für die Leistungsfähigkeit von CHs gewonnen und die verbundenen Vor- und Nachteile im Vergleich zu herkömmlichen Algorithmen ermittelt.
Die Ergebnisse dieser Arbeit können wichtige Erkenntnisse liefern, die zur Weiterentwicklung und Optimierung von Routenplanungssystemen beitragen. Darüber hinaus bietet die Implementierung in Rust einen wertvollen Beitrag zur wachsenden Gemeinschaft von Rust-Entwicklern und demonstriert die Anwendung der Sprache in einem relevanten Anwendungsfall.

1.4 Struktur der Arbeit

Die folgenden Abschnitte der Arbeit werden die theoretischen Grundlagen von Routenplanungsalgorithmen, insbesondere des Dijkstra-Algorithmus, des A*-Algorithmus und der CH-Technik, erläutern. Anschließend wird die Implementierung der CH-Technik in Rust beschrieben und detailliert analysiert. Abschließend werden die Ergebnisse der Leistungsanalyse präsentiert und diskutiert, gefolgt von einem Fazit, das die Erkenntnisse dieser Arbeit zusammenfasst und mögliche Ansätze für zukünftige Forschungen aufzeigt.

2 Theoretische und technische Grundlagen

2.1 Datenstrukturen

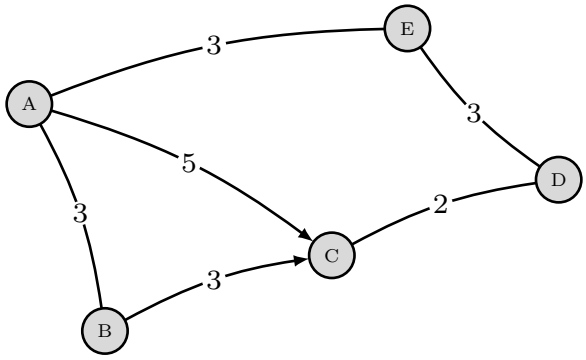
2.1.1 Graph

Um das kürzeste-Wege-Problem zu lösen, muss zunächst das reale Straßennetz in eine abstrakte Form gebracht werden. Hierzu wird das Straßennetz als Graph modelliert. Ein Graph $G = (V, E)$ besteht aus einer Menge von Knoten V und einer Menge von Kanten E. Jede Kante $e = (u, v) \in E$ verbindet zwei Knoten $u, v \in V$. Knoten bilden Kreuzungen ab und Kanten Straßensegmente zwischen zwei Kreuzungen. Ein Graph kann als gerichtet oder ungerichtet definiert werden. Bei einem ungerichteten Graphen sind die Kanten bidirektional und können in beide Richtungen durchlaufen werden, während bei einem gerichteten Graphen die Kanten nur in eine Richtung durchlaufen werden können.

Der Graph in dieser Arbeit ist gerichtet und gewichtet. Die Kantenrichtung spiegelt dabei die Richtung des Verkehrs wieder, d. h. sie gibt an, ob eine Straße in einer bestimmten Richtung befahren werden darf oder nicht. Zusätzlich wird auf jede Kante eine Kostenfunktion angewandt, die dann ein Gewicht $w(e)$ festgelegt. In der Routenplanung ist normalerweise nicht die kürzeste Strecke von Interesse, sondern die Strecke mit der kürzesten Reisezeit. Diese lässt sich aus der Länge des Straßensegments S und der maximal erlaubten Geschwindigkeit V auf dieser Straße $t = \frac{V}{S}$ in Sekunden berechnen.

Es gibt generell zwei Möglichkeiten, einen Graphen als Datenstruktur darzustellen. Eine davon ist die Verwendung von Adjazenzlisten. Dabei wird für jeden Knoten $u \in V$ eine Liste der benachbarten Knoten bzw. ausgehenden Kanten $e(u, v) \in E$ gespeichert. Die Summe der Länge aller Adjazenzlisten in einem gerichteten Graph entspricht der Anzahl an Kanten $|E|$, bzw. $2|E|$ in einem ungerichteten Graph. Der Speicherverbrauch für die Adjazenzliste ist damit $\Theta(|V| + |E|)$ [4].
Die zweite Möglichkeit ist die Verwendung einer Adjazenzmatrix. Hier wird eine zweidimensionale Matrix der Größe $|V| \times |V|$ verwendet, bei der die Zeilen und Spalten den Knoten entsprechen. Der Eintrag an Position (i, j) in der Matrix gibt an, ob eine Kante zwischen den Knoten i und j existiert. Unabhängig von der Anzahl an Kanten ist der Speicherverbrauch $\Theta(|V|^2)$ [4]. Ein Beispiel für beide Darstellungen anhand eines einfachen Graph befindet sich in Abbildung 1.

Für diese Arbeit werden Adjazenzlisten verwendet, da zum einen Straßennetze dünne Graphen sind ($|E| << |V|^2$), und damit die Speichereffizienz gegenüber einer Adjazenzmatrix deutlich effizienter ist. Zum anderen ist für es wichtig, für den Aufbau der **CHs!** (**CHs!**) und der Suche, schnell auf alle Nachbarn eines Knotens zuzugreifen zu können. Der Zugriff auf eingehende Kanten ist allerding nur sehr aufwendig möglich, wird aber für den Aufbau der Kontraktionshierarchien benötigt. Das Problem lässt sich jedoch lösen indem auch Adjazenzlisten für alle eingehenden Kanten eines Knotens angelegt werden. Damit erhöht sich der Speicherverbrauch auf $\Theta(|V|+2|E|)$, was aber immer noch deutlich effizienter ist als eine Darstellung als Adjazenzmatrix.



(a)

	A	B	C	D	E
A	0	3	5	0	3
B	3	0	3	0	0
C	0	0	0	2	0
D	0	0	2	0	3
E	3	0	0	3	0

(b)

A →	B	C	E
B →	A	C	
C →	D		
D →	C	E	
E →	A	D	

(c)

Abbildung 1: Der oben gezeigte gerichtete Graph (a) dargestellt als Adjazenzmatrix(b) oder Adjazenzliste (c). Viele Einträge der Matrix sind 0, da der Graph dünn ist, d. h. $|E|$ um einiges kleiner ist als $|V^2|$.

