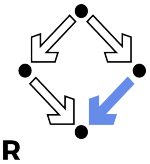
RISC



RESEARCH INSTITUTE FOR
SYMBOLIC COMPUTATION

JOHANNES KEPLER
UNIVERSITY LINZ

# Highway Node Routing

Christian Huber

July 2024

**RISC Report Series No. 24-08**

ISSN: 2791-4267 (online)

Available at https://doi.org/10.35011/risc.24-08





**JOHANNES KEPLER**
**UNIVERSITY LINZ**
Altenberger Str. 69
4040 Linz, Austria
www.jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITÄT LINZ

Eingereicht von
**Christian Huber**

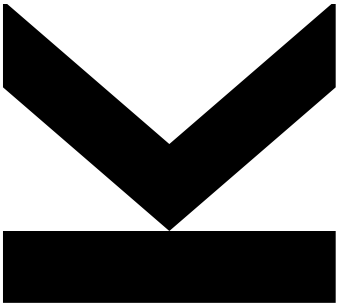
Angefertigt am
**Research Institute for Symbolic Computation**

Beurteiler
**Univ.-Prof. Dr. Carsten Schneider**

Mitbetreuung
**Dr. Karoly Bosa (RISC Software GmbH)**
**Karl-Heinz Kastner, MSc (RISC Software GmbH)**

07 2024

# Highway Node Routing



Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Science

im Bachelorstudium

Computer Science

**JOHANNES KEPLER**
**UNIVERSITÄT LINZ**
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

# Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Bachelorarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Ort, Datum

Unterschrift

# Danksagung

Mein Dank gilt meinem Betreuer Carsten Schneider, der mich mit anregenden Diskussionen und seiner Unterstützung während des gesamten Projekts maßgeblich beeinflusst hat. Ein herzlicher Dank gebührt der RISC Software GmbH, ohne deren Beitrag diese Arbeit nicht realisierbar gewesen wäre, insbesondere für die Zurverfügungstellung des Frameworks und der Testdaten. Ich schätze die umfangreiche Hilfe von Karl-Heinz Kastner und Karoly Bosa, die stets bereit waren, technische Fragen zu beantworten.

# Zusammenfassung

Ein Routingserver steht und fällt mit dem dahinterliegenden Routing-Algorithmus. In der heutigen Zeit, wo Wartezeiten immer weniger toleriert werden, ist es umso wichtiger, die Antwortzeit so kurz wie möglich zu halten. Denn wenn die Abfrage zu lange dauert, werden die Nutzer eher den Anbieter wechseln, als auf das Ergebnis zu warten. Je mehr Abfragen das System zu bearbeiten hat, desto gravierender wird das Problem. Als Lösung kommt einem schnell der Dijkstra-Algorithmus in den Sinn. Doch bei größeren Verkehrsgraphen stößt dieser an seine Grenzen, wie wir später sehen werden. Hier ist eine ausgefeiltere Lösung erforderlich: das von D. Schultes eingeführte Highway-Node-Routing-Verfahren. Wir werden uns dieses Schritt für Schritt ansehen und prüfen, ob es unser Problem entsprechend löst.

Die Idee hinter diesem Verfahren ist es, den Verkehrsgraphen um eine Hierarchie zu erweitern. Jede Schicht ist eine Teilmenge der unteren Schichten, dabei bleibt die Eigenschaft der optimalen Route erhalten. Bei der Abfrage wechseln wir Schritt für Schritt in eine höhere Schicht, wobei immer weniger Knoten und Kanten berücksichtigt werden müssen, wodurch eine signifikante Beschleunigung erzielt wird. Diese Idee geht aus dem Verkehrsnetz hervor. Angenommen, in der untersten Schicht befinden sich alle Straßen. In der ersten Schicht liegen Landes-, Bundesstraßen und Autobahnen. In der zweiten Bundesstraßen und Autobahnen und in der letzten nur Autobahnen. Das Routing in solch einer Hierarchie liefert zwar nicht mehr das richtige Ergebnis, aber sie repräsentiert die Grundidee des Algorithmus. In dieser Bachelorarbeit werden wir uns ansehen, wie wir eine korrekte Hierarchie berechnen können und welche Vor- und Nachteile dieser Algorithmus mit sich bringt.

# Abstract

A routing server stands and falls with the routing algorithm behind it. In today's world, where waiting times are less and less tolerated, it is all the more important to keep the response time as short as possible. After all, if the query takes too long, users are more likely to switch providers than wait for the result. The more queries the system has to process, the more serious the problem becomes. The Dijkstra algorithm quickly comes to mind as a solution. However, this reaches its limits with larger traffic graphs, as we will see later. A more sophisticated solution is required here: the highway node routing method introduced by D. Schultes. We will look at this approach step by step and check whether it satisfies the requirements of our problem.

The idea behind this method is to add a hierarchy to the traffic graph. Each layer is a subset of the lower layers, maintaining the property of the optimal route. During the query, we move step by step to a higher layer, whereby fewer and fewer nodes and edges have to be taken into account, thereby achieving a significant acceleration. This idea emerges from the traffic network. Assume that all roads are in the lowest layer. The first layer contains state and federal roads and motorways. The second layer contains federal roads and motorways and the last layer contains only motorways. Routing in such a hierarchy no longer provides the correct result, but it represents the basic idea of the algorithm. In this bachelor thesis we will look at how we can calculate a correct hierarchy and elaborate the advantages and disadvantages of this approach.

# Definitionen

| | |
|---|---|
| $x_l$ | Der Index l beschreibt immer das Level oder die Hierarchiestufe. $x$ ist nur ein Platzhalter |
| $G_0 = (V_0, E_0)$ | Beschreibt den Ursprungsgraphen |
| $G_l$ | Beschreibt den Graphen der Schicht l |
| $V_l$ | Knotenmenge von $G_l$ |
| $E_l$ | Kantenmenge von $G_l$ |
| $G'_l = (V'_l, E'_l)$ | Beschreibt den Kern von $G_l$ |
| $V'_l$ | Kernknotenmenge von $G_l$ |
| $E'_l$ | Kernkantenmenge von $G_l$ |
| $rk_u(v)$ | Dijkstrarang. Gibt die Ordnung an, in welcher die Elemente aus der Queue während einer Dijkstrasuche entfernt werden. Dabei hat der Startknoten Rang 0, der nächste Nachbar Rang 1 und so weiter |
| $u, v \in V_l : \ d_l(u, v)$ | Kürzeste Distanz im Vorwärtsgraphen von $u$ zu $v$ |
| $u, v \in V_l : \ \ d_l^{\leftrightarrow}(u, v)$ | Kürzeste Distanz im ungerichteten Graphen von $u$ zu $v$ |
| $P\langle s, \ldots, p\rangle : \ s \prec p$ | Die Relation beschreibt, dass $s$ ein Vorgänger von $p$ auf den Pfad $P$ ist. Es muss nicht der direkte Vorgänger sein |
| $B$ | Beschreibt den Suchbaum einer Dijkstrasuche |
| $\gamma(x)$ | Beschreibt die Elternmenge von $x$ in $B$ |
| $\nu(u)$ | Beschreibt die Kindermenge von $x$ in $B$ |

# 1 Einleitung

## 1.1 Motivation

Die RISC Software GmbH betreibt derzeit einen Routing-Server, der im Hintergrund einen Dijkstra-Algorithmus implementiert hat. Dabei wurde ein Problem beobachtet. Es entstehen lange Wartezeiten, wenn viele Abfragen in einem kurzen Zeitraum eintreffen. Die Ursache für dieses Verhalten liegt in der Berechnungszeit des Dijkstra-Algorithmus [Dij59]. In der ursprünglichen Implementierung läuft dieser mit $n$ Knoten in $O(n^2)$ Zeit (die Knoten Definition ist in Kapitel 6 zu finden). Aus diesem Grund ist die Abfrage für größere $n$ sehr langsam, was bei einem Verkehrsgraphen fast immer der Fall ist. Somit stauen sich die Abfragen und es kommt zu ungewollt langen Wartezeiten. In der schnelllebigen Welt von heute fällt dies umso mehr negativ auf.

## 1.2 Ansatz

Das Problem war schon länger bekannt und es wurde bereits zahlreiche Recherchearbeit hineingesteckt. Was das Konzept der Highway Hierarchien aufbrachte, aber von der RISC Software GmbH nie umgestezt wurde. Dabei bezieht sich meine folgende Information zu dem Thema auf Dominik Schultes Dissertation 'Route Planning in Road Networks' [Sch08]. Genauer gesagt wurde daraus das Highway Node Routing implementiert, welches auf dem Konzept der Highway Hierarchien aufsetzt.

Die Strategie dahinter ist, den Graphen Schritt für Schritt zu verkleinern, indem die wichtigen Kanten herausextrahiert werden. Die Idee geht aus der Struktur unseres Verkehrsnetzes hervor. In der untersten Schicht befinden sich alle Straßen. In der nächsten befinden sich nur Landesstraßen und höher wertige. Dies wird fortgesetzt, bis sich in der letzten Schicht nur mehr Autobahnen befinden. Bei der Abfrage werden dann nur die Kanten und Knoten berücksichtigt, von der aktuellen Hierarchie Stufe. Begonnen wird in der untersten Schicht und sobald ein Knoten aus einer höheren erreicht wurde, wird in diese gewechselt. Die wirkliche Implementierung weicht von jener ab, aber es ist eine gute Veranschaulichung von dem Konzept. Das Problem an dieser Idee ist, dass nicht gewährleistet werden kann, dass die optimale Route sich nicht verändert, wenn alle Kanten der unteren Schichten vernachlässigt werden.

Der korrekte Ansatz: Genau wie in der obigen Intuition erzeugen wir eine Hierarchie nur mit dem Unterschied, dass die Kanten Auswahl für die jeweilige Schicht nicht von der Art der Straße abhängt (Landesstraße, Bundesstraße usw.), sondern berechnet wird. Die Vorberechnug besteht aus mehreren Phasen. In der ersten bauen wir eine Hierarchie auf. Um dies zu ermöglichen, extrahieren wir Schritt für Schritt die wichtigen Knoten und Kanten aus dem Graphen. In der nächsten Phase betrachten wir nur die Knotenmenge der jeweiligen Schicht. Nehmen wir an, wir wollen zwischen diesen Knoten die optimale Route berechnen. Hierfür gibt es mehrere Methoden. Eine davon ist die Verwendung eines Überlagerungsgraphen. In einem Überlagerungsgraphen bleibt die Eigenschaft der optimalen Routen zwischen den Knoten erhalten. Folglich ist das Routing in diesem Graphen viel schneller, da weniger Kanten und Knoten berücksichtigt werden müssen. Was das Ziel dieser Arbeit ist.

In den nächsten Abschnitten gehen wir auf die oben beschrieben Phasen näher ein und erläutern, warum diese Vorberechnung korrekt ist und die gewünschte Beschleunigung liefert.

Der Dijkstra-Algorithmus wird in Kapitel 2 näher analysiert. Für die weitere Arbeit ist es wichtig, diesen Algorithmus zu verstehen. Zu diesem Zweck werden wir uns ansehen, wie er funktioniert und seine Korrektheit beweisen.

Die Vorberechnung wird in Kapitel 3 ausführlich behandelt. Sie ist der wichtigste Teil der Arbeit, denn nur durch die Berechnung der neuen Struktur ist es möglich, die gewünschte Laufzeitbeschleunigung zu erreichen. Zur besseren Verständlichkeit wird die Vorberechnung in mehrere Teile aufgeteilt und separat besprochen.

Das Highway Node Routing besteht nicht nur aus der Vorberechnung, sondern auch aus der Abfrage. Diese wird in Kapitel 4 betrachtet.

Die Korrektheit des Algorithmus wird in Kapitel 5 behandelt. In den vorherigen Kapiteln wurde nur die Funktionalität besprochen und die Korrektheit ausgelassen. Der Algorithmus muss jedoch nicht nur schnell, sondern auch korrekt sein, um das Problem zu lösen.

Das Framework was für die Implementierung verwendet wurde, wird in Kapitel 6 kurz erläutert.

In der Implementierung musste auf ein paar Sonderheiten Rücksicht genommen werden, welche in Kapitel 7 behandelt werden.

Mögliche Verbesserungen, die die Laufzeit oder die Vorberechnungszeit weiter reduzieren und die in Zukunft aufgegriffen und umgesetzt werden können, werden in Kapitel 8 diskutiert.

Ob sich die Implementierung des Highway Node Routing lohnt, wird in Kapitel 9 behandelt. Hier werden verschiedene Versionen des Highway Node Routings dem Dijkstra-Algorithmus gegenübergestellt. Dabei wird die Auswirkung der Parameterwahl auf die Laufzeit betrachtet.

# 2 Dijkstra

In dieser Arbeit wird oft auf den Dijkstra-Algorithmus [Dij59] verwiesen, weshalb es wichtig ist, diesen zu verstehen. Der Algorithmus findet in einem gewichteten Graphen $G = (V, E)$ mit nicht negativen Gewichten $\forall(u, v) \in E : w((u, v)) \geq 0$ immer die kürzeste Route zwischen zwei Knoten, falls eine existiert. Während der Suche erzeugt er einen Baum $B$, welcher die kürzesten Pfade vom Ursprung $s$ zu all seinen Knoten enthält. Für die Suche benötigen wir einen Priorityqueue. Diese ordnet die Knoten gemäß einer Relation. In unserem Fall werden die Knoten anhand ihrer Distanz zum Ursprung geordnet, wobei ein Knoten mit der kleinsten Distanz ganz vorne in der Queue eingereiht ist. Zu Beginn der Suche sind alle Knoten unerreicht und ihre Distanz zum Ursprung unendlich. Wir fügen den Ursprungsknoten $s$ mit einer Distanz von null in die Queue ein. Die folgenden Schritte werden so lange ausgeführt, bis die Queue leer oder der gesuchte Endknoten $t$ abgearbeitet ist.

1. Wir entfernen den Knoten $u$ mit der kleinsten vorläufigen Distanz $\delta(u)$ aus der Priorityqueue. $u$ ist jetzt abgearbeitet und die kürzeste Distanz $d(s, u)$ wurde gefunden.

2. Wir betrachten alle Kanten $(u, v)$ von $u$. Dabei wird zwischen folgenden Fällen unterschieden:

   - $v$ ist unerreicht. Wir fügen $v$ in die Priorityqueue mit einer vorläufigen Distanz $\delta(s, v) = d(s, u) + w((u, v))$ ein.
   - $v$ ist erreicht. Wir überprüfen, ob der neue Pfad über $u$ ein kürzerer ist als der bisher gefundene. Dazu prüfen wir ob, $\delta(s, v) > d(s, u) + w((u, v))$ gilt. Wenn ja, setzten wir den Key von $v$ auf $d(s, u) + w((u, v))$ in der Queue und legen $u$ als neuen Elternknoten von $v$ fest.
   - $v$ ist abgearbeitet. Wir betrachten die Kante $(u, v)$ nicht, da die kürzeste Distanz $d(s, v)$ bereits gefunden wurde.

Wenn die Suche endet, wurde der kürzeste Pfad $P$ von $s$ nach $t$ gefunden, oder es gibt keinen solchen Pfad.

In der Motivation 1.1 wurde angedeutet, dass die Zeitkomplexität des Dijkstra-Algorithmus $O(n^2)$ mit $|V| = n$ beträgt. Die Komplexität setzt sich aus der Knotenmenge und des Knotengrads zusammen. Im schlechtesten Fall haben wir einen vollständig verbundenen Graphen. Der Algorithmus durchläuft alle Knoten und für jeden müssen wir alle anderen Knoten berücksichtigen, da zu jedem Knoten eine Kante führt $\rightarrow O(n^2)$

**Algorithmus 1 :** Psudo-Code Dijkstra

**Input :** Graph $G = (V, E)$ mit Gewichtsfunktion $w : E \to \mathbb{R}^{\geq 0}$, Startknoten $s$
**Output :** Kürzeste Distanz von $s$ zu allen Knoten in $V$

```
Initialisiere dist[u] ← ∞ für alle u ∈ V;
dist[s] ← 0;
Initialisiere priority queue Q;
Q ← V;
while Q ist nicht leer do
    u ← Extract-Min(Q);
    if dist[u] = ∞ then
        break;
    for für jeden nicht abgearbeiteten Nachbar v von u do
        if dist[u] + w(u, v) < dist[v] then
            dist[v] ← dist[u] + w(u, v);
            Decrease-Key(Q, v, dist[v]);
return dist;
```

Im Algorithmus beschreibt $dist[u]$ die Distanz vom Startknoten $s$ zu $u$ ($dist[u] = \delta(s, u)$), wenn der Knoten abgearbeitet ist, gleicht sie $d(s, u)$.

## 2.1 Korrektheitsbeweis

Wir wollen feststellen, ob der Dijkstra-Algorithmus korrekt ist. Dafür müssen wir prüfen, ob ein abgearbeiteter Knoten wirklich auf den kürzesten Pfad erreicht wurde. Wir behaupten, dass jeder Knoten der zum Baum $B$ hinzugefügt wird, auf den kürzesten Pfad erreicht wurde. Es muss somit zum Zeitpunkt des Hinzufügens die vorläufige kürzeste Distanz $\delta(s, w)$ gleich der wirklichen kürzesten Distanz $d(s, w)$ entsprechen. Dies gilt unter der Annahme, dass alle Kantengewichte größer als null sind: $\forall(u, v) \in E : w((u, v)) \geq 0$.

Im folgenden Beweis wird die Tatsache benötigt, dass ein Subpath von einem kürzesten Pfad auch ein kürzester Pfad ist; siehe Lemma 1 in [DI04]. Die Gültigkeit kann man leicht nachvollziehen. Betrachten wir dazu einen beliebigen kürzesten Pfad von $x$ nach $p$: $P\langle x, u, \ldots, v, p\rangle$. Würde das Lemma nicht gelten, kann es einen Subpath $P_{uv}$ von $u$ nach $v$ geben, der kürzer ist. Dann wäre der Pfad $P^*\langle x, P_{uv}, p\rangle$ kürzer als der oben angenommene.

Gegeben ist ein Graph $G = (V, E)$ mit Startknoten $s$ und Gewichtsfunktion $w : E \to \mathbb{R}^{\geq 0}$, welche die Kanten nach den nicht negativen reellen Zahlen abbildet. Das Ziel ist es, die Distanz $d(s, u)$ vom Startknoten $s$ zu allen Knoten in $V$ zu berechnen. Das heißt, am Ende beinhaltet der Baum $B$ alle von $s$ erreichbaren Knoten, welche über den kürzesten Pfad erreicht wurden. Der Beweis wird mit Induktion über den Dijkstra-Rang bewiesen, welcher mit folgenden Annahmen gezeigt wird.

Annahmen:

A1: $\forall v \in B : d(s, v)$. Für alle Knoten in $B$ gilt, dass diese auf den kürzesten Pfad erreicht wurden und dieser in $B$ existiert.

A2: $D = \{w \mid (v, w) \in E \wedge v \in B\} \setminus B$, $\forall w \in D : \delta(s, w)$. Die Menge $D$ beschreibt alle Knoten, um welche $B$ erweitert werden kann. Somit alle Knoten, die direkt von $B$ erreichbar sind. Für diese Knoten gilt, dass sie über den kürzestmöglichen Pfad erreicht wurden, nur unter der Verwendung von $B$.

*Induktionsanfang* : Der Baum beinhaltet nur den Ursprungsknoten $s$ mit der Distanz null $\to B = \{s\}$, $d(s, s) = 0$. Die Distanz zu allen anderen Knoten ist $\infty$. Die Behauptung gilt, da die kürzeste Distanz zu sich selber null ist. $rk_s(s) = 0$.

*Induktionsannahme* : Wähle $B$, sodass A1 und A2 erfüllt sind.

*Induktionsschritt* : $rk_s = n \to rk_s = n + 1$. Nach unserer Behauptung muss

$$\delta(s, w) = d(s, w)$$

gelten für jeden weiteren Knoten, der zum Baum hinzugefügt wird (A1). Wir zeigen dies durch einen Widerspruch. Nehmen wir an, A1 gilt nicht für den nächsten Knoten, der zum Baum hinzugefügt wird. Für diesen Knoten gilt

$$\delta(s, w) > d(s, w).$$

Der neue Baum $B' = B \cup \{w\}$ erfüllt A1 nicht mehr. Für alle Knoten im Baum $B$ gilt $\forall v \in B : \delta(s, v) = d(s, v)$.

Wenn $w$ über einen suboptimalen Pfad $P$ erreicht wird, muss es einen kürzeren Pfad $P^*$ geben, der Knoten beinhaltet, die noch nicht im Baum enthalten sind. Dies hat den Hintergrund, da jeder Knoten in $B$ über den kürzesten Pfad erreicht wurde. Ebenfalls wurden alle Kanten berücksichtigt. In Folge dessen haben wir alle möglichen Pfade zu $w$ in $B$ gefunden und wir nehmen den kleinsten davon. Somit gibt es in $B$ keinen besseren Pfad (A2). Zudem wird immer der Knoten aus der Queue entfernt, mit dem kleinstem Gewicht. Der optimale Pfad $P^*$ muss in $B$ starten, den der Startknoten ist in $B$ enthalten. Nehmen wir an, $u$ ist der erste Knoten, der außerhalb von $B$ und auf $P^*$ liegt ($u \neq w$, $u \notin B$, $u \in D$, $u \in P^*$). $v$ ist der direkte Vorgänger von $u$ in $P^*$ ($v \prec_{P^*} u$, $v \in B$).

Wir wissen, dass $\delta(s, v) = d(s, v)$ gilt, denn $v$ wurde vor $w$ zu $B$ hinzugefügt und $w$ ist der erste suboptimale Knoten. Wir wissen ebenfalls, dass die vorläufige Distanz zu $u$

$$\delta(s, u) = \delta(s, v) + w((v, u))$$
$$\delta(s, u) = d(s, v) + w((v, u))$$

ist. Dies gilt, da $u$ bereits erreicht wurde. $v$ ist in $B$ enthalten und dadurch wurden alle Kanten von $v$ entspannt, auch $w((v, u))$. Wir wissen zusätzlich, dass diese Distanz größer gleich als jene von $w$ sein muss, denn $w$ wird als nächstes entfernt.

$$\delta(s, w) \leq \delta(s, u). \tag{2.1}$$

Weil ein Subpath von einen kürzesten Pfad auch ein kürzester Pfad ist, gilt:

$$\delta(s, u) = d(s, u). \tag{2.2}$$

Weiters gilt auch:

$$w(P^*) = d(s, w) = d(s, u) + d(u, w). \tag{2.3}$$

Dadurch folgt:

$$\begin{aligned} \delta(s, w) &\overset{(2.1)}{\leq} \delta(s, u) \\ &\overset{(2.2)}{\leq} d(s, u) \\ &\overset{(2.3)}{\leq} \underbrace{d(s, u) + d(u, w)}_{d(s,w)} \end{aligned}$$

d.h.

$$\delta(s, w) \leq d(s, w).$$

$\delta(s, w)$ kann nicht kleiner als $d(s, w)$ sein, somit gilt

$$\delta(s, w) = d(s, w)$$

was im Widerspruch zur Annahme $\delta(s, w) > d(s, w)$ steht. Somit muss $\delta(s, w) = d(s, w)$ gelten für jeden Knoten, der zu $B$ hinzugefügt wird. A1 gilt auch für $B'$.

Es bleibt noch zu zeigen, ob A2 auch für $B'$ gilt. Das heißt, alle Nachbarn von $w$, die nicht in $B'$ enthalten sind, müssen über den kürzestmöglichen Pfad in $B'$ erreicht werden. Es gibt zwei Fälle:

a) Der Nachbarknoten wurde bereits erreicht.

b) Der Nachbarknoten wurde noch nicht erreicht.

Dieser Code Abschnitt ist für die nächsten Behauptungen relevant:

```
for für jeden nicht abgearbeiteten Nachbar v von u do
    if dist[u] + w(u, v) < dist[v] then
        dist[v] ← dist[u] + w(u, v);
        Decrease-Key(Q, v, dist[v]);
```

a) Wir müssen wieder zwischen zwei Fälle unterscheiden:

1. $d(s, w) + w((w, v)) \geq \delta(s, v)$

   Der neu gefundene Pfad ist größer als der bisher kürzeste. Betrachten wir die If-Bedingung. In diesem Fall unternehmen wir nichts. Was auch richtig ist, denn der neue Pfad ist größer als der aktuelle. A2 bleibt erhalten.

2. $d(s, w) + w((w, v)) < \delta(s, v)$

   Der neue Pfad über $w$ zu $v$ ist kürzer als der bisherige. Der aktuelle ist der kürzestmögliche in $B$ wegen A2. Die If-Bedingung ist erfüllt, wir reduzieren den Key von $v$. A2 bleibt erhalten.

b) Hier brauchen wir nur einen Fall betrachten, denn das Gewicht eines unerreichten Knotens $v$ ist unendlich $\rightarrow$ die If-Bedingung ist immer erfüllt. Wir reduzieren den Key von $v$. Das ist der kürzeste Pfad in $B'$. Durch A1 und A2 kann es in $B$ keinen kürzeren Pfad geben. A2 bleibt erhalten.

In allen Fällen bleibt A2 erhalten. A2 gilt für $B'$.

Termination:
Durch die While Loop iterieren wir so lange, bis die Queue leer ist. Die Queue beinhaltet am Anfang alle Knoten $\rightarrow B = V$. Korrekt durch A1.

So bald ein Knoten mit der Distanz $\infty$ aus der Queue entfernt wird, wird abgebrochen. In diesem Fall gibt es keinen Knoten mehr, um B zu erweitern (Alle Knoten in der Queue haben eine Distanz von $\infty$). Korrekt durch A2.

# 3 Vorberechnung

In diesem Abschnitt werden Schritt für Schritt die verschiedenen Algorithmen und Phasen erklärt, die für die Erzeugung der Hierarchie notwendig sind. Dabei basiert die folgende Information auf Dominik Schultes Dissertation [Sch08].

Kurzer Überblick:

- Kern 3.1 [Sch08, S. 56–57]
- Nachbarschaftsradius 3.2 [Sch08, S. 49]
- Highway Hierarchie 3.3 [Sch08, S. 49–56]
- Überlagerungsgraph 3.4 [Sch08, S. 93–102]

Die Vorberechnung für $L$ Schichten wird folgendermaßen durchgeführt:

$l = 1$
while $l < L$:

1. Berechnung des Kerns
2. Berechnung des Nachbarschaftsradius
3. Berechnung der Highway-Hierarchie
4. $l{+}{+}$

Der Überlagerungsgrpah wird anschließend mit der Knotenmenge der jeweiligen Schicht berechnet.

## 3.1 Kern

Ein Verkehrsnetz beinhaltet viele Knoten mit einem niedrigen Grad. In Abbildung 3.1 ist dies auch an unseren Graphen ersichtlich. Die Idee ist, alle Knoten zu überspringen, welche einen geringen Grad besitzen. Wir erhoffen uns dadurch einige Vorteile. Die Vorberechnung sowie die Abfragen werden schneller, da die überbrückten Knoten nicht mehr berücksichtigt werden müssen. Des weiterem sorgt es später für eine schnellere/bessere Kontraktion der Hierarchie.

Die Auswahl, welche Knoten überbrückt werden sollen, muss vorsichtig getroffen werden. Durch die Überbrückung mit zusätzlichen Kanten steigt der durchschnittliche Grad der Knoten. Dies bedeutet im Normalfall, dass bei einer Abfrage mehr Kanten berücksichtigt werden müssen, was den Vorteil zunichte macht. Um das zu vermeiden, brauchen wir ein Kriterium, welches nur Knoten überbrückt, wo es mehr Vorteile als Nachteile bringt.

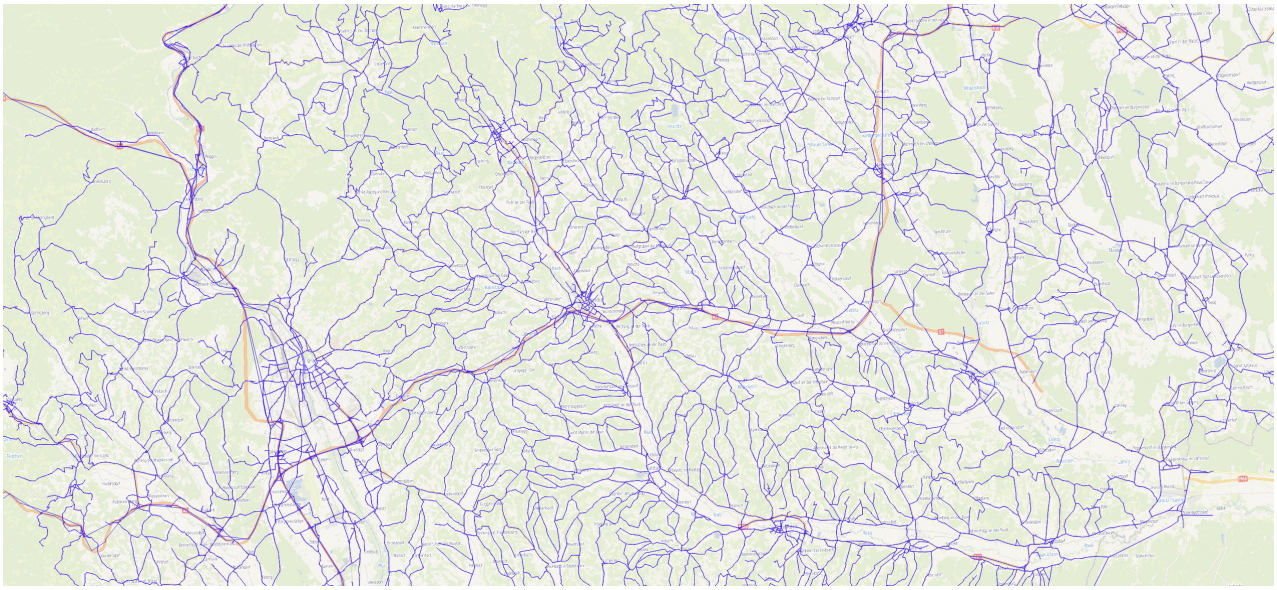


**Abbildung 3.1:** Die Abbildung zeigt einen Ausschnitt aus dem Ursprungsgraphen. Die Straßen sind sehr detailreich, was für das Routing nicht relevant ist. Viele können durch eine gerade Linie ersetzt werden.