2.1.2 Vorrangwarteschlangen

Eine Vorrangwarteschlange (eng. Priority Queue) (PQ) ist eine Datenstruktur, in der nur auf das Element mit der höchsten Priorität zugegriffen werden kann. Eine PQ unterstützt in der Regel die folgenden Operationen:

1. Einfügen (Push): Ein Element wird mit seiner zugehörigen Priorität in die Warteschlange eingefügt. Das Element wird entsprechend seiner Priorität platziert.

2. Entfernen (Pop): Das Element mit der aktuell höchsten Priorität wird aus der Warteschlange entfernt und zurückgegeben.

PQs werden allen nachfolgenden Suchalgorithmen verwendet, daher hat die Implementierung der Warteschlange einen großen Einfluss auf die Laufzeit der Algorithmen. In der Arbeit wird die Implementierung als binärern Min-Heap verwendet mit einer Zeitkompexität für das Einfügen von $\theta(1) \sim$ und für das Entfernen von $\theta(\log n)$.

2.2 Kürzeste-Wege-Algorithmen

Grundsätzlich arbeiten kürzeste-Wege-Algorithmen daran, den kürzesten Weg zwischen einem Startknoten s und einem Zielknoten t in einem gewichteten Graph zu finden. Der kürzeste Weg P bezieht sich dabei auf den Weg mit dem geringsten Gesamtgewicht $dist(s,t)$, der sich aus der Summe der Kosten zum Überqueren der einzelnen Kanten ergibt. Neben dem Point-to-Point-kürzeste-Wege-Problem existieren noch weitere Varianten, wie das One-to-Many-Problem, bei dem der kürzeste Weg von einem Knoten s zu allen anderen Knoten im Graphen gesucht wird, oder das Many-to-Many-Problem, bei dem jeder kürzeste Weg zwischen einer Knotenmenge S und einer Knotenmenge T gesucht wird [2]. In dieser Arbeit wird sich hauptsächlich auf das Point-to-Point-Problem fokusiert.

2.2.1 Grundlegende Technik

Der Algorithmus von Dijkstra löst das One-to-Many-Problem auf einem gewichteten gerichten Graphen. Dabei ist zu beachten dass alle Kanten $e \in E$ nicht-negativ gewichtet sind, also $e(u,v) \geq 0$.
Der Algorithmus besteht aus einer Initialisierungsphase, in der die Kosten *dist* für alle Knoten auf den Wert ∞ gesetzt werden. Um am Ende nicht nur die Kosten, sondern auch den Weg zu erhalten wird zusätzlich der direkte Vorgängerknoten für jeden Knoten in P gespeichert. In folgender Implementierung wird eine Min-Vorrangwarteschlange Q verwendet, in der zu Beginn der Startknoten s mit Kosten 0 eingefügt wird. In der Hauptschleife wird nun solange ein Knoten $u \in V - S$ aus der Warteschlange entnommen, bis diese leer ist oder der Zielknoten erreicht wurde. Wenn ein Nachbarknoten v von u noch nicht besucht wurde, werden die Kosten $dist(v)$ aktualisiert, falls der Weg über u kürzer ist. Der Vorgänger $P(v)$ wird ebenfalls aktualisiert und der Knoten v in die Warteschlange eingefügt. Wenn der Zielknoten gefunden wurde, dann wird der kürzeste Weg mithilfe der gemerkten Vorgängerknoten ausgehend von t rekonstruiert und zurückgegeben. Durch Weglassen des Abbruchskrieterium in Zeile 8–10

kann der Algorithmsu von einer Point-to-Point suche zu einer One-to-Many Suche erweitert werden, sodass jeder Knoten im Graph traversiert wird.

Algorithm 1 Dijkstra Point-To-Point

```
1:  function DIJKSTRA(G=(V,E), s,t)
2:      Q ← ∅
3:      dist(s) = 0
4:      P ← ∅                                  ▷ Vorgängerknoten für Pfadrekonstruktion
5:      Q ← Q ∪ {s}                                        ▷ Prioritätswarteschlange
6:      while Q ≠ ∅ do
7:          u =← pop(Q)
8:          if u = t then
9:              return shortest_path(P,t)
10:         end if
11:         for each ausgehende Kante e(u,v) ∈ Adj[u] do
12:             tentative_distance ← dist(u) + w(u,v)
13:             if tentative_distance < dist(v) then
14:                 dist(v) ← tentative_distance
15:                 P(v) ← u
16:                 Q ← Q ∪ {v}
17:             end if
18:         end for
19:     end while
20: end function
```

Die Laufzeit des Algorithmus hängt hauptsächlich von der Implementierung der Prioritätswarteschlange ab. Mit der verwendeten Implementierung ergibt sich eine Laufzeit von $\Theta(|E| + |V| \log |V|)$, wenn Adjazenzlisten verwendet werden [4].

2.2.2 Bidirektionale Suche

Eine Variante des klassischen Dijkstra-Algorithmus ist die Erweiterung um Bidirektionalität. Im Gegensatz zum herkömmlichen Algorithmus, der nur in einer Richtung vom Startknoten zum Zielknoten arbeitet, führt der bidirektionale Dijkstra-Algorithmus eine Suche von beiden Knoten gleichzeitig durch. Die Rückwärtssuche traversiert dabei den transponierten Graphen, d. h. die Richtung der Kanten sind invertiert.
Der Algorithmus verwendet zwei Prioritätswarteschlangen, eine für die Vorwärtsrichtung und eine für die Rückwärtsrichtung. Der Ablauf des Algorithmus ist in Abbildung 2 illustriert.

1. Initialisierung: Jede Prioritätswarteschlange wird mit dem Startknoten bzw. dem Zielknoten initialisiert. Die vorläufigen Werte werden auf 0 für den Startknoten bzw. auf

unendlich für den Zielknoten gesetzt.

2. Expansionsschritt: In jedem Schritt wird der Knoten mit dem kleinsten vorläufigen Wert von beiden Warteschlangen ausgewählt. In der Schlange mit dem kleineren Knoten wird eine Dijkstra-Iteration ausgeführt.

3. Überprüfung des Treffpunkts: Bei jedem Expansionsschritt wird überprüft, ob der aktuelle Knoten in beiden Richtungen erreicht wurde. Wenn ein Knoten in beiden Richtungen erreicht wurde, gibt es einen potenziellen Pfad von Start zu Ziel.