### 3.1.1 Überbrückungskriterium

$$shortcuts \leq c \cdot (deg_{in}(u) + deg_{out}(u))$$

- *shortcuts* steht für die Anzahl an Shortcuts, die nötig wären, um den Knoten zu überbrücken.
  - Die Anzahl an Shortcuts ist der Grad der Eingangskanten mal dem Grad der Ausgangskanten. Dabei werden die shortcuts, die in einem Loop enden würden, abgezogen.
- $c$ ist ein fixierter Hyperparameter, welcher die Stärke der Komprimierung festlegt.
- $deg_{in}(u)$ steht für den Grad der Eingangskanten
- $deg_{out}(u)$ steht für den Grad der Ausgangskanten

Das Kriterium vergleicht die Anzahl an shortcuts mit den Eingangs- und Ausgangsgrad von u. Desto höher c gewählt wird, desto mehr Knoten werden überbrückt.

### 3.1.2 Algorithmus

**Algorithmus 2 :** Psudo-Code Kern

**Input :** Graph $G_{l-1} = (V_{l-1}, E_{l-1})$, $c > 0$
**Output :** Graph $G'_l = (V'_l, E'_l)$

```
G'_l ← Copy(G_{l-1});
Initialisiere Stack S;
S ← V_{l-1};
while S ist nicht leer do
    u ← Pop(S);
    if shortcuts ≤ c · (deg_in(u) + deg_out(u)) then
        create-shortcuts(u) ;                    // fügt die shortcuts zu G'_l hinzu
        remove(u) ;                              // entfernt u und all seine Kanten von G'_l
        for v Nachbar von u ∧ v ∉ S ∧ v ist nicht überbrückt do
            if shortcuts ≤ c · (deg_in(v) + deg_out(v)) then
                Push(S, v);
return G';
```

Zum Start werden alle Knoten einem Stack hinzugefügt. Der Stack wird Knoten für Knoten abgearbeitet. Für jeden Knoten $u \in V_l$ überprüfen wir das Überbrückungskriterium. Wenn es erfüllt wird, werden die erforderlichen shortcuts erzeugt und $u$ zur Überbrückungsmenge $B_l$ hinzugefügt. Dabei ist zu beachten, dass sich der Grad der anliegenden Knoten durch die shortcuts ändert. Dadurch kann ein Knoten, der zuerst nicht überbrückbar war, es jetzt sein. Folge dessen werden alle anliegende Knoten, die bereits vom Stack genommen und nicht überbrückt wurden und jetzt überbrückbar sind, wieder zum Stack hinzugefügt.

Die resultierende Kern Knoten Menge $V'_l$ beinhaltet alle Knoten die nicht überbrückt wurden, $V'_l = V_{l-1}/B_l$ . Die Kern Kanten Menge $E'_l$ beinhalten alle Kanten von $E_{l-1}$ die zu den Kern Knoten gehören und alle shortcut Kanten, $E'_l = (E_{l-1} \cap (V'_l \times V'_l)) \cup S_l$. Wobei $S_l$ alle shortcut Kanten beinhaltet. Der Graph $G'_l = (V'_l, E'_l)$ beschreibt den Kern von $G_l$.

### 3.1.3 Erweiterung

Für ein größeres $c$ wird der Graph am Anfang stark komprimiert. Aufgrund dessen ist die erste Schicht sehr viel kleiner als der Ursprungsgraph. Das wirkt sich negativ auf die Abfrage aus, denn die Suche wechselt erst später in die höhere Schicht. Um dieses Verhalten zu unterdrücken, implementieren wir einen Sprungzähler. Der Zähler gibt an, wie viele Knoten in Folge überbrückt werden dürfen. Wenn dieser erreicht ist, wird der Knoten nicht mehr übersprungen.

## 3.2 Nachbarschaftsradius

Mit dem Nachbarschaftsradius wollen wir eine Art Lokalität definieren. Der Nachbarschaftsradius $r_l(u)$ mit $u \in V_l'$ wird für die Berechnung der Highway Hierarchie benötigt und kann über den Hyperparameter H variiert werden. $H$ gibt die Anzahl an Knoten an, die zur Nachbarschaft gehören. Das bedeutet, dass die kürzeste Distanz von $u$ zu diesen Knoten nicht den Nachbarschaftsradius überschreitet. Es ist sehr schwer bis unmöglich, einen allgemein passenden Nachbarschaftsradius festzulegen, wegen der starken Unterschiede in der Knotendichte von Stadt- zu Landgebieten. Deshalb wird der Radius für jeden Knoten einzeln berechnet.

In der Definition wird zwischen Vorwärts- und Rückwärtsradius unterschieden, für den Vorwärts- beziehungsweise den Rückwärtsgraphen. Dominik Schultes stellte fest, dass es fast keinen Unterschied macht, ob zwischen Vorwärts- ($r_l^{\rightarrow}(u)$) und Rückwärtsradius ($r_l^{\leftarrow}(u)$) unterschieden wird oder nicht. Aus diesem Grund wird nur ein Radius für beide Richtungen verwendet. Dadurch sparen wir Speicher und Vorberechnungszeit ein. Somit kann die Berechnung auch im ungerichteten Graphen stattfinden. Das heißt, jede Kante die gerichtet ist, wird als ungerichtet betrachtet, auch wenn die Kante im ursprünglichen Graphen nicht existiert.

### 3.2.1 Algorithmus

**Algorithmus 3 :** Psudo-Code Nachbarschaftsradius

**Input :** Graph $G_l' = (V_l', E_l')$, Hyperparameter $H \in \mathbb{N}$
**Output :** Nachbarschaftsradius für jeden Knoten $u \in V_l'$

```
Initialisiere Stack S;
S ← V_l';
while S ≠ ∅ do
    u ← Pop(S);
    d ← calculate-Dijkstra-H(u);
    r_l(u) ← d
```

wobei calculate-Dijkstra-H($u$) die Distanz im ungerichteten Graphen zu dem Knoten mit dem Dijkstra-Rank H zurückgibt.

Die Berechnung erfolgt im Graph $G_l' = (V_l', E_l')$. Für jeden Knoten $u \in V_l'$ und gegebenes fixes $H \in \mathbb{N}$, setzen wir den Nachbarschaftsradius vom Knoten $u$ auf die Distanz $d_l^{\leftrightarrow}(u, v)$, wobei $v \in V_l'$ und $rk_u(v) = H$ (Dijkstra-Rank von $v$ bezogen auf $u$) gilt. Der Dijkstra-Rank bezieht sich ebenfalls auf den ungerichteten Graphen. $s \in \mathcal{N}_l(u)$ bedeutet, $s$ gehört zur Nachbarschaft von $u$.

$$r_l(u) = r_l^{\rightarrow}(u) = r_l^{\leftarrow}(u) = d_l^{\leftrightarrow}(u, v).$$

In Abbildung 3.2 ist der Nachbarschaftsradius visualisiert. Alle Knoten in dem blauen Kreis gehören zu $N_l(u)$. $v$ besitzt den Dijkstra-Rank H, somit ist $r_l(u) = d^{\leftrightarrow}(u, v)$.

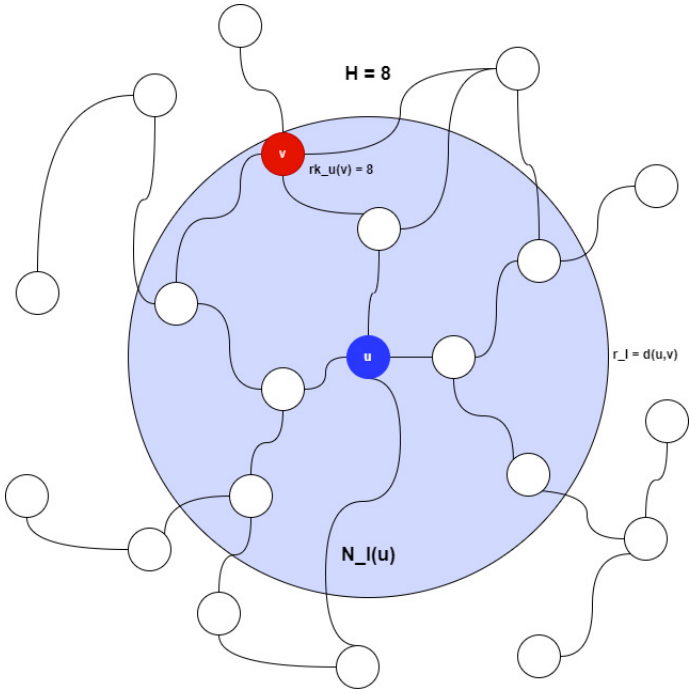



**Abbildung 3.2:** Diese Abbildung zeigt den Nachbarschaftsradius für H=8.

## 3.3 Highway Hierarchie

Die Grundidee hinter den folgenden Schritten ist, den Graphen zu minimieren, um bei der Abfrage weniger Knoten und Kanten berücksichtigen zu müssen. Es ist offensichtlich, dass nicht irgendwelche Kanten weggelassen werden dürfen, denn dadurch ist es höchst wahrscheinlich nicht mehr möglich, die optimale Route zu berechnen. Ein Vorschlag wäre, die bestehende Hierarchie des Verkehrsnetzes zu nutzen. Es gibt bereits Landes-, Bundesstraßen oder auch Autobahnen, wo es wahrscheinlich sinnvoll wäre, diese den Gemeindestraßen vorzuziehen. Dieser Ansatz hat nur ein Problem. Es gibt keine Gewährleistung, dass die optimale Route immer über größere Straßen verläuft. Es kann manchmal sinnvoll sein, eine größere Straße zu verlassen, um eine Abkürzung über eine niedrigere Straße zu nutzen. Die Idee geht in die richtige Richtung, ist aber noch nicht ausgereift genug.

Ein besserer Ansatz wäre, wir nehmen unseren Nachbarschaftsradius und berechnen die optimale Route zwischen zwei Knoten $s$ und $t$, $\langle s, \ldots, u, v, \ldots, t \rangle$. Die Kante $(u, v) \in E_l$ auf dieser Route gehört zu der Highway Hierarchie, wenn $v$ nicht in der Nachbarschaft von $s$ ($v \notin N_l(s)$) und $u$ nicht in der Nachbarschaft von $t$ ($u \notin N_l(t)$) liegt. Die Kante darf nicht gänzlich in einer Nachbarschaft liegen. Die Idee dahinter ist, dass wir nur wichtige Straßen zur Hierarchie hinzufügen, die von vielen Knoten verwendet werden. Deswegen betrachten wir die lokalen Straßen nicht, da diese nur für sehr wenige Knoten relevant sind. Stellen wir uns eine Abfrage in dieser Hierarchie vor. Wir starten eine Vorwärtssuche von $s$ und eine Rückwärtssuche von $t$. Während der lokalen Suche von $s$ stoßen wir auf den Knoten $u$, und dasselbe gilt für $v$ von $t$. Danach wechselt die Suche in die Highway-Hierarchie. Durch den Wechsel in die höhere Schicht ignorieren wir alle Kanten und Knoten der unteren Schicht. Damit die optimale Route trotzdem bestehen bleibt, müssen alle Kanten zwischen $u$ und $v$ zur Highway-Hierarchie gehören. Wie wir später feststellen werden, ist das auch immer der Fall.

### 3.3.1 Berechnung der Hierarchie

Wir wollen einen Grpahen $G_l = (V_l, E_l)$ von $G'_l$ berechnen, mit $G_l \subseteq G'_l$. Wir berechnen die Highway-Hierarchie vom Kern, damit die Vorberechnung schneller vonstattengeht und der Graph stärker komprimiert wird.

*Unterteilung der Berechnung*

Für ein besseres Verständnis wird die Berechnung der Hierarchie unterteilt. Für jeden Knoten $s_0 \in G'_l$ werden die beiden Schritte nacheinander ausgeführt.

1. Es wird mithilfe einer lokalen Dijkstra-Suche von $s_0$ ein Suchbaum $B$ erzeugt. Das Wesentliche hierbei ist, die Suche lokal zu halten und zum richtigen Zeitpunkt abzubrechen.

2. Im zweiten Schritt wird der Suchbaum $B$ von den Blättern zur Quelle durchlaufen und bei jeder Kante entschieden, ob diese zu $E_l$ hinzugefügt werden soll oder nicht.

Der ganze Algorithmus beruht auf der Tatsache, dass die Highway-Hierarchie auch in einem lokalen Suchbaum vorkommt. Dadurch spart man sich die teure Berechnung der kürzesten Pfade zwischen allen Knoten.

#### 3.3.1.1 Erzeugung von $B$

Wir starten eine Dijkstra-Suche von $s_0$. Daraus ergibt sich der Suchbaum $B$, welcher die kürzesten Routen zu den abgearbeiteten Knoten beinhaltet. Bei dieser Definition wurde anders als in der Referenz angenommen, dass jeder Knoten nur einen Elterknoten besitzen kann. Folge dessen existiert nur eine kürzeste Route. Diese Annahme ist vertretbar, da eine exakt gleich lange Route in der Realität sehr unwahrscheinlich ist. Die Folgen wurden in Abschnitt 7.3.1 behandelt. Ebenfalls wurde bereits angewendet, dass es keinen Unterschied zwischen der Vorwärts/Rückwärtsnachbarschaft gibt. Jeder Knoten besitzt während der Suche einen Status von aktiv oder passiv. Am Anfang sind alle Knoten aktiv. Ein abgearbeiteter Knoten ist passiv, wenn die Bedingung

$$\textit{shortest path } P\langle s_0, ..., p\rangle : s_1 \prec p \wedge p \notin N_l(s_1) \wedge s_0 \notin N_l(p) \wedge \mid P \cap N_l(s_1) \cap N_l(p) \mid \leq 1$$

[Sch08, Formel (3.1)] erfüllt ist. Ein erreichter Knoten ist passiv, wenn sein Elternknoten passiv ist, ansonsten ist er aktiv. Die Suche ist beendet, wenn die Queue keinen aktiven Knoten beinhaltet.

*Erklärung der Abbruchbedingung*

- $s_1$ ist der direkte Nachfolge von $s_0$
- $s_1 \prec p$, $s_1$ ist ein Vorgänger von $p$ auf den Pfad $P$
- $p \notin N_l(s_1)$, $p$ ist nicht in der Nachbarschaft von $s_1$
- $s_0 \notin N_l(p)$, $s_0$ ist nicht in der Nachbarschaft von $p$
- $\mid P \cap N_l(s_1) \cap N_l(p) \mid \leq 1$, es gibt genau einen oder weniger Knoten auf dem Pfad $P$, welcher in der Nachbarschaft von $s_1$ und in der Nachbarschaft von $p$ ist.

Eine kurze Erläuterung, warum das Abbruchkriterium sinnvoll ist. Wenn wir uns entscheiden, einen Knoten auf passiv zu setzen, entscheiden wir uns indirekt dafür, alles was dahinter liegt, zu ignorieren. Um die Suche lokal zu halten, müssen wir die Knoten auf passiv setzen. Damit wir das dürfen, kommt $s_1$ in dem Kriterium vor. Dieser übernimmt die Verantwortung für alles, was nach $p$ kommt. Wir sehen durch $s_1$ leicht über die Nachbarschaft von $s_0$ hinaus. Dadurch können wir die ersten wichtigen Kanten zur Hierarchie hinzufügen. Weitere wichtige Kanten werden bei der Suche von $s_1$ hinzugefügt. Dieser gibt wiederum Verantwortung an seinen direkten Nachfolger ab. Und so weiter.

#### 3.3.1.1.1 Umsetzung

Für die Implementierung wird die obere Bedingung geteilt. Es gibt eine Grenzdistanz $b(x)$ und eine Referenzdistanz $a(x)$. Um später Fallunterscheidungen zu sparen, ist der Quellknoten $s_0$ gleichzeitig auch sein eigener Elternknoten.

*Grenzdistanz*
Diese beschreibt die Distanz von $s_0$ zum Rand der Nachbarschaft von $s_1$. Für $s_0$ ist die Grenzdistanz null. Sobald $s_1$ verarbeitet wurde, wird die Grenzdistanz auf den oben genannten Wert gesetzt $(d_l(s_0, s_1) + r_l(s_1))$ und an allen Nachfolgern weitergereicht. Durch die Grenzdistanz lässt sich der erste Knoten außerhalb der Nachbarschaft von $s_1$ ermitteln, wir nennen diesen $w$. Der direkte Vorfahre von $w$ ist $\bar{v}$, der letzte in $N_l(s_1)$, welcher für die Abbruchbedingung benötigt wird.

Formel [Sch08, Seite 53]: Für den Quellknoten gilt, $b(s_0) = 0$. Für jeden Knoten $x \neq s_0$ gilt, $b'(x) := d_l(s_0, x) + r_l(x)$ wenn $s_0 \in \gamma(x)$, sonst 0. $b(x) := max(\{b'(x)\} \cup \{b(y) \mid y \in \gamma(x)\})$, wobei $\gamma(x)$ die Elternmenge von $x$ beschreibt. Es sei angemerkt, dass bei uns die Elternmenge wegen der Annahme nur aus einem Knoten besteht.

*Referenzdistanz*
Werfen wir einen Blick auf die Abbruchbedingung. Dabei fällt uns auf, dass wir noch einen Teil prüfen müssen. Nämlich, ob $\bar{v}$ der einzige Knoten in $N_l(s_1) \cap N_l(p)$ ist. Dies ist gleichzusetzen mit der Überprüfung, ob der direkte Vorfahre von $\bar{v}$, $\bar{u}$ zu $N_l(p)$ gehört. Um die Überprüfung effizient umzusetzen, gibt es die Referenzdistanz. Die Referenzdistanz gibt die Distanz von $s_0$ zu $\bar{u}$ an, sobald $w$ abgearbeitet wurde. Sie wird ebenfalls wie die Grenzdistanz an die Nachfolger weitergereicht. Mit der Referenzdistanz, der aktuellen Distanz und dem Nachbarschaftsradius von p lässt sich leicht ermitteln, ob $\bar{u} \notin N_l(p)$ gilt.

Formel [Sch08, Seite 53]: Für den Quellknoten gilt $a(s_0) = \infty$. Für jeden Knoten $x \neq s_0$, gilt, $a'(x) := max\{a(y) \mid y \in \gamma(x)\}$; und $a(x) := max\{d_l(s_0, u) \mid y \in \gamma(x) \wedge u \in \gamma(y)\}$ wenn $a'(x) = \infty \wedge d_l(s_0, x) > b(x)$, sonst $a'(x)$.

Mit der Grenzdistanz und der Referenzdistanz haben wir jetzt die richtigen Werkzeuge. Durch die beiden vereinfacht sich die Abbruchbedingung für den Knoten $p$ auf:

$$a(p) + r_l(p) < d_l(s_0, p)$$

#### 3.3.1.1.2 Erweiterung zur Optimierung der Vorberechnungszeit

Der Algorithmus von oben ist sehr empfindlich auf lange Kanten. Das Problem entsteht, wenn der Endpunkt einer langen Kante aktiv ist. Derzeit beenden wir die Suche erst, wenn kein aktiver Knoten in der Queue existiert. Dadurch kann ein einziger aktiver Knoten den Suchbaum stark aufblasen. Um dies zu vermeiden, gibt es das Konzept der Außenseiter (Mavericks). Ein Knoten ist ein Außenseiter, wenn sein Gewicht das $f$-fache des Nachbarschaftsradius des Quellknotens überschreitet:

$$mav = d_l(s_0, v) > f \cdot r_l(s_0).$$

Die Suche von passiven Knoten wird nicht mehr fortgeführt, wenn alle aktiven Knoten Außenseiter sind, was einer künstlichen Beschneidung des Suchbaums gleicht. Die Folge: eine verkürzte Vorberechnungszeit. Der Nachteil: wir berechnen nur eine Teilmenge der Highway Hierarchie, was zu schlechteren Abfragezeiten führt. Näheres wird in Abschnitt 7.3.1 erläutert.

### 3.3.1.2 Auswahl der Highway Kanten

Nun wird der Suchbaum $B$ durchlaufen und die Highway Kanten ausgewählt. Es gehören alle Kanten $(u, v)$ zu der Hierarchie, die sich auf einem Pfad $\langle s_0, \ldots, u, v, \ldots, p\rangle$ in $B$ befinden und $u$ nicht zur Nachbarschaft von $p$ ($u \notin N_l(p)$) und $v$ nicht zur Nachbarschaft von $s_0$ ($v \notin N_l(s_0)$) gehört.

#### 3.3.1.2.1 Algorithmus

Um die Entscheidung, ob eine Kante zur Hierarchie hinzugefügt werden soll oder nicht, einfacher zu gestalten, definieren wir uns eine Wandergröße namens Schlupf. Dazu benötigen wir eine Menge $\mathcal{B}(u) = \{u\} \cup \{v \mid v \text{ ist ein Nachfolger von } u \text{ in } B\}$. Für einen Blattknoten $b$ ist $\mathcal{B}(b) := \{b\}$. Der Schlupf ist definiert als $\Delta(u) := min_{w \in \mathcal{B}(u)}(r_l(w) - d_l(u, w))$. Für einen Blattknoten $b$ ist $\Delta(b) := r_l(b)$.

Wir können den Schlupf rekursiv durch seine Kinder $\nu(u)$ berechnen. Dies ist durch $\Delta(u) = min(r_l(u), min_{c \in \nu(u)}\Delta_c(u))$ gegeben, wobei $\Delta_c(u) := \Delta(c) - d_l(u, c)$ ist. Dafür durchlaufen wir die abgearbeiteten Knoten $u \in B$ (nicht alle erreichten) in umgekehrter Reihenfolge. Durch diese Vorgehensweise wird sichergestellt, dass die Kindknoten immer vor den Elternknoten behandelt werden. Es wird immer der vorläufige Schlupf $\hat{\Delta}(u)$ der Elternknoten berechnet. Wir nehmen an, dass der vorläufige Schlupf nach der Verarbeitung aller seiner Kinder den wirklichen Schlupf $\Delta(u)$ gleicht. Für jeden Knoten $v$ berechnen wir den aktuellen Schlupf seiner Eltern $u$ mit $\Delta_v(u) = \Delta(v) - d_l(u, v)$.

Das Entscheidende hierbei ist, dass eine Kante zum Highway Netzwerk hinzugefügt wird, wenn $\Delta_v(u) < 0$ ist. Ist der aktuelle Schlupf kleiner als der vorläufige Schlupf $\left(\Delta_v(u) < \hat{\Delta}(u)\right)$, dann gleicht der vorläufige Schlupf dem aktuellen Schlupf $\left(\hat{\Delta}(u) = \Delta_v(u)\right)$. Die Berechnung stoppt, sobald ein Knoten $v \in N_l(s_0)$ abgearbeitet wird.

## 3.4 Überlagerungsgraph

Unser nächster und letzter Schritt in der Vorberechnung ist, einen Überlagerungsgraphen zu konstruieren. Die Motivation für einen Überlagerungsgraphen hat einen einfachen Hintergrund. Die kürzesten Pfade zwischen den Knoten bleiben erhalten. Ein Überlagerungsgraph für eine Knotenteilmenge $V_l \subseteq V_{l-1}$ und einer Kantenmenge stellt sicher, dass die Distanz für jeden Pfad zwischen zwei Knoten $(u, v) \in V_l \times V_l$ der Distanz im Ursprünglichem Graphen gleicht. Wir beziehen uns in den nächsten Definitionen immer auf die Layer $l-1$ und $l$. Dies hat den Hintergrund, da für die Berechnung der l-ten Schicht des Überlagerungsgraphen der Graph $G_{l-1}$ und die Knotenteilmenge $V_l$ benötigt wird.

*Definition* [Sch08, Seite 100]
Angenommen, wir haben einen Graphen $G_{l-1} = (V_{l-1}, E_{l-1})$ und eine Knotenteilmenge $V_l \subseteq V_{l-1}$. Der Graph $G_l = (V_l, E_l)$ ist ein Überlagerungsgrah von $G_{l-1}$, wenn für jedes Knotenpaar $(u, v) \in V_l \times V_l$, $d_l(u, v) = d_{l-1}(u, v)$ gilt. Wobei $d_{l-1}(u, v) = d_{G_{l-1}}(u, v)$ und $d_{G_{l-1}}(u, v)$ die kürzeste Distanz in $G_{l-1}$ beschreibt.

Diese Definition kann einfach auf eine Multilevel Version erweitert werden. Für die Highway-Hierachie Knoten Menge $V =: V_0 \supseteq V_1 \supseteq \ldots \supseteq V_L$ definieren wir die Überlagerungsgraphen $\mathcal{G} = (G_0, G_1, \ldots, G_L)$. Dabei beschreibt $G_0 = G$ und für die Layer $l \geq 1$ gilt, $G_l$ ist der Überlagerungsgraph von $G_{l-1}$.

Wir müssen noch definieren, wie wir die Knoten- und Kantenmenge des Überlagerungsgraphen ermitteln. Für die Knotenmenge $V_l$ des Überlagerungsgraphen $G_l$ wählen wir die Highway-Hierarchie Knoten der dazugehörigen Schicht. Diese bieten bereits eine gute Abstufung, wo wichtigere Knoten weiter oben in der Hierarchie vorkommen. Für die Kanten benötigen wir ein neues Konzept, nämlich jenes der abgedeckten Pfade (Covering-Paths). Dieses Konzept wird später auch für die Abfrage benötigt.

### 3.4.1 Covering-Paths

Der Name lässt bereits vermuten, was die Idee hinter diesem Algorithmus ist. Wir berechnen eine Menge an Pfaden in $G_{l-1} = (V_{l-1}, E_{l-1})$, die von einem Knoten $u \in V_l$ abgedeckt sind. Abgedeckt bedeutet das sich auf den kürzesten Pfad ein Knoten der Menge $V_l$ befindet. Näheres in der Definition.

*Definition*
Für einen Graphen $G_{l-1} = (V_{l-1}, E_{l-1})$, einer Knotenteilmenge $V_l \subseteq V_{l-1}$ und einen Knoten $s \in V_{l-1}$, definieren wir eine Menge $C \subseteq \{\langle s, \ldots, u\rangle \mid u \in V_l\}$. Diese Menge beschreibt die Covering-Paths von $s$ in Bezug auf $V_l$, wenn für jeden Knoten $t \in V_l$ der von $s$ erreichbar ist, ein Knoten $u \in V_l$ auf den kürzesten $s$-$t$ Pfad $P$ liegt. Dann ist der Pfad $P\,|_{s \to u} \in C$. Kurz gesagt:

$$P = \langle \underbrace{s, \ldots, \overbrace{u}^{\in V_l}}_{\in C}, \ldots, \overbrace{t}^{\in V_l} \rangle.$$

#### 3.4.1.1 Algorithmus

Es gibt mehrere Ansätze, um dieses Problem zu lösen. Sie unterscheiden sich in der Art und Weise, wie die Suche beschnitten wird. In dieser Arbeit betrachten wir nur einen Ansatz näher, der meiner Meinung nach am vielversprechendsten aussieht, nämlich den Stall-on-Demand Algorithmus. Für die Implementierung nutzen wir eine Dijkstra-Suche. Wir starten die Suche von $s \in V_{l-1}$. Ein abgearbeiteter Knoten $u \in V_{l-1}$ wird als abgedeckt bezeichnet, wenn sich auf dem kürzesten Pfad von $s$ zu $u$ ein Knoten $v \in V_l$ befindet. Ein erreichter Knoten ist abgedeckt, wenn sein vorläufiger Elternknoten abgedeckt ist. Vorläufig, da bei einem erreichten Knoten noch nicht sichergestellt ist, ob der derzeitige Pfad der kürzeste ist. Der Suchbaum $B$ ist abgedeckt, wenn alle Knoten in der Queue abgedeckt sind.

Bei dieser Variante wird ein abgedeckter Knoten gestutzt (pruned). Gestutzt bedeutet, die Kanten werden nicht weiter verfolgt. Dieses Verhalten hat zwei Nachteile. Erstens: die Covering-Paths Menge wird unnötig groß. Zweiten: der Suchbaum kann wachsen, da um die Knoten herum gesucht wird. Um dieses Verhalten zu umgehen, gibt es den stalling Prozess (Abwürgen des Prozesses), welcher von gestutzten Knoten ausgeführt wird. Der Prozess findet Knoten, die über einen suboptimalen Pfad erreicht wurden (wir bezeichnen diese Knoten als stalled). Suboptimaler Pfad bedeutet, wenn es eine alternative Route von $s$ über $u$ nach $v$ gibt, die kürzer ist als der beste Pfad bis jetzt zu $v$. Auch bei solchen Knoten wird die Hauptsuche nicht mehr fortgeführt, somit gesutzt. Wie finden wir diese Pfade? Wir starten eine Suche von $u$ ($u$ ist ein gestutzter Knoten) und berücksichtigen dabei lediglich erreichte Knoten. Die Suche wird nur bei Knoten fortgesetzt, die über einen suboptimalen Pfad erreicht wurden. Der Prozess wird gestartet, wenn in der Hauptsuche eine Kante von einem erreichten Knoten (in unserem Beispiel $v$) zu einem gestutzten Knoten ($u$) entspannt wird.