4. Terminierung: Der Algorithmus terminiert, wenn beide Prioritätswarteschlangen leer sind oder ein Treffpunkt gefunden wurde.

5. Kürzester Weg: Für jeden Knoten u der von beiden Seiten erreicht wurde (u besitzt eine endliche Distanz von s und t), wird die Summe $dist(s,t) = dist(s,u) + dist(t,u)$ berechnet. Der kürzeste Weg verläuft über den Knoten, bei dem die Summe minimal ist.

6. Rückverfolgung des kürzesten Pfades: Wenn ein Treffpunkt u gefunden wurde, kann der kürzeste Pfad durch Rückverfolgung der Vorgängerknoten von Startknoten bis zum Treffpunkt in der Vorwärtsrichtung und von Zielknoten bis zum Treffpunkt in der Rückwärtsrichtung rekonstruiert werden.

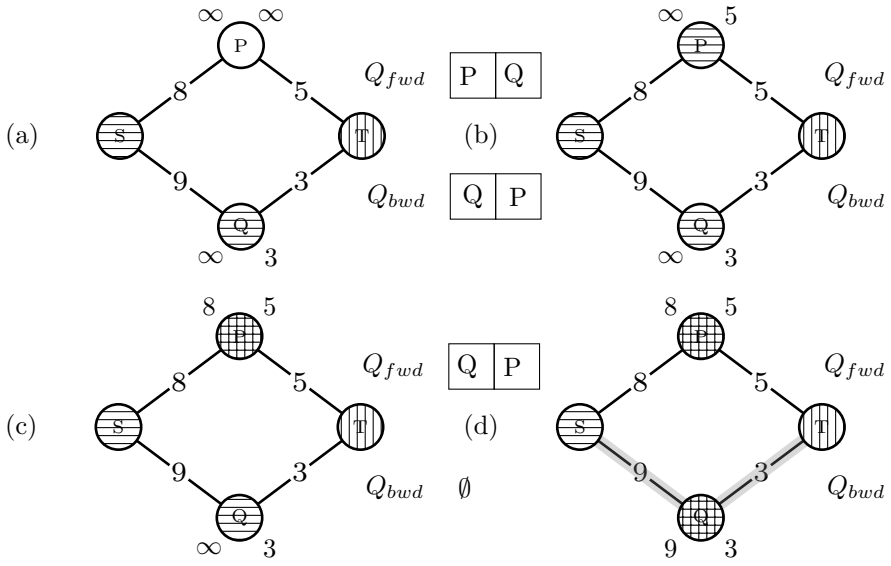


Abbildung 2: Bidirektionale Suche von S nach T: In Schritt (a) und (b) wird zuerst Knoten Q und P von der Rückwärtssuche relaxiert. In den Schritten (c) und (d) wird P und Q von der Vorwärtssuche relaxiert. Obwohl P der erste Knoten ist, der von beiden Suchen relaxiert wurde, ist der Weg über Q am kürzesten.

Der bidirektionale Dijkstra Algorithmus kann die Anzahl der untersuchten Knoten im Vergleich zum herkömmlichen Dijkstra reduzieren, insbesondere in großen Graphen. Durch die gleichzeitige Suche in beiden Richtungen kann er die Laufzeit verbessern, indem er die Anzahl der Expansionsschritte und die Anzahl der Knoten, die in Betracht gezogen werden, reduziert. In der Praxis wird der Suchraum etwa um die Hälfte reduziert (siehe Abbildung 3):

$$A = \pi r^2$$
$$A_{bidir} = 2\pi(\frac{r}{2})^2 = \frac{A}{2}$$

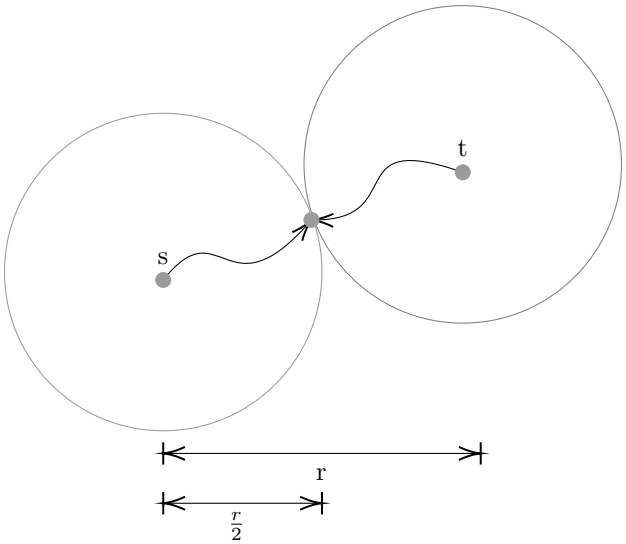


Abbildung 3: Suchraum einer bidirektioanlen Suche. Die Anzahl der Knoten die untersucht werden müssen, halbiert sich.

Korrektheitsbeweis

Lemma 2.1. *Für alle Knoten u die von beiden Seiten erledigt wurden, gilt:*
$$dist(s,t) = min\{dist(s,u) + dist(t,u)\}.$$

Beweis. Gegeben sei der Graph in Abbildung 4. Die Länge des kürzesten Pfads von s nach t wird als D bezeichnet. Der Knoten p ist der erste Knoten der von beiden Suchen erledigt wurde. Wenn gilt $dist(s,p) = dist(t,p)$, dann ist p garantiert auf dem kürzesten Weg.
Angenommen $dist(s,p)$ und $dist(t,p)$ sind ungleich D/2, dann muss entweder $dist(s,p) < D/2$ oder $dist(t,p) < D/2$ gelten. Dies wiederum bedeutet , dass alle Knoten mit einem kürzesten-Pfad-Wert, der kleiner oder gleich $D/2$ ist, bereits gesetzt wurden.