Zur Veranschaulichung werfen wir einen Blick auf das Beispiel in Abbildung 3.3. Wir wollen die Covering-Paths von $s$ ermitteln. Dazu starten wir die Suche von $s$. Als erstes wird der Knoten $u$ erreicht. Dieser ist aus der Menge $V_l$, der Pfad ist abgedeckt und die Suche wird dort nicht mehr weitergeführt. Als nächstes wird $v$ über einen suboptimalen Pfad erreicht. Nehmen wir an, es wird als erstes die Kante $(v, w)$ entspannt. Somit wird auch $w$ über einen suboptimalen Pfad erreicht. Die Kante $(v, u)$ löst ein stalling Prozess von $u$ aus. Dabei werden die Knoten $v$ und $w$ gefunden, welche über einen suboptimalen Pfad erreicht wurden. Die beiden Knoten sind stalled. Die Suche wird bei $w$ nicht weiter fortgeführt, $v$ wurde bereits abgearbeitet. Als nächstes wird $x$ erreicht und die Kante $(x, w)$ löst einen weiteren stalling Prozess aus. In diesem Beispiel besteht die Covering-Paths Menge $C$ nur aus dem Pfad von $s$ nach $u$.

### 3.4.2 Kantenmenge des Überlagerungsgraphen

Wir müssen noch definieren, wie wir jetzt mit diesem Algorithmus die Kanten des Überlagerungsgraphen berechnen. Wie bereits oben definiert, ist die Knotenmenge des Überlagerungsgraphen die Knotenmenge der jeweiligen Highway Hierarchie Schicht $V_l$.

1. Wir berechnen für jeden Knoten $u \in V_l$ die Covering-Paths Menge in $G_{l-1}$ in Bezug auf $V_l \setminus \{u\}$.

2. Von jedem Pfad nehmen wir den nächsten Knoten $v \in V_l$ zu $u$ und fügen eine Kante $(u, v)$ zu $E_l$ hinzu mit dem Gewicht $d_{G_{l-1}}(u, v)$ (länge des Pfades von $u$ zu $v$ in $G_{l-1}$).

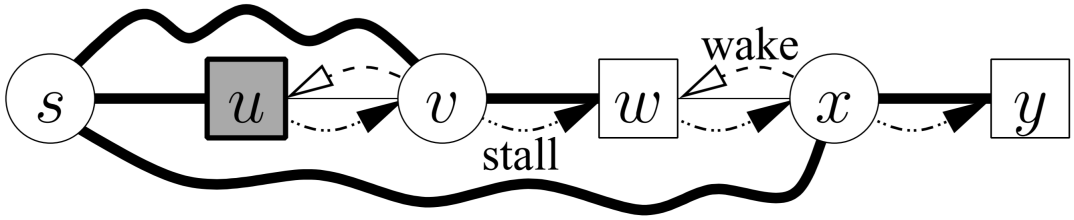



**Abbildung 3.3:** [Sch08, Figure 4.3] Ein kurzes Beispiel der Covering-Paths. Legende: $s$ ist der Quellknoten. Die Rechteckigenknoten sind aus der Menge $V_l$. Alle geraden Kanten haben ein Gewicht von 1, die geschwungenen von 10. Die dicken Kanten gehören zum Suchbaum $B$. Der Knoten $u$ ist der einzige Endpunkt der Covering-Paths

Die beiden Schritte führen wir für jede Schicht $l \geq 1$ aus, bis $l = L$.

#### 3.4.2.1 Erweiterung zur Minimierung der Kantenmenge

In der vorherigen Konstruktion fügen wir eventuell einige redundante Kanten hinzu, die später die Abfrage verlangsamen. Um die redundanten Kanten zu entfernen, starten wir eine Suche von jedem Knoten $u \in V_l$ in $G_l$.

1. Die Suche wird beendet, wenn alle benachbarten Knoten $v$ abgearbeitet sind.
2. Wenn während der Suche ein Knoten über zwei verschiedene Pfade mit dem gleichen Gewicht erreicht wird, entscheiden wir uns für den Pfad mit mehr Kanten.
3. Wir entfernen alle Kanten $(u, v)$, wo $v$ über einen Pfad mit mehr als einer Kante erreicht wurde. Die Kante $(u, v)$ liefert keinen Mehrwert.

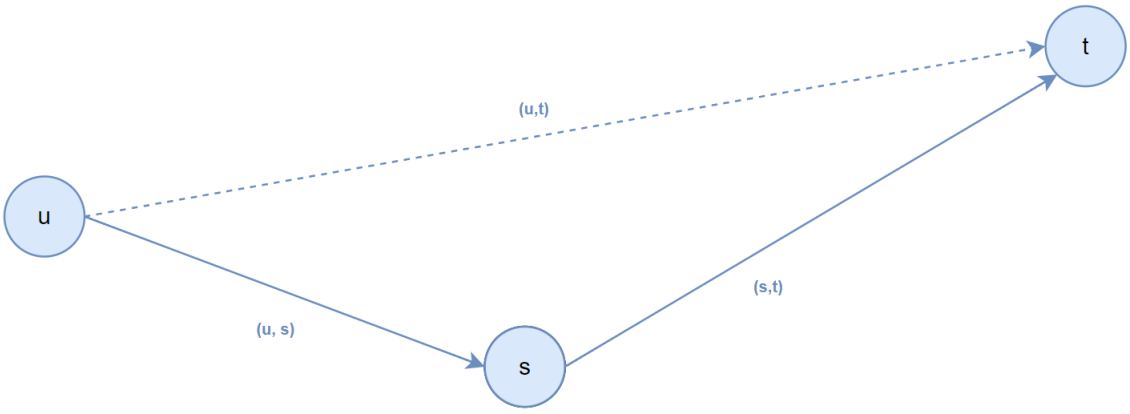



**Abbildung 3.4:** Beispiel: Wir entfernen die gestrichelte Kante $(u, t)$, wenn der Pfad $P\langle u, s, t\rangle$ dasselbe Gewicht besitzt.

**Referenzen:**
Der gesamte Abschnitt bezieht sich auf [Sch08, S. 43–113]. Dominik Schultes verweist noch auf [[Sch05], [SS07], [DGJ09], [SS05], [SS12], [SS12], [SS06]].

# 4 Abfrage

Die vorherigen Schritte dienen nur dazu, damit die Abfrage so schnell wie möglich vonstattengeht. In dieser Arbeit fokussieren wir uns auf die Level-synchronisierte Version. Es gibt auch eine asynchrone Variante, die wir im Kapitel 8 kurz betrachten.

**Ablauf** Die Suche startet in $G_0$. Wir ermitteln die Covering-Paths vom Startpunkt $s$ in $G_0$ in Bezug auf $V_1$. Für jeden Endpunkt $u$ der Pfade berechnen wir die Covering-Paths $C(u)$ in $G_1$ in Bezug auf $V_2$. Im nächsten Schritt berechnen wir für jeden Endpunkt der Pfade $\bigcup C(u)$ die Covering-Paths in $G_2$ mit $V_3$. Gleiches gilt für alle weiteren Schichten. Die vorherigen Schritte werden von beiden Seiten immer abwechselnd ausgeführt. Das heißt, nachdem die Covering-Paths der Vorwärtssuche in $G_0$ ermittelt wurden, wird das gleiche in der Rückwärtssuche vom Endpunkt $t$ ausgeführt. Wenn beide Suchen in $G_0$ abgeschlossen sind, wechseln wir nach $G_1$. Und so weiter. Während des Prozederes halten wir Ausschau nach dem vorläufigen kürzesten Pfad. Dieser ergibt sich, wenn ein Knoten aus beiden Richtungen erreicht wurde. Die Vorwärts- oder Rückwärtssuche wird beendet, wenn die Gewichte der Knoten in der jeweiligen Queue größer sind als der vorläufige kürzeste Pfad, oder wenn alle Knoten abgearbeitet wurden. Dies gewährleistet, dass wirklich der kürzeste Pfad gefunden wurde.

**Referenzen:**
Der Abschnitt bezieht sich auf [Sch08, S. 102]. Dominik Schultes verweist auf [SS07].

# 5 Anstoß für die Richtigkeit

Bis jetzt haben wir nur die Funktionsweise der verwendeten Algorithmen betrachtet. Dabei haben wir einen wesentlichen Teil vernachlässigt, nämlich die Sicht auf die Richtigkeit. Was einer der wichtigsten Punkte ist. Denn wenn der Algorithmus schnell, aber ein falsches Ergebnis liefert, sind unsere ganzen Bemühungen umsonst. Das Ziel in dem nächsten Abschnitt ist es, eine Intuition für die Richtigkeit zu geben. Für eine genauere Beweisführung verweise ich auf die dementsprechenden Kapitel in [Sch08].

## 5.1 Ansatz

Lassen wir die oberen Schritte Revue passieren. Einer der entscheidendsten Punkte ist die Auswahl der Highway Kanten. Denn dadurch ergibt sich unsere Knotenmenge, woraus wir unseren Overlaygraphen erzeugen. Das Spannende dabei ist, dass die Auswahl nur die Vorbereichnungszeit des Netzwerks sowie die Abfragezeit beeinflusst, aber nicht die Richtigkeit des Algorithmus.

Zur Wiederholung werfen wir noch einmal einen Blick auf die Abfrage. Die Suche startet in $G_0$. Zuerst berechnen wir für den Startknoten $s \in V_0$ die Covering-Paths in Bezug auf $V_1$. Die Definition der Covering-Paths besagt, dass auf dem kürzesten Pfad zu jedem erreichbaren Knoten $t \in V_1$, sich ein Knoten $u \in V_1$ befinden muss (Die Definition ist bereits angewendet auf die derzeitige Schicht). Für jeden Endknoten der Pfade berechnen wir die Covering-Paths in $G_1$ in Bezug auf $V_2$. Und so weiter. Der Covering-Paths-Algorithmus 3.4.1 implementiert die Definition wie folgt. Er gleicht einer Dijkstra-Suche mit der Erweiterung, dass die Suche an gewissen Knoten gestutzt wird. Für die Abfrage prüft der Algorithmus ebenfalls, ob ein Knoten bereits von beiden Seiten erreicht wurde, oder der Zielknoten ist $\rightarrow$ Pfad von $s$ nach $t$. Die Suche wird abgebrochen, wenn der Suchbaum abgedeckt oder der niedrigste Key in der Vorwärts- sowie in der Rückwärtsqueue größer ist als der vorläufige kürzeste Pfad von $s$ nach $t$.

Um zu zeigen, dass die Abfrage korrekt ist, berücksichtigen wir folgende Fälle:
Es wurde angenommen, dass der Covering-Paths Algorithmus einer Dijkstra-Suche gleicht, ohne die in Kapitel 3.4.1 beschriebenen Optimierungen. Diese betrachten wir separat.

1. Es existiert auf jedem Pfad ein Knoten der Bezugsmenge. Suchbaum ist abgedeckt:

   Die Covering-Paths konnten in der Vorwärts- sowie in der Rückwärtssuche berechnet werden. Wir wechseln in die nächste Schicht des Overlaygraphen. Durch diesen Wechsel wird die kürzeste Route nicht verändert. Betrachten wir dazu die Eigenschaft des Overlaygraphen: Für jedes Kontenpaar $(u, v) \in V_{l+1} \times V_{l+1}$ gilt, $d_{l+1}(u, v) = d_l(u, v)$. Das heißt, die kürzeste Distanz zwischen den Knoten bleibt erhalten. Wir wissen ebenfalls, dass auf jeden kürzesten Pfad im Suchbaum sich ein Knoten der Menge $V_{l+1}$ befindet. Der kürzeste Pfad von s nach t muss somit durch einen dieser Knoten gehen (Ein Subpath von einem kürzesten Pfad ist ebenfalls ein kürzester Pfad). Demzufolge reicht es aus, die Suche nur von den Endknoten der Covering-Paths fortzuführen.

2. Es existiert nicht auf jedem Pfad ein Knoten der Bezugsmenge. Suchbaum ist nicht abgedeckt (gilt auch für die letzte Schicht):

Die Covering-Paths konnten in der Vorwärts- oder in der Rückwärtssuche nicht berechnet werden. Wenn ein Knoten von beiden Suchen erreicht wird oder der Endknoten ist, verfahren wir, wie in Drittens beschrieben. Ist das nicht der Fall, wird die Suche solange fortgeführt, bis alle erreichbaren Knoten abgearbeitet wurden. Denn es sind nicht alle Pfade abgedeckt und somit wird auch nicht abgebrochen, bis alle Knoten abgearbeitet sind. Infolgedessen gibt es keinen Pfad von $s$ nach $t$. Denn der Dijkstra-Algorithmus findet immer die kürzeste Route, falls eine existiert. Die Suche kann zu diesem Zeitpunkt beendet werden.

3. Die beiden Suchen treffen sich.

   Eine vorläufige kürzeste Route $P^*$ wurde gefunden. Die Suche darf zu diesem Zeitpunkt noch nicht abgebrochen werden. Es besteht nach wie vor die Möglichkeit, dass es einen kürzeren Pfad gibt. Der Ursprung liegt im Suchaufbau. Wenn die Suche einmal in eine höhere Schicht gewechselt ist, werden die Knoten einer unteren Schicht nicht mehr berücksichtigt. Betrachten wir dazu das Beispiel in Abbildung 5.1. Im oberen Pfad $P$ haben beide Suchen $a$ erreicht und somit einen Pfad von $s$ nach $t$ gefunden. Der untere Pfad $Q$, welcher kürzer ist, wurde bis dato noch nicht gefunden. Das liegt daran, da die Vorwärtssuche von $c$ nicht weitergeführt wird. Wie oben schon beschrieben, werden Knoten einer unteren Schicht nicht mehr betrachtet. Um den Pfad zu finden, muss die Rückwärtssuche von $b$, $c$ erreichen. Deswegen wird zuerst P gefunden. Wegen diesem Verhalten beenden wir die Suche erst, wenn die Keys in den Queues größer sind als die Länge der kürzesten Route. Sobald die Suche endet, haben wir die kürzeste Route gefunden. Diese Behauptung basiert auf der Tatsache, dass der Dijkstra-Algorithmus immer den kürzesten Pfad findet und wie in Erstens beschrieben unsere Hierarchie diese Eigenschaft nicht verletzt.

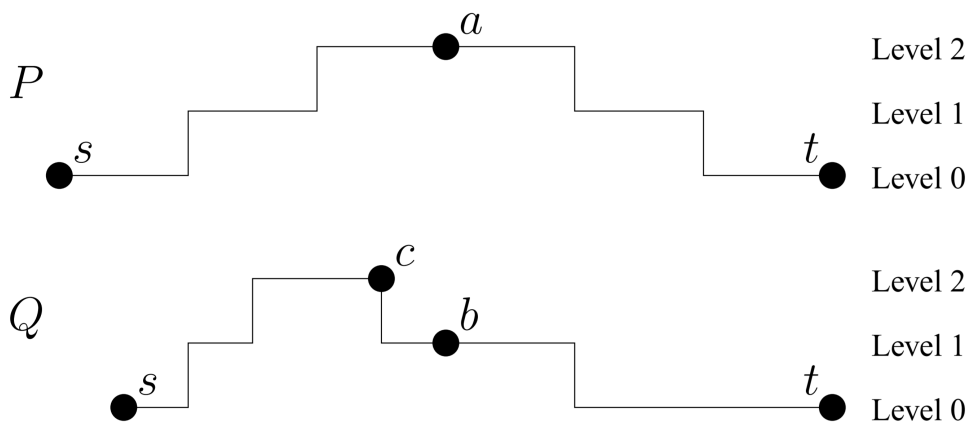



**Abbildung 5.1:** [Sch08, Figure 3.11] Verschiedene Suchverläufe

4. Spezialfall in der ersten Schicht: Der Endknoten wird in der Vorwärts- oder Rückwärtssuche erreicht und abgearbeitet.

   Durch den Dijkstra-Algorithmus haben wir die kürzeste Route $P$ gefunden Die Vorwärts- oder Rückwärtssuche wird beendet, da jeder Key in der Queue ein hö-

heres Gewicht besitzt als die Länge von $P$. Wenn die Keys in der anderen Queue geringer sind, wird die Suche dort weitergeführt, bis das Abbruchkriterium gilt.

Es bleibt noch zu zeigen, dass dies auch für den Covering-Paths Algorithmus mit den Optimierungen gilt. Im Gegensatz zu jenen im Beweis wird die Suche bei abgedeckten Knoten nicht mehr fortgeführt; für Details siehe Kapitel 3.4.1. Betrachten wir folgende Fälle:

1. Kein Pfad ist abgedeckt oder alle Pfade sind abgedeckt.

   In diesen beiden Fällen ändert sich nichts zu oben. Dabei ist anzumerken, dass das Abbrechen der Suche bei abgedeckten Pfaden die Richtigkeit nicht beeinflusst. Dies hat den Hintergrund, da auf dem Pfad bereits jeder Knoten abgedeckt wäre. Eine Folge jedoch ist, es wird um die abgebrochenen Knoten rundherum gesucht. Dazu werfen wir einen kurzen Blick auf Abbildung 3.3. Die Suche wird von $u$ nicht fortgeführt. $v$ und alle Knoten danach werden über einen suboptimalen Pfad erreicht. Gegensätzlich zur Annahme wird die Covering-Paths Menge größer, nicht kleiner. Um sie wieder zu verkleinern, gibt es den Stalling-Prozess. Dieser wird von abgedeckten Knoten ausgeführt und findet Knoten, die über einen suboptimalen Pfad erreicht wurden. Dies beeinflusst aber nur die Laufzeit und nicht die Richtigkeit, da nur die suboptimalen Pfade entferent werden.

2. Pfade sind teils abgedeckt.

   Die Pfade werden nicht vollständig abgedeckt. Folglich wird das gesamte Netzwerk abgesucht, bis auf die Pfade, die bereits abgedeckt sind. Die Suche endet, wenn keine Knoten mehr in der Queue vorhanden sind. Es werden nur die abgedeckten Pfade zurückgeliefert. In der Abfrage würde die Suche nur von den Endpunkten jener Pfade fortgeführt werden, der Rest wurde bereits durchsucht.

Zusammenfassend: Der Algorithmus gleicht im schlechtesten Fall einem Dijkstra. Im Schnitt werden wir aber die Abfragezeit beschleunigen, wobei die Richtigkeit nicht beeinflusst wird.

# 6 Umgebung

## 6.1 Projekt

Das Thema der Bachelorarbeit und die Umsetzung wurden von der RISC Software GmbH vorgeschlagen und betreut. Insbesondere die Daten zur Testung der Implementierung. Die nachfolgende Information wurde mir von der RISC Software GmbH zur Verfügung gestellt.

Das Forschungsprojekt EVIS.AT (Echtzeit Verkehrsinformation Straße Österreich) und die daraus gewonnenen Verkehrsdaten dienen dabei als Basis für die weiters genannten Projekte. In EVIS.AT wurden mehrere Flotten (LKW, PKW, eine Kombination aus LKW und PKW, Einsatzfahrzeug, Taxi) für eine Datensammlung akquiriert sowie Streckenabschnitte mit Sensorik (VDL/Zählschleifen, Bluetooth) für die Verkehrszählung ausgestattet. Ziel dieses Projektes ist Echtzeitverkehrslageinformationen für den Raum Oberösterreich zur Verfügung zu stellen, diese Verkehrslage ist auch ein integraler Bestandteil der österreichweiten Verkehrslage der VAO.

Das Wesentliche dabei: Die Informationen umfassen mehr als 12.000 Kilometer Straßennetz (GIP), somit neben den Autobahnen und Schnellstraßen auch die wichtigsten Landes- und Gemeindestraßen. Diese Infos werden in Echtzeit übermittelt und bilden so auch die Basis für ein verkehrsträgerübergreifendes (also nicht „nur“ auf den individuellen Straßenverkehr beschränktes) Verkehrsmanagement. EVIS.AT war ein Kooperationsprojekt unter der Leitung der ASFINAG, welches mittlerweile erfolgreich abgeschlossen wurde und jetzt die Ergebnisse direkt den Partner:innen zur Verfügung stellt. Die RISC Software GmbH als Teil der IST-OÖ Organisation ist EVIS-Kooperationspartner und auch Datenlieferant für das oberösterreichisch Landesstraßennetz.

- Die Graphenintegrations-Plattform GIP, das Referenzsystem der öffentlichen Hand für Verkehrsinfrastrukturdaten. `https://gip.gv.at/`
- Echtzeit Verkehrsinformation Straße Österreich (EVIS_AT). `http://www.evis.gv.at/`

Die RISC Software GmbH gab die Arbeit in Auftrag. Ziel war es, einen zusätzlichen Routingalgorithmus dem bestehenden Framework hinzuzufügen. In diesem Abschnitt geht es genau um dieses Framework und den Graphen, der für das Testen verwendet wurde.

## 6.2 Graph

Als Basis dient der Verkehrsgraph von Österreich. Der Graph besteht aus 663 347 Kanten und 331 094 Knoten. Die Kanten stellen die Straßen und die Knoten Straßenschnittstellen dar, wobei auch Knoten existieren, welche nur für den Straßenverlauf zuständig sind. Wie man in Abbildung 6.1 erkennen kann, ist der Graph nicht vollständig, besonders in der westlichen Hälfte. Zum Testen ist der Graph bereits groß genug, deswegen wurde nicht ganz Österreich herangezogen.

### 6.2.1 Interne Repräsentation

Intern wird der Graph in einem Array von Kanten und Knoten abgespeichert. Die Knoten referenzieren auf die eingehenden und ausgehenden Kanten. Jede Kante hat wiederum Spu-

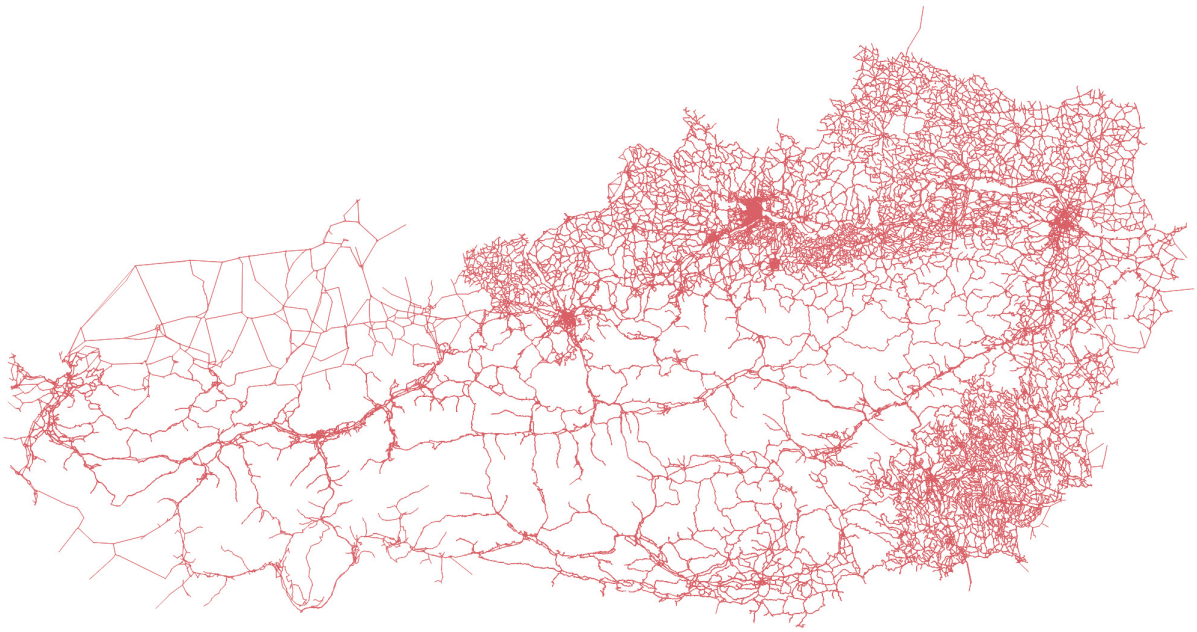


**Abbildung 6.1:** Testgraph von OpenStreetMap, Österreich

ren (Lanes), welche die Fahrstreifen widerspiegeln. Diese besitzen wiederum Verbindungen (Connections), welche angeben, wo abgebogen oder weitergefahren werden darf. Die Kanten besitzen ebenfalls eine Länge als Gewicht, um die kürzeste Distanz zu berechnen.

## 6.3 Framework

Das Framework stellt bereits viele Klassen und Interfaces bereit. Die Bedingung war, dass der neue Algorithmus auf denselben Interfaces aufbaut, um einen konfliktfreien Wechsel zu gewährleisten ohne weitere Sideeffects. Dadurch wurden alle Klassen gleich belassen, jede die erweitert werden musste, wurde abgeleitet und die notwendigen Interfaces implementiert.

# 7 Implementierungsdetails

Anmerkung: In den nächsten Abschnitten ist oft vom Dijkstra-Algorithmus die Rede. Bei diesem wurde die vorhandene Implementierung verwendet und an gewissen Stellen erweitert, für die jeweilige Anwendung.

## 7.1 Kern

Der gesamte Graph wird kopiert und in die nächste Schicht eingefügt. Dieser kann nach Belieben geändert werden, ohne den Graph in der unteren Schicht zu beeinflussen. Im Laufe der Vorberechnung werden alle überbrückten Kanten und Knoten entfernt. Vorteil: Es wird keine Überbrückungsmenge benötigt und der Check, ob eine Kante/Knoten zum Kern gehört oder nicht, fällt weg. Nachteil: Die Knoten werden redundant gespeichert. Bei den Kanten ist dies nicht so stark ausgeprägt, da viele nur in einer Schicht existieren.

Wir müssen die Definition von Kapitel 3.1 erweitern. Der verwendete Graph besteht, wie in Sechstens beschrieben, auch aus Spuren und Verbindungen. Dadurch gibt es Shortcut-Spuren und Verbindungen. Alle Shortcut-Spuren, die zum selben Knoten führen und vom selben Knoten kommen, werden in einer Shortcut-Kante zusammengefasst. Es wird somit nicht für jede Verbindung eine neue Kante erzeugt, sondern versucht, diese wieder in einer Kante zu bündeln.

Des Weiteren mussten einige Sonderfälle behandelt werden. Bei beiden Fällen wird der Knoten übersprungen und zur Kernknotenmenge hinzugefügt.

- Wenn der Knoten keine weiteren Kanten besitzt.
  - Die Spuren der Kanten, die zu diesem Knoten führen, besitzen keine weiteren Verbindungen mehr. Dadurch würde keine Überbrückungskante erzeugt werden. Resultat: Kante und Knoten werden gelöscht.
- Gleiches gilt für eine Spur, die an einem Knoten endet.

### 7.1.1 Erweiterung

Um den erzeugten Graphen zu vereinfachen, können in der Hierarchie die Spuren und Verbindungen ausgelassen werden, sodass lediglich Knoten und Kanten verwendet werden. Dies führt zu einer Verringerung des Speicherverbrauchs und vereinfacht den gesamten Algorithmus.

## 7.2 Nachbarschaftsradius

Für jeden Knoten der aktuellen Schicht wird eine Dijkstrasuche gestartet. Wegen des ungerichteten Graphens werden die Fahrstreifen und Verbindungen nicht genutzt, sondern nur die Kanten (Straßen) der Knoten. Es wird nicht zwischen Eingangs- und Ausgangskanten unterschieden - jede Kante wird entspannt. Die Suche endet, sobald der H-ste Knoten aus der Queue entfernt wird. Wenn der Knoten weniger Nachbarn besitzt, dann wird der weitest entfernte herangezogen. Der Nachbarschaftsradius wird auf die Distanz zu diesem Knoten

gesetzt. Nach jedem Durchlauf müssen die Felder der besuchten Knoten zurückgesetzt werden, damit die nachfolgenden Suchen nicht beeinflusst werden. Um dies effizient umzusetzen, merken wir uns alle besuchten Knoten.

**Abbildung 7.1:** Beispiel für H = 50. Die Rote Linie ist der Ausgangspunkt. Alle Blauen Linien gehören zur Nachbarschaft.

## 7.3 Highway Hierarchie

### 7.3.1 Erzeugung von B

Die Implementierung weicht von der Definition in einem Bereich ab. Es kann jeder Knoten nur einen Elternknoten besitzen, wie bereits in Kapitel 3.3 erwähnt wurde. Die Annahme beruht auf der Tatsache, dass es in der Realität sehr unwahrscheinlich ist, dass zwei unterschiedliche Pfade das gleiche Gewicht besitzen. Durch die Annahme wird Speicher und Rechenzeit eingespart.

Um die Folgen einzuschätzen, nehmen wir an, es existieren zwei solche Pfade. Die Grenzdistanz und Referenzdistanz können höchstens kleiner ausfallen als in der Theorie definiert, da immer das Maximum weitergereicht wird. Somit brauchen wir nur jene Fälle betrachten, wo die Grenzdistanz und/oder Referenzdistanz kleiner ausfallen als idealerweise.

Angenommen, wir reichen die kleinere der beiden Grenzdistanzen weiter. Folge davon ist, dass die Referenzdistanz ebenfalls kleiner ausfallen kann.

$$a(x) := max\{d_l(s_0, u) \mid y \in \gamma(x) \wedge u \in \gamma(y)\} \text{ wenn } a'(x) = \infty \wedge \underbrace{d_l(s_0, x) > b(x)}_{\text{wird früher erfüllt}}$$

In weiterer Folge wird die Abbruchbedingung früher erfüllt, was wiederum zu einem kleineren Suchbaum führt. Durch dieses Verkleinern wird die Richtigkeit nicht eingeschränkt, da nur weniger Knoten erreicht werden. Die kürzeste Distanz zwischen diesen bleibt erhalten. Wir berechnen somit eine Teilmenge des wirklichen Highway-Netzwerks. Mit dem Konzept der Außenseiter erzielen wir im Prinzip das Gleiche, nur gewollt. Wie bereits im Kapitel 3.3.1.1.2 beschrieben, sieht dies für die Abfragezeit anders aus. Wichtige Kanten aus dem Netzwerk wegzulassen sorgt dafür, dass bei der Abfrage später in die höhere Schicht gewechselt wird. Späterer Wechsel → größerer Suchbaum → schlechtere Abfragezeit.