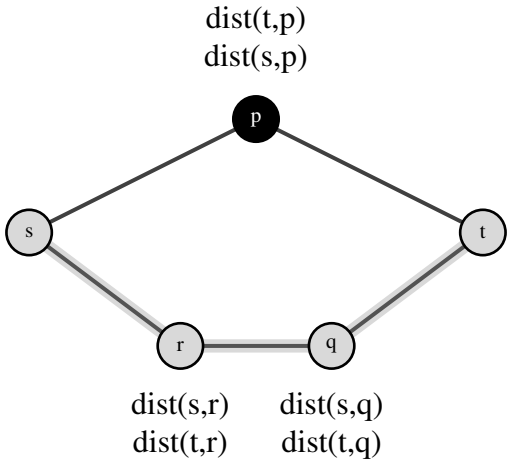


Abbildung 4: Graph für den Korrektheitsbeweis der bidirektionalen Suche

Knoten r und q liegen auf dem kürzesten Weg von s nach t und $dist(s,r) \leq D/2$ und $dist(t,q) \leq D/2$. Knoten r wurde von s erledigt und Knoten q von t. Dann wurde die Kante $e(r,q)$ bereits von beiden Seiten aus relaxiert uns somit haben r und q Distanzwerte von beiden Richtungen erhalten.
Die länge des kürzesten Weges von s nach t ist der kleinere Wert aus $dist(s,r)+dist(t,r)$ und $dist(s,q)+dist(t,q)$, welche in diesem Fall den gleichen Wert haben.
Somit ist der Knoten, der zuerst von beiden Seiten erledigt wurde nicht unbedingt auf dem kürzesten Weg, aber es wurde zumindest ein Knoten mit einer kürzeren Distanz gefunden, der auf dem kürzesten Weg liegt und einen Distanzwert von beiden Seiten erhalten hat. □

2.2.3 Zielorientierte Suche

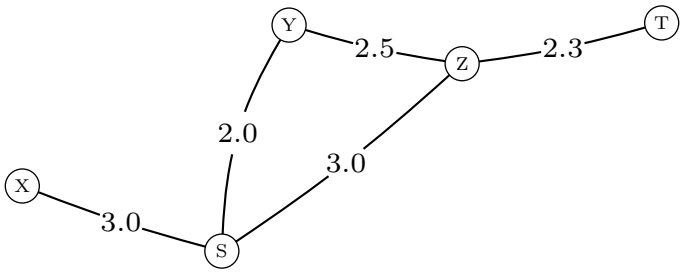
Die Technik der Zielorientierten Suche (engl. Goal-Directed Search) wurde erstmals in [14] vorgestellt und ist im allgemeinen als A* -Algorithmus („A-Stern“) bekannt. Hier wird jeder vorläufigen Distanz $dist(u)$ ein zusätzliches Potential $\rho_t : V \to \mathbb{R}_0^+$ (auch Heuristik genannt) hinzugefügt. Die Priorität eines Knotens in der Prioritätswarteschlange ergibt sich damit aus der Summe $dist(u)+\rho_t(u)$. Das Potential hat den Effekt, dass Knoten die in Richtung Ziel führen bevorzugt werden, da sie einen niedrigeren ρ-Wert besitzen.
Das Verändern der Prioritätswerte kann auch als Veränderung der Kantengewichte verstanden werden. Jede Kante $e(u,v) \in E$ wird neu gewichtet, sodass gilt:

Definition 2.1. $\bar{w}(u,v) = w(u,v) - \rho_t(u) + \rho_t(v)$

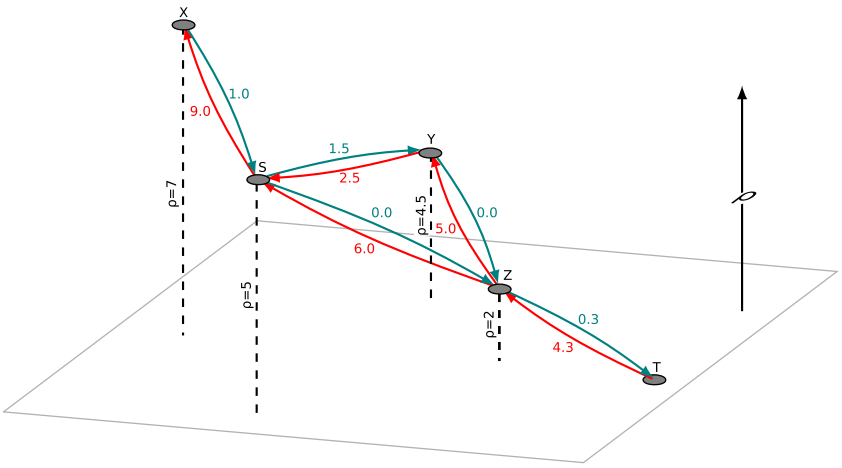
Kanten, die vom Ziel wegführen werden länger und Kanten die in Richtung Ziel führen werden kürzer. Nachdem ein Potential gewählt wurde und alle Kanten neu gewichtet wurden, kann

Dijkstra wie gewohnt auf dem neuen Graph ausgeführt werden. Durch ein gut gewähltes Potential wird die Suche in die Richtung des Ziels gelenkt und somit die Laufzeit verringert [18].

Abbildung 5 zeigt einen $\bar{G}$, in dem alle Kantengewichte neu gewichtet wurden. Knoten die weg vom Ziel führen haben ein höheres Potential und sind schwerer zu erreichen, da mehr Arbeit aufgebracht werden muss, um nach oben zu klettern. Da die Bewegung nach unten bevorzugt wird, ist es daher wahrscheinlicher das Ziel T früher zu treffen.



(a) Graph vor Neugewichtung der Kanten



(b) Graph nach Neugewichtung der Kanten

Abbildung 5: Graph vor (a) und nach (b) Neugewichtung der Kantengewichte. Das Potential ρ ist als Höhe dargestellt. Kanten die nach „oben“ verlaufen sind schwerer zu erreichen als Kanten die nach „unten“ verlaufen.

Der kürzeste Weg von S nach T bleibt erhalten:

$$dist_G(S,T) = w(S,Z) + w(Z,T) \quad = 3.0 + 2.3 \quad = 5.3$$
$$dist_{\bar{G}}(S,T) = \bar{w}(S,Z) + \bar{w}(Z,T) + \rho_t(S) \quad = 0.0 + 0.3 + 5.0 \quad = 5.3$$

Da nach Definition 2.1 die Längen der Kanten bzw. die Gewichte umgeschrieben werden, ist es naheliegend, dass bei einem schlecht gewählten Potential Kantengewichte negativ werden können. Wenn dies passiert, kann Dijkstra nicht mehr ausgeführt werden, da der Algorithmus nur für nicht-negative Kanten definiert ist. Um dies zu verhindern, muss das Potential so gewählt werden, dass gilt:

Definition 2.2. In einem gewichteten Graph $G = (V,E)$ muss ein Potential ρ_t so gewählt werden, dass $w(u,v) - \rho_t(u) + \rho_t(v) \leq 0$ für alle $e(u,v) \in E$. Die Bedingung ist erfüllt, wenn $\rho_t(u) \leq dist(u,t)$ für jeden Knoten $u \in V$.

Die Potentialfunktion ist damit eine optimistische Schätzung und untere Schranke. Sie schätzt den Weg von u nach t immer kleiner oder gleich dem tatsächlichen kürzesten Weg ein und verhindert damit, dass Kanten negative Gewichte erhalten können.
Da die Koordinaten der Knoten im Graphen eine Straßennetzes in der Regel bekannt sind, ist die euklidische bzw. Großkreisdistanz als Potential eine gute Wahl:

$$\rho_t(u) = \sqrt{(x_t - x_u)^2 + (y_t - y_u)^2} \tag{1}$$

Der Ansatz mit einer kompletten Neugewichtung aller Kanten ist zwar intuitiv, aber in der Praxis nicht sinnvoll, da es aufwendig ist jede Kante neu zu berechnen. Stattdessen erfolgt die A* -Implementierung analog zu Dijkstra (siehe 1). Der einzige Unterschied des Algorithmus (siehe 2) findet sich in Zeile 7 wieder. Hier wird nicht mehr das Element mit der niedrigsten Distanz extrahiert sonder das Element mit der niedrigsten Distanz plus der Heuristik λ. Wenn für die Heuristik die Nullheuristik $\lambda_0 : V \to 0$ gewählt wird, dann verhält sich A* identisch zu Dijkstra.

2.3 OpenStreetMap

OpenStreetMap (OSM) ist ein Projekt, das sich der Erstellung und Bereitstellung von freien geografischen Daten verschrieben hat. Es handelt sich dabei um Daten die von einer weltweiten Gemeinschaft von Freiwilligen erstellt und gepflegt wird [8].
Das grundlegende Konzept von OpenStreetMap basiert auf OpenData, was bedeutet, dass die Daten frei verfügbar und für jeden zugänglich sind. Im Gegensatz zu kommerziellen Kartenanbietern, die ihre Daten schützen und für den Zugriff hohe Gebühren verlangen, ermutigt OpenStreetMap Menschen dazu, ihre eigenen Daten beizutragen und von den vorhandenen Daten zu profitieren.

Algorithm 2 AStar Implementierung

```
1:  function ASTAR(G = (V,E), s, t, λ)
2:      Q ← ∅
3:      P ← ∅                                   ▷ Vorgängerknoten für Pfadrekonstruktion
4:      dist(s) ← 0
5:      Q ← Q ∪ {s}
6:      while Q ≠ ∅ do
7:          u ← node in Q minimizing dist(node) + λ(node)      ▷ Heuristik berücksichtigen
8:          if u = t then
9:              return shortest_path(P,t)
10:         end if
11:         for v ∈ adj(u) do
12:             tentative_distance ← dist(u) + w(u,v)
13:             if tentative_distance < dist(v) then
14:                 dist(v) ← tentative_distance
15:                 P(v) ← u
16:                 Q ← Q ∪ {v}
17:             end if
18:         end for
19:     end while
20: end function
```

Das OpenStreetMap-Projekt verwendet eine Kombination aus verschiedenen Datenquellen, um eine detaillierte und umfassende Karte zu erstellen. Dazu gehören zum Beispiel Satellitenbilder, GPS-Tracks, Luftaufnahmen und auch öffentlich verfügbare geografische Daten. Die Daten werden von Freiwilligen erfasst, indem sie Straßen, Gebäude, Gewässer, Landnutzung und andere geografische Merkmale auf der Karte markieren oder Informationen darüber hinzufügen.

Die OpenStreetMap-Daten sind unter einer offenen Lizenz, der Open Data Commons Open Database Lizenz (ODbL), verfügbar. Das bedeutet, dass die Daten frei verwendet, kopiert, modifiziert und weiterverbreitet werden können, solange die Lizenzbedingungen eingehalten werden [9]. Dadurch wird Entwicklern ermöglicht, die OpenStreetMap-Daten in ihre eigenen Anwendungen und Dienste zu integrieren.

3 Methodik

Dieses Kapitel beschreibt die Methodik der Arbeit. Zunächst wird die verwendete Datenbasis erläutert und die Datenbeschaffung beschrieben. Anschließend wird die verwendete Methode zur Berechnung der kürzesten Wege vorgestellt.

3.1 Datenbeschaffung und -aufbereitung

3.1.1 Datenformat

Das OpenStreetMap (OSM)-Datenformat ist ein XML-basiertes Format, das aus drei verschiedenen Elementen besteht:

- Knoten (Nodes): Knoten repräsentieren einzelne Punkte im Raum und werden durch ihre Längen- und Breitengrade (Geokoordinaten) definiert. Sie können Attribute wie Tags (Schlüssel-Wert-Paare) enthalten, um Sachdaten über den Punkt zu speichern, z. B. den Namen, die Art des Ortes oder andere Eigenschaften.
- Wege (Ways): Wege bestehen aus einer geordneten Liste von Knoten und repräsentieren Linienzüge. In der Regel werden Straßen, Flüsse und Grenzen als Wege dargestellt. Wege können auch Tags enthalten, um zusätzliche Informationen über den Linienzug zu speichern, z. B. den Straßentyp oder die Geschwindigkeitsbegrenzung.
- Relationen (Relations): Relationen bestehen aus ein oder mehreren Attributen und einer sortierten Liste von Datenelementen (Wege, Knoten, Relationen), wobei jedem Datenelement eine Rolle zugewiesen wird. Sie werden verwendet, um logische oder geometrische Zusammenhänge zwischen anderen Elementen zu beschreiben.

Grundsätzlich können beliebige Tags gesetzt werden, es haben sich jedoch zwei Arten von Tags etabliert:
Klassifizierende Tags haben einen von wenigen Schlüsseln, für jeden der wenigen Schlüssel gibt es nur eine begrenzte Anzahl von Werten. Abweichende Werte werden als Fehler betrachtet. So wird das gesamte öffentliche Straßennetz für Kraftfahrzeuge mit dem Schlüssel *highway* und einem von wenigen gemeinsamen Werten identifiziert. Für Gebäude ist in den meisten Fällen nur *building* mit dem Wert *yes* definiert.
Im Gegensatz dazu haben die *beschreibenden* Tags nur feste Schlüssel, während der Wert ein freier Text ist, der in Groß- und Kleinbuchstaben geschrieben werden kann und auch Sonderzeichen enthalten kann. Der Hauptanwendungsfall sind Namen. Dazu können Beschreibungen, Bezeichner oder auch Größenangaben kommen.