Der Algorithmus wurde identisch zur Definition implementiert, mit dem Konzept der Außenseiter. Es werden Zähler verwendet, um zu ermitteln, ob alle Knoten passiv oder Außenseiter sind. Der Suchbaum wird mittels eines Stacks gespeichert. Wenn ein Knoten aus der Queue entfernt wird, wird dieser auf den Stack gepusht. Nach jedem Knoten müssen die Felder zurückgesetzt werden, um die nachfolgenden Suchen nicht zu beeinflussen. Wie auch beim Nachbarschaftsradius merken wir uns alle besuchten Knoten, um nur diese zurückzusetzen.

### 7.3.2 Auswahl der Highway Kanten

Die Annahme bei der Berechnung des Suchbaumes $B$, dass jeder Knoten nur einen Elterknoten besitzen kann, macht sich auch hier bemerkbar. Wir müssen für jeden Knoten nur einen Elternknoten berücksichtigen, um den aktuellen Schlupf zu berechnen. Wir nehmen nach und nach die Knoten vom Stack, um den Suchbaum von den Blättern zum Ursprung zu durchlaufen. Für jeden berechnen wir den vorläufigen Schlupf des Elternknotens und überprüfen, ob die Kante zur Hierarchie hinzugefügt werden soll oder nicht. Wenn ein Knoten vom Stack genommen wird, der innerhalb der Nachbarschaft von $s_0$ ist, wird abgebrochen. Der Rest ist identisch zur Theorie.

## 7.4 Covering-Paths

Die Implementierung weicht nicht von der Beschreibung in Kapitel 3.4.1 ab. Für jeden Knoten prüfen wir, ob dieser über einen suboptimalen Pfad erreicht wurde. Dazu inspizieren wir alle eingehenden Kanten, ob ein Nachbar abgedeckt oder stalled ist. Wenn ja, lösen wir einen stalling Prozess aus. Er gleicht einer Breath-First-Search. Das hat den Hintergrund, da wir die Suche nur bei Knoten fortführen, wo eine Suboptimalität festgestellt wird.

Ein Zähler gibt Auskunft, ob alle Pfade bereits abgedeckt sind. Der Algorithmus liefert nur die Endpunkte (der erste Knoten auf dem Pfad aus der höheren Schicht) mit der Distanz zum Startkonten zurück. Um diese zu ermitteln, verwenden wir einen Stack, der alle abgearbeiteten Knoten beinhaltet. Der Stack wird durchlaufen und alle abgedeckten Knoten überprüft, ob der Elternkonten ebenfalls abgedeckt ist. Wenn nicht, ist das der erste abgedeckte Knoten auf dem Pfad und gehört zur Menge der Endkonten.

## 7.5 Überlagerungsgraph

Wir berechnen die Covering-Paths für alle Knoten aus $V_l$ in $G_{l-1}$. Für jeden Endknoten wird eine neue Kante hinzugefügt. Das Gewicht der Kante entspricht der Distanz zum Ursprungsknoten. Wir speichern ebenfalls den Pfad, aus welchem die Kante besteht. Das hat den Grund, damit wir später den Pfad wieder rekonstruieren können. Wir müssen ebenfalls die richtigen

Verbindungen zwischen den Knoten/Kanten herstellen. Dazu betrachten wir die Abbildung 7.2. Wir fügen eine Verbindung von $(u, s)$ nach $(s, t)$ hinzu, wenn es eine Verbindung von $(a, s)$ nach $(s, b)$ gibt. Dabei besteht die Kante $(u, s)$ aus dem Pfad $P'$ und $(s, t)$ aus $P''$.

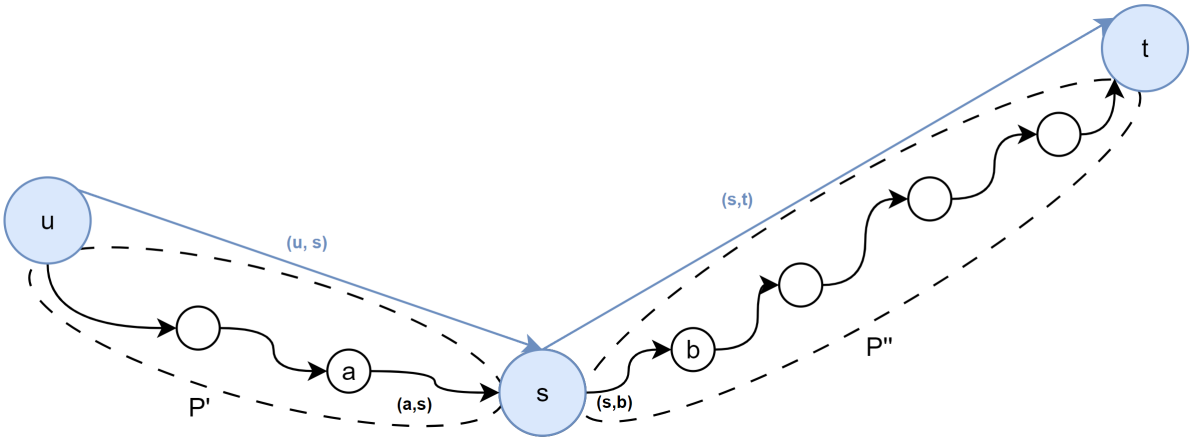



**Abbildung 7.2:** Beispiel: Erstellung der Verbindungen zwischen den Kanten des Überlagerungsgraphen. Die schwarzen Kreise/Pfeile gehören zu $G_{l-1}$ alle blauen zu $G_l$.

Nachdem die gesamte Schicht $G_l$ erzeugt wurde, reduzieren wir die Kantenmenge. Wir führen dazu eine Suche von jedem Knoten in $G_l$ aus, bis alle Nachbarknoten abgearbeitet sind. Es werden direkte Kanten zu den Nachbarn entfernt, wenn es einen anderen Pfad gibt, der das gleiche Gewicht besitzt. Die Gleichheitsrelation muss eigenständig definiert werden, da Floatingpoint Zahlen so gut wie nie gleich sind, wegen der vielen Nachkommastellen und Rundungsfehlern. Dazu verwenden wir einen relativen Fehler, welcher kleiner als Epsilon sein muss, um als gleich zu gelten. Durch Epsilon lässt sich die Komprimierung regeln. Desto größer Epsilon, desto mehr Kanten werden entfernt, was die Abfrage beschleunigt, aber auch eine gewisse Ungenauigkeit mit sich bringt. In der Implementierung wurde ein Epsilon von 0.01% verwendet.

## 7.6 Abfrage

Es wurde die Abfrage wie in Kapitel 4 beschrieben implementiert. Ab der zweiten Schicht werden die Covering-Paths von mehreren Endpunkten hintereinander berechnet. Damit diese Berechnungen ohne side effects stattfinden können, müssen wir nach jeder Berechnung die besuchten Knoten zurücksetzen. Das Gewicht der Knoten sowie der Elternknoten muss beibehalten werden, um zu verhindern, dass eine spätere Suche einen kürzeren Pfad überschreibt. Der aktuell kürzeste Pfad wird verfolgt. Dieser ist das Gewicht der Vorwärtssuche plus dem Gewicht der Rückwärtssuche vom Kreuzungsknoten. Bei jedem abgearbeiteten Knoten wird überprüft, ob dieser bereits von der anderen Suche abgearbeitet wurde. Wenn ja, wird das Gewicht des Pfades berechnet und der aktuelle kürzeste Pfad aktualisiert, falls er kürzer ist. Abgebrochen wird, wenn die Gewichte in beiden Queues größer sind als der kürzeste Pfad.

Nach der Suche müssen wir noch den Pfad rekonstruieren. Dabei erzeugen wir eine Liste, die alle Kanten der Vorwärtssuche beinhaltet. Jede Kante wird rekursive durch seine Ursprungskanten ersetzt, um den Pfad in $G_0$ zu bekommen. Gleiches in der Rückwärtssuche. Die beiden Pfade werden kombiniert, um den gesamten Pfad in $G_0$ zu erhalten.

## 7.7 Erzeugte Struktur

Im Kapitel 3 berechnen wir Schicht für Schicht der Highway Hierarchie. Was noch nicht besprochen wurde, ist, wie wir diese Struktur abspeichern.

In der Dissertation [Sch08] wird der Graph durch ein um Knotenschichten erweitertes Adjazenzarray repräsentiert. Die Struktur konnte nicht übernommen werden, da die Implementierung auf den aktuellen Standard aufsetzen soll.

Um eine Hierarchie darstellen zu können, müssen wir die Struktur erweitern. Es wurde eine Liste von Arrays herangezogen, damit das alte Konzept erhalten bleibt, wenn nur die Knoten der ersten Schicht verwendet werden. Jedes Array spiegelt eine Schicht wieder. Die Knoten aus Schicht $l$ sind mit den Knoten der Schicht $l-1$ über einen Downlink verlinkt. Wenn ein Knoten aus Schicht $l$ auch in $l+1$ vorkommt, dann existiert auch ein Uplink. Die Down- und Uplinks werden für die Abfrage benötigt.

Der Vorteil dieses Aufbaus ist, dass bei der Abfrage und Vorberechnung keine Level-Checks benötigt werden. Nachteil ist, dass die Knoten und Kanten redundant gespeichert werden, was den Speicherverbrauch erhöht.

# 8 Optimierungen

## 8.1 Parallelisierung der Vorberechnung

Diese Optimierung dient nur zur Beschleunigung der Vorberechnung. Während der Vorberechnung wird oft über die gesamte Knotenmenge iteriert, wobei eigenständige Berechnungen durchgeführt werden, was gute Voraussetzungen für eine Parallelisierung sind. Das gilt für den gesamten Nachbarschaftsradius, Highway-Hierarchies und Überlagerungsgraphen. Bei diesen Algorithmen kann die Knotenmenge auf verschiedene Prozesse aufgeteilt werden. Beim Kern wird eine gemeinsame Menge benötigt, um abzugleichen, welche Knoten bereits abgearbeitet wurden oder nicht. Jeder Prozess braucht eine eigene Gewichtmatrix, um side effects zu vermeiden.

## 8.2 Asynchrone Abfrage

Die Informationen zu diesem Abschnitt basieren auf [Sch08, S. 102-104]. Derzeit ist eine synchrone Abfrage implementiert. Diese ist einfacher umzusetzen, hat aber den Nachteil, dass die Laufzeit länger ist. Bei der synchronen Abfrage ist die Vorwärts- und Rückwärtssuche immer in derselben Schicht. Der Nachteil davon. Wenn sich die Berechnung der Covering Paths in einer Suche schwerer gestaltet, kann die andere Suche in der nächsten Schicht ebenfalls nicht fortsetzt werden, weil sich beide Suchen immer in derselben Schicht befinden müssen. Bei einer asynchronen Abfrage sind die beiden Suchen nicht gekoppelt. Sie fungieren vollkommen eigenständig. Dadurch kann sich jede Suche in unterschiedlichen Schichten befinden. Der Algorithmus lässt sich ebenfalls leichter parallelisieren, weil die Abhängigkeiten reduziert wurden. In Abbildung 5.1 kann eine Asynchronität beobachtet werden.

## 8.3 $A^*$ Erweiterung

Wir können den Covering-Paths Algorithmus um einen $A^*$ erweitern. Der Algorithmus wurde erstmals in [HNR68] veröffentlicht. Die Idee dahinter ist es, die Suche Richtung Ziel zu richten, um nicht kreisartig suchen zu müssen. Das ist möglich, indem wir die Distanz von jedem Knoten zum Ziel schätzen. Es ist eine Erweiterung vom Dijkstra-Algorithmus. Dabei bleibt alles identisch bis auf die Gewichtsfunktion. Diese wird um folgende Heuristik erweitert.

$$f(n) = g(n) + h(n)$$

- $f(n)$ beschreibt die Gewichtsfunktion
- $g(n)$ beschreibt die wirklichen Kosten vom Startknoten zu $n$
- $h(n)$ beschreibt die Heuristik, eine Schätzung der Distanz von $n$ zum Zielknoten

$g(n)$ entspricht der aktuellen Gewichtsfunktion. Damit der Algorithmus korrekt bleibt, muss $h(n)$ eine Eigenschaft erfüllen. Sie darf die wirklichen Kosten nicht überschätzen. Das heißt, wenn $h^*(n)$ die Kosten des wirklichen kürzesten Pfads beschreibt, dann muss $\forall n \in V_l$ : $h(n) \leq h^*(n)$ gelten. Folglich ist $f(n)$ immer kleiner gleich als das Gewicht des kürzesten Pfads. Es sei auch angemerkt, dass $h(n)$ vom Zielknoten null sein muss, denn die kürzeste Distanz zu sich selbst ist null. Außerdem entspricht das Gewicht $f(t) = g(t)$ vom Zielknoten

$t$, jenen vom Dijkstra-Algorithmus. Als Beispiel für eine korrekte Heuristik wäre die Luftline. Diese ist immer kleiner als die wirkliche Distanz. In Abbildung 8.1 ist der Dijkstra dem $A^*$ gegenübergestellt.

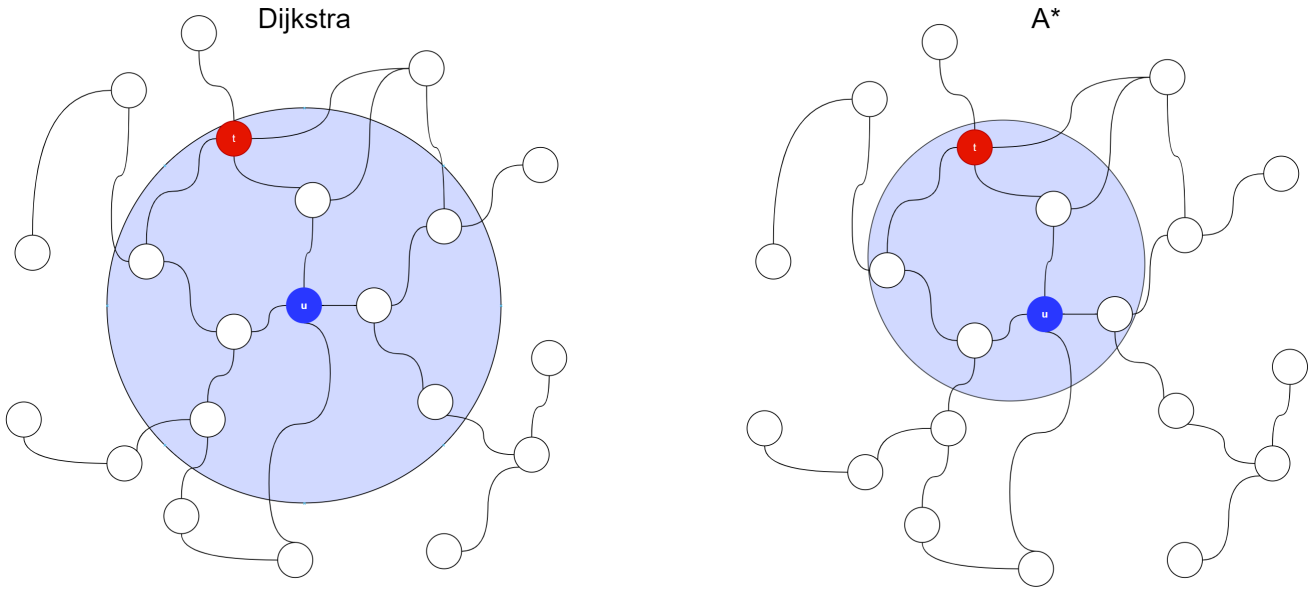



**Abbildung 8.1:** Vergleich der Suchbäume zwischen Dijkstra und $A^*$. Der blaue Bereich spiegelt den jeweiligen Suchbaum wieder. $u$ ist der Start- und $t$ der Endknoten.