Für diese Arbeit sind vorallem die Elemente *Node* und *Way* relevant, sowie der Tag *highway*. Abbildung 6 zeigt ein Beispiel eines OSM-Dokuments. Es enthält drei Knoten und einen Weg. Der Weg referenziert die drei Knoten über die `ref`-Attribute. Die Straße ist vom Typ „residential“, also eine Straße für Anlieger. Die weiteren Tags geben außerdem die Information, dass die Straße eine Einbahnstraße ist und dass maximal 30 km/h erlaubt sind.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <osm version="0.6" upload="true" generator="OpenStreetMap server">
3    <bounds minlat="51.5073601795557" minlon="-0.108157396316528" maxlat
         ="51.5076406454029" maxlon="-0.107599496841431"/>
4    <node id="1" lat="51.5074089" lon="-0.1080108" timestamp="1970-01-01T00
         :00:01Z" user="matt" visible="true" version="1"/>
5    <node id="2" lat="51.5074343" lon="-0.1080123" timestamp="1970-01-01T00
         :00:01Z" user="snsn1" visible="true" version="1"/>
6    <node id="3" lat="51.5075933" lon="-0.1076109" timestamp="1970-01-01T00
         :00:01Z" user="snsn1" visible="true" version="1"/>
7    <way id="12" action="modify" timestamp="1970-01-01T00:00:01Z" visible="
         true" version="3">
8      <nd ref="1" />
9      <nd ref="2" />
10     <nd ref="3" />
11     <tag k="highway" v="residential" />
12     <tag k="oneway" v="false" />
13     <tag k="maxspeed" v="30"/>
14     <tag k="name" v="Amselweg"/>
15   </way>
16 </osm>
```

Abbildung 6: Beispiel eines OSM-XML-Dokuments. Es enthält drei Knoten und einen Weg. Hierbei handelt es sich um eine Anliegerstraße (erkennbar am Tag `highway=residential`) mit erlaubter Höchstgeschwindigkeit von 30 km/h.

Es gibt keine klaren Vorgaben zum Taggen von Objekten nur Empfehlungen. Daher ist eine große Variation an Tags zu erwarten, die ensprechend behandelt werden müssen.

3.1.2 Datenextraktion

Die erste Aufgabe besteht darin, die relevanten Daten aus dem OpenStreetMap-Datensatz zu extrahieren. Hierzu gibt es zahlreiche Möglichkeiten und Werkzeuge. Die Geofabrik GmbH Karlsruhe (Mitglied der OSM-Foundation) bietet u. a. ein Service an, bei dem sich bereits vorgefertigte Regionen (hierarchisch unterteilt in Kontinente, Länder, Bundesländer, etc.) direkt herunterladen lassen. Die Datensätze werden in der Regel täglich aktualisiert und sind

als OSM- oder PBF-Format[1] verfügbar [16].
Eine weitere Möglichkeit ist die Verwendung von Application Programming Interfaces (APIs) wie z. B. der Overpass-API[2], eine kostenlose Schnittstelle mit der sich gezielt OSM-Daten extrahieren lassen. Die Abfragen werden in der Overpass Query Language (kurz: Overpass QL) erstellt, mit der sich Daten u. a. nach bestimmten Tags filtern lassen, z. B. nach bestimmten Straßentypen oder Straßennamen [10].
In der Arbeit werden hauptsächlich vorgefertigte Daten im PBF-Format verwendet und verarbeitet. Zur Untersuchung von kleinen Gebieten wird die Pyhtonbibliothek OSMnx [3] verwendet, die es ermöglicht, OSM-Daten über eine Highlevel-API (intern: Overpass) abzufragen, zu verarbeiten und zu visualisieren.

3.1.3 Datenverarbeitung

Zunächst wird die *PBF*-Datei eingelesen und geparst. Ein Problem der Rohdaten ist, dass viele Elemente der Daten für die Routenplanung nicht relevant sind. Dazu zählen alle Elemente, die keine Straßen sind (z. B. Gebäude, Flüsse, etc.) sowie alle Straßen, die nicht für den Autoverkehr zugelassen sind (z. B. Fußgängerzonen, Radwege, etc.). Diese Elemente werden daher während des Parsens der Datei gefiltert und aus dem Graphen entfernt, sodass ein Netzwerk wie in Abbildung 7 übrigt bleibt.

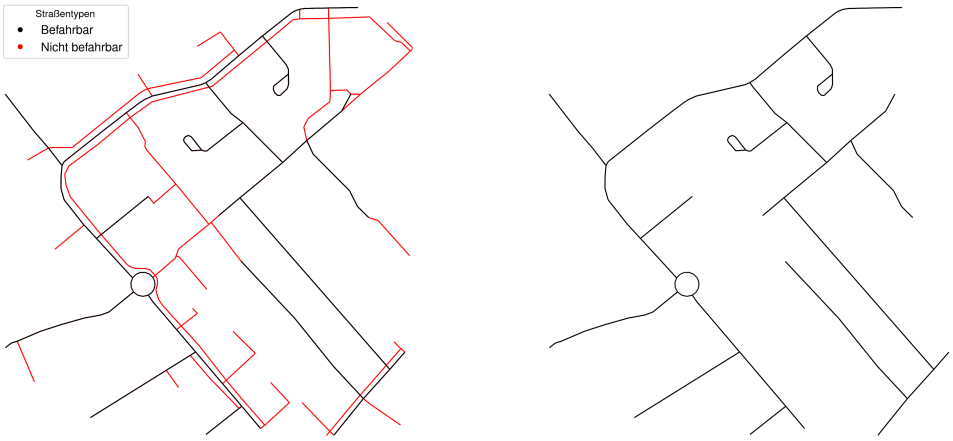
Anschließend wird der Graph vereinfacht, indem Knoten entfernt werden die nicht für die Topologie des Graphen relevant sind.

3.2 Contraction Hierarchies

3.2.1 Überblick

Straßennetzwerke weisen eine sehr hierarchische Struktur auf. Es gibt „wichtigere“ Straßen wie z. B. Autobahnen und „unwichtigere“ Straßen wie z. B. Straßen in Wohnsiedlungen (siehe Abbildung 8). Um von einem entfernten Ort zu einem anderen zu gelangen, ist es daher nur sinvoll, sich über Straßen mit zunehmender Wichtigkeit zu bewegen. Bei einer Suche könnte die Information des Straßentyps und der damit verbundenen Wichtigkeit als Heuristik verwendet werden, um bestimmte Kanten, die auf weniger wichtige Straßen führen, zu ignorieren und

[1] PBF: („Protocolbuffer Binary Format“) ist eine Alternative zum XML Format, die OSM-Rohdaten speichereffizient in Blöcken abspeichert [7]).
[2] `https://overpass-api.de/`



(a) Ungefiltertes Straßennetzwerk (b) Gefiltertes Straßennetzwerk

Abbildung 7: Straßennetzwerk gefiltert und ungefiltert. Das ungefilterte Netzwerk (a) enthält alle Straßen und Wege, die in der OSM-Datei enthalten sind. Das gefilterte Netzwerk (b) enthält nur Straßen, die für den Autoverkehr zugelassen sind.

damit die Suche zu beschleunigen. Das Problem hierbei ist allerding, dass mit dieser Methode keine Garantie besteht, dass der gefundene Weg auch wirklich der *exakt* kürzeste Weg ist. Die Methode der Contraction Hierarchies nach Geisberger et al. [12] [11] [13] löst dieses Problem, indem in einer Vorverarbeitungsphase Abkürzungskanten in den Graph eingefügt werden, die in der Suche ausgenutzt werden. Die Abkürzungen erhalten dabei die kürzesten Wege [2]. Während der Suche wird ein modifizieren bidirektioanlen Dijkstra-Algorithmus angewendet, der Kanten die zu Knoten mit niedrigerem Level führen ignoriert. Dadurch wird der Suchraum extrem verkleinert, was zu schnellen Antwortzeiten führt. Der CH-Algorithmus lässt sich in zwei Komponenten unterteilen:

1. Vorverarbeitung: In dieser Phase werden die Knoten geordnet und die Hierarchie augebaut.

2. Suche: Ausführung der bidirektionale Suche auf dem erweiterten Graph.

3.2.2 Vorverarbeitung

In dieser Phase wird der Graph $G = (V,E)$ um zusätzliche Abkürzungskanten erweitert. Die Abkürzungskanten werden im Prozess der Knotenkontraktion (eng. Node Contraction) eingefügt. Wenn ein Knoten $v \in V$ *kontraktiert* wird, dann wird er und alle Kanten die mit v inzident sind aus dem Graphen entfernt. Der Zeitpunkt des Entfernens ist gleichzeitig das *Level* des Knotens. Je später der Knoten entfernt wird, desto höher ist sein Level und damit

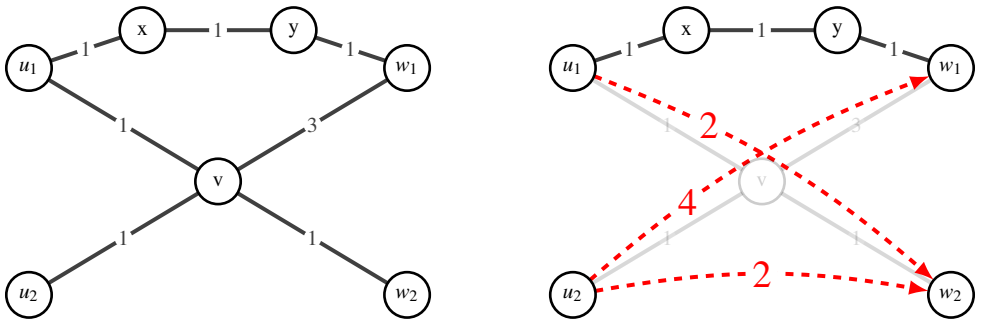
Abbildung 8: Hierarchie in Straßennetzen. Autobahnen (schwarz) sind ganz oben in der Hierarchie und sind sehr „wichtig“. Dagegen sind Straßen in Wohnsiedlungen weniger wichtig. © OpenStreetMap contributors

seine Relevanz. Wenn v auf dem kürzesten Weg zwischen zwei benachbarten Knoten u und w liegt, dann wird eine Abkürzungskante $e(u,w)$ mit $w(u,w) = w(u,v) + w(v,w)$ eingefügt, um den kürzesten Weg zu erhalten.

Abbildung 9 demonstriert den Prozess: Knoten v soll kontraktiert werden und damit er und seine inzidenten Kanten entfernt werden. Sei U die Menge aller eingehenden Kanten und W die Menge aller ausgehenden Kanten, dann muss für jedes Paar überprüft werden, ob v auf dem kürzesten Weg $<u,v,w>$ zwischen zwei benachbarten Knoten $u \in U$ mit $Level(u) > Level(v)$ und $w \in W$ mit $Level(w) > Level(v)$ liegt. Zum Beispiel ist dies zwischen u_1 und w_2 der Fall, denn es gibt keine andere Möglichkeit w_2 zu erreichen, als über v. Um den kürzesten Weg zu erhalten, wird eine Abkürzungskante $e(u_1,w_2)$ mit dem Gewicht $w(u_1,w_2) = w(u_1,v) + w(v,w_2) = 1+1 = 2$ eingefügt. Das gleiche gilt für $<u_2,v,w_1>$ und $<u_2,v,w_2>$. Der Weg von u_1 nach w_1 kann allerding über $<u_1,x,y,w_1>$ schneller erreicht werden (Kosten 3 sind kleiner als 4) und es muss daher keine Abkürzungskante eingefügt werden.

Nachdem jeder Knoten kontraktiert wurde, erhält man einen neuen Graph $G^* = (V,E')$, der als *Overlaygraphh* bezeichnet wird. E' enthält alle ursprünglichen Kanten sowie die neu hinzugefügten Abkürzungskanten (siehe Abbildung 10).

Algorithmus zur Erstellung einer CH



(a) Graph $G = (V,E)$ (b) Graph $G' = (V',E')$ mit $V' = V - \{v\}$

Abbildung 9: Beispiel einer Knotenkontraktion

Algorithm 3 Highlevel-Algorithmus der Knotenkontraktion

function NODECONTRACTION($G = (V,E)$)
  for each $v \in V$ geordnet nach Level **do**
    for each $(u,v) \in E$ mit $Level(u) > Level(v)$ **do**
      for each $(v,w) \in E$ mit $Level(w) > Level(v)$ **do**
        if <u,v,w> ist kürzester Weg von u nach w **then**
          $E = E \cup \{e(u,w)\}$ mit Gewicht $w(u,w) = w(u,v) + w(v,w)$
        end if
      end for
    end for
  end for
end function

Beweis: Kontraktion erhält kürzeste Wege

Lemma 3.1. *Sei $G = (V,E)$ ein beliebiger Graph und $G' = (V',E')$ der Graph nach Kontratkion von einem beliebigen Knoten $v \in V$ mit $V' = V - \{v\}$.*
Dann gilt für alle $s,t \in V'$: $dist_{G'}(s,t) = dist_G(s,t)$.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed euismod, nisl quis tincidunt pellentesque, nunc nisl ultrices ipsum, quis aliquam nunc nisl ut nunc. Nulla facilisi. Nulla

3.2.3 Suche

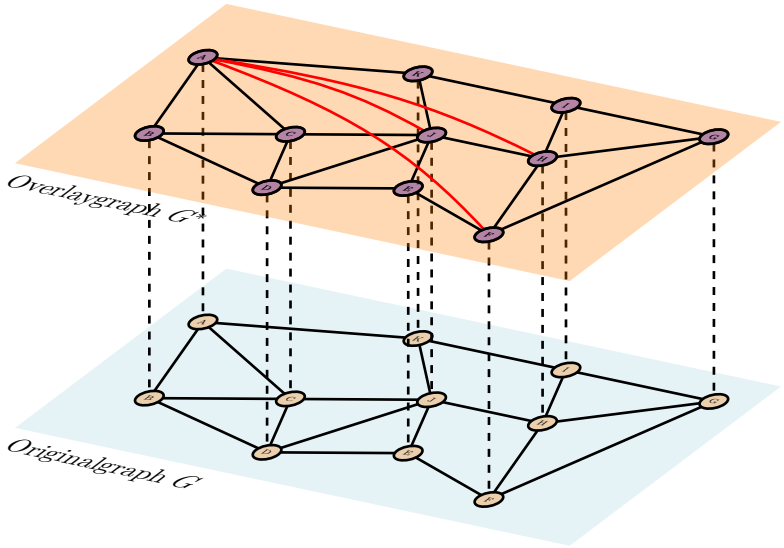


Abbildung 10: Overlaygraph nach Kontraktionsprozess. Der Graph G^* enthält alle ursprünglichen Kanten sowie die neu hinzugefügten Abkürzungskanten.

4 Schlussbetrachtung

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Literatur

[1] „9th DIMACS Implementation Challenge: Shortest Paths“. In: *International Workshop on Experimental and Efficient Algorithms*. DIMACS. 2006. URL: http://www.dis.uniroma1.it/~challenge9/.

[2] Hannah Bast u. a. *Route Planning in Transportation Networks*. URL: http://arxiv.org/pdf/1504.05140v1.

[3] Geoff Boeing. „OSMnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks“. In: *Computers, Environment and Urban Systems* 65 (2017), S. 126–139. ISSN: 0198-9715. DOI: https://doi.org/10.1016/j.compenvurbsys.2017.05.004. URL: https://www.sciencedirect.com/science/article/pii/S0198971516303970.

[4] Thomas H. Cormen u. a. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009, S. 589–592, 661–662. ISBN: 978-0-262-03384-8. URL: http://mitpress.mit.edu/books/introduction-algorithms.

[5] Daniel Delling u. a. „Engineering Route Planning Algorithms“. In: *Algorithmic of Large and Complex Networks*. Bd. 5515. Springer, 2009, S. 117–139.

[6] E. W. Dijkstra. „A note on two problems in connexion with graphs“. In: *Numerische Mathematik* 1 (1959), S. 269–271.

[7] OpenStreetMap Foundation. *PBF Format*. 2022. URL: https://wiki.openstreetmap.org/wiki/PBF_Format (besucht am 21. 07. 2023).

[8] OpenStreetMap Foundation. *About OpenStreetMap*. 2023. URL: https://www.openstreetmap.org/about (besucht am 16. 07. 2023).

[9] OpenStreetMap Foundation. *OpenStreetMap License*. 2023. URL: https://www.openstreetmap.org/copyright (besucht am 16. 07. 2023).

[10] OpenStreetMap Foundation. *Overpass API*. 2023. URL: https://wiki.openstreetmap.org/wiki/Overpass_API (besucht am 21. 07. 2023).

[11] Robert Geisberger. „Diploma Thesis - Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks“. In: 2008.

[12] Robert Geisberger u. a. „Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks“. In: *Experimental Algorithms*. Hrsg. von Catherine C. McGeoch. Springer, 2008, S. 319–333.

[13] Robert Geisberger u. a. „Exact Routing in Large Road Networks using Contraction Hierarchies“. In: *Transportation Science*. Bd. 46. 2012, S. 388–404.

[14] Peter Hart, Nils 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. ISSN: 0536-1567. DOI: 10.1109/TSSC.1968.300136.

[15] Steve Kalbink und Carol Nichols. *The Rust Programming Language*. Bd. 2. 2022.

[16] Geofabrik GmbH Karlsruhe. *Geofabrik Downloads*. 2023. URL: https://download.geofabrik.de/ (besucht am 21. 07. 2023).

[17] Thomas Liebig u. a. „Dynamic route planning with real-time traffic predictions“. In: *Information Systems* 64 (2017), S. 258–265. ISSN: 0306-4379. DOI: https://doi.org/10.1016/j.is.2016.01.007. URL: https://www.sciencedirect.com/science/article/pii/S0306437916000181.

[18] Dorothea Wagner und Thomas Willhalm. „Speed-Up Techniques for Shortest-Path Computations“. In: *STACS 2007*. Hrsg. von Wolfgang Thomas und Pascal Weil. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, S. 23–36. ISBN: 978-3-540-70918-3.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen Hilfsmittel als die angegebenen verwendet habe.

München, den 23. Juli 2023