## 8.4 Transit Node Routing

Tansit-Node-Routing ist eine Erweiterung von Highway Node Routing. Hier wird nur die grundsätzliche Idee skizziert, für detailliertere Informationen zur Funktionsweise verweise ich auf [Sch08, Chapter 6]. Der Algorithmus beruht auf der Tatsache, dass es Schlüsselknoten gibt. Nehmen wir an, wir fahren weiter weg. Dann gibt es ein paar entscheidende Kreuzungen, durch welche wir unsere aktuelle Position verlassen. Es ähnelt dem Konzept der wichtigen Straßen der Highway-Hierarchien. Jeder Knoten besitzt eine Menge dieser Kreuzungen, welche Access-Nodes genannt werden. Wir unterscheiden zwischen Vorwärts- und Rückwärts Access-Nodes. Die Vereinigungsmenge der Access-Nodes des Startknotens sowie des Endknotens im Rückwärtsgraphen werden Transit-Nodes bezeichnet. Für einen Algorithmus, der funktioniert, brauchen wir noch einen Lokalitätsfilter. Dieser beschreibt, ob der Start- und Endknoten lokal zueinander sind oder nicht. Lokal bedeutet, die beiden Knoten sind so nahe beisammen, das die Route nicht über Transit-Nodes erfolgt. Die kürzeste Route besteht aus

$$d_\tau(s,t) = min\{d(s,u) + d(u,v) + d(v,t) \mid u \in \overrightarrow{A}(s), v \in \overleftarrow{A}(t)\}$$

wenn die beiden Knoten nicht lokal sind. $\overrightarrow{A}(s)$ beschreiben die Access-Nodes vom Startknoten im Vorwärtsgraphen. $\overleftarrow{A}(t)$ beschreiben die Access-Nodes vom Endknoten im Rückwärtsgraphen. Wobei alle Distanzen vorberechnet und zur Abfragezeit nachgeschlagen werden. Die Menge der Transit-Nodes $\overrightarrow{A}(s) \cup \overleftarrow{A}(t)$ ist relative klein und somit muss nur eine kleine Anzahl an Kombinationen probiert werden, um die kürzeste Route zu finden. Wenn die beiden Knoten lokal sind, erzielt man keine Beschleunigung, aber hier kann die Route relative schnell berechnet werden. Das Konzept kann ebenfalls auf mehrere Schichten erweitert werden.

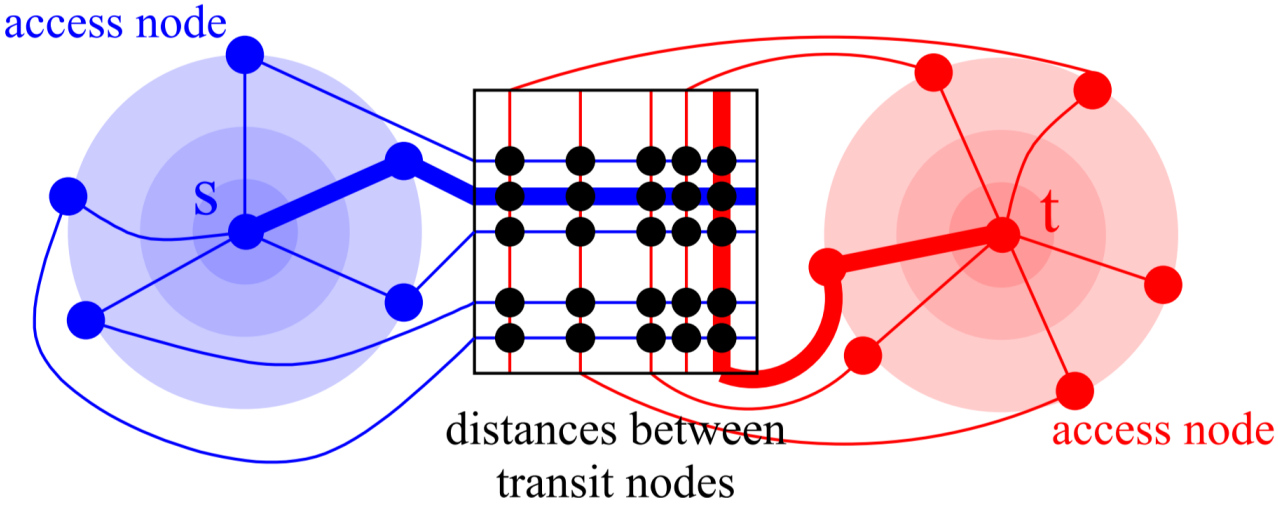



**Abbildung 8.2:** [Sch08, Figure 6.1] Veranschaulichung der Idee von Transit-Node-Routing. Die blauen Punkte sind die Access-Nodes vom Startkonten, die roten vom Endknoten. Die beiden sind nicht lokal. Um die kürzeste Route zu finden, werden alle Kombinationen der Access-Nodes geprüft, was eine Reihe von Tabellenabfragen gleicht.

# 9 Experimente

In diesem Abschnitt betrachten wir die Auswirkungen auf die Hierarchie und die Laufzeit, wenn der Nachbarschaftsradius $H$ oder der Überbrückungsfaktor $c$ verändert wird. Wir variieren nur $H$ und $c$. Der Sprungzähler (Kapitel 3.1.3) wurde auf 10 sowie der Maverikfaktor (Kapitel 3.3.1.1.2) auf 2 fixiert. Diese beiden wurden nicht näher betrachtet, da ihr Einfluss auf das Endergebnis ein kleinerer ist. Die Laufzeit zum Berechnen der Routen basiert auf einfachen Messungen. Dadurch hängt das Ergebnis stärker von äußeren Faktoren ab, wie die aktuelle Auslastung des Computers. Um das Ergebnis mehr Ausschlagkraft zu geben, wurde jeweils der Dijkstra-Algorithmus mit ausgeführt, um einen Referenzwert zu generieren. In den Laufzeittabellen ist eine Spalte HNR angeführt. HNR steht für **H**ighway **N**ode **R**outing. Die Anzahl an Schichten wurde ebenfalls auf 8 fixiert. Die Hierarchie ändert sich nach 8 Schichten kaum.

Die Routen wurden zufällig von mir gewählt. Dabei wurde darauf geachtet, dass sich die Längen der Routen unterscheiden.

- Route 1: Länge = 687.92 km
- Route 2: Länge = 754.55 km
- Route 3: Länge = 253.52 km

## 9.1 Variation des Nachbarschaftsradius

Bei allen Graphen wurde der Überbrückungsfaktor $c$ auf 2 fixiert um nur die Auswirkungen des Nachbarschaftsradius zu beobachten. In der Tabelle 9.1 kann man erkennen, dass die

| | H = 50 | | H = 150 | | H = 300 | |
|---|---|---|---|---|---|---|
| Schicht | Knoten | Kanten | Knoten | Kanten | Knoten | Kanten |
| 0 | 331 094 | 663 347 | 331 094 | 663 347 | 331 094 | 663 347 |
| 1 | 51 719 | 133 429 | 44 397 | 114 454 | 38 081 | 99 215 |
| 2 | 13 420 | 53 337 | 9 761 | 39 155 | 6 993 | 29 502 |
| 3 | 6 202 | 29 308 | 3 848 | 18 067 | 2 419 | 10 851 |
| 4 | 4 397 | 20 175 | 2 442 | 10 571 | 1 397 | 5 134 |
| 5 | 3 697 | 16 029 | 1 849 | 7 085 | 1 065 | 3 521 |
| 6 | 3 369 | 14 045 | 1 584 | 5 743 | 890 | 3 109 |
| 7 | 3 129 | 13 050 | 1 357 | 4 754 | 761 | 2 788 |

**Tabelle 9.1:** Veränderung der Kanten- und Knotenmenge

Kanten und Knotenmenge mit steigendem Nachbarschaftsradius abnimmt. Was auch unseren Erwartungen entspricht. Wir fügen nur Kanten zur Hierarchie hinzu, die nicht lokal sind. Wenn der lokale Bereich wächst, gehören auch weniger Kanten zur Hierarchie, was wir in der schnelleren Komprimierung des Graphen erkennen können.

Eine stärkere Komprimierung gleicht nicht automatisch einer schnelleren Laufzeit. Denn wenn der Graph gerade am Anfang stark komprimiert wird, wechseln wir später in eine höhere Schicht, was sich negativ auf die Laufzeit auswirkt. Gleiches gilt, wenn der Graph kaum komprimiert wird. In Tabelle 9.2 erkennen wir dieses Verhalten wieder. Der Hyperparameter $H = 150$ liefert das beste Ergebnis. $H = 50$ komprimiert zu schwach. $H = 300$ zu stark. Wir können auch entnehmen, dass der Algorithmus bei weiteren Stecken eine kürzere Laufzeit aufweist und keine längere. Der Grund für dieses Verhalten kann in einer schnelleren Berechnung der Covering-Paths liegen oder die anderen Routen verwendeten die letzten Schichten nicht ausreichend.

| | H = 50 | | H = 150 | | H = 300 | |
|---|---|---|---|---|---|---|
| Route | HNR | Dijkstra | HNR | Dijkstra | HNR | Dijkstra |
| Route 1 | 0.1326 s | 0.7259 s | 0.1383 s | 0.6526 s | 0.1993 s | 0.6842 s |
| Route 2 | 0.1634 s | 0.6426 s | 0.0873 s | 0.6561 s | 0.1029 s | 0.6917 s |
| Route 3 | 0.2104 s | 0.4055 s | 0.1581 s | 0.4092 s | 0.1983 s | 0.3263 s |

**Tabelle 9.2:** Veränderung der Laufzeit

## 9.2 Variation der Überbrückungsmenge

Bei allen Graphen wurde der Nachbarschaftsradius auf $H = 300$ fixiert, um nur die Auswirkungen der Überbrückungsmenge zu beobachten. In der Tabelle 9.3 können wir erkennen, dass sich die Graphen von $c = 1 \mid 1.5 \mid 2$ nicht viel unterscheiden. Grund dafür kann die Häufigkeit der Knoten rund um den Bereich 1-2 sein und es gibt wenige, die nur von einem größeren Faktor berücksichtigt werden. Ein größerer Unterschied existiert zwischen $c = 0.5$ und dem Rest. Der geringe Unterschied zieht sich in der Laufzeit fort. Es gibt nicht wie bei dem Nachbarschaftsradius einen eindeutigen besten Parameterwert. Die Unterschiede können rein von den äußeren Faktoren stammen.

| | c = 0.5 | | c = 1 | | c = 1.5 | | c = 2 | |
|---|---|---|---|---|---|---|---|---|
| Schicht | Knoten | Kanten | Knoten | Kanten | Knoten | Kanten | Knoten | Kanten |
| 0 | 331 094 | 663 347 | 331 094 | 663 347 | 331 094 | 663 347 | 331 094 | 663 347 |
| 1 | 50 306 | 110 746 | 38 718 | 98 550 | 38 233 | 99 333 | 38 081 | 99 215 |
| 2 | 14 706 | 38 005 | 7 177 | 27 158 | 7 003 | 29 192 | 6 993 | 29 502 |
| 3 | 7 504 | 20 115 | 2 543 | 9 944 | 2 448 | 10 728 | 2 419 | 10 851 |
| 4 | 4 574 | 12 708 | 1 432 | 4 793 | 1 418 | 5 402 | 1 397 | 5 134 |
| 5 | 3 128 | 8 623 | 1 126 | 3 672 | 1 001 | 3 460 | 1 065 | 3 521 |
| 6 | 2 204 | 6 080 | 960 | 3 273 | 838 | 2 912 | 890 | 3 109 |
| 7 | 1 860 | 4 730 | 865 | 3 452 | 714 | 2 562 | 761 | 2 788 |

**Tabelle 9.3:** Veränderung der Kanten- und Knotenmenge

| | c = 0.5 | | c = 1 | | c = 1.5 | | c = 2 | |
|---|---|---|---|---|---|---|---|---|
| Route | HNR | Dijkstra | HNR | Dijkstra | HNR | Dijkstra | HNR | Dijkstra |
| Route 1 | 0.1269 s | 0.6149 s | 0.1331 s | 0.5863 s | 0.2014 s | 0.6126 s | 0.1993 s | 0.6842 s |
| Route 2 | 0.1182 s | 0.6419 s | 0.1355 s | 0.6854 s | 0.0990 s | 0.7404 s | 0.1029 s | 0.6917 s |
| Route 3 | 0.1950 s | 0.3610 s | 0.1292 s | 0.3240 s | 0.2100 s | 0.3544 s | 0.1983 s | 0.3263 s |

**Tabelle 9.4:** Veränderung der Laufzeit

## 9.3 Zusammenfassung

Highway Node Routing hat im Schnitt eine kürzere Laufzeit als der Dijkstra-Algorithmus. Somit sind unsere Bemühungen nicht umsonst. Der Laufzeitvorteil wird bei größeren Graphen noch steigen. Die höheren Schichten werden bei längeren Routen nützlicher, weil durch eine Kante eine viel längere Strecke zurückgelegt wird. Die Auswahl der Hyperparameter (Nachbarschaftsradius, Überbrückungsfaktor, Sprungzähler und Maverikfaktor) muss auf den jeweiligen Graphen angepasst werden, um eine optimale Beschleunigung zu erzielen. Diese Faktoren müssen empirisch ermittelt werden, da es keine Berechnung gibt, um den optimalen Wert zu finden. Dazu können Methoden wie Gittersuche aus dem maschinellen Lernen herangezogen werden. Die richtige Wahl der Parameter ist essenziell, um das volle Potential des Algorithmus zu nutzen.

# Literatur

[DGJ09] Demetrescu, C. ; Goldberg, A. und Johnson, D., Hrsg.: *The Shortest Path Problem*. Bd. 74. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. Providence, Rhode Island: American Mathematical Society, 2009.

[DI04] Demetrescu, C. und Italiano, G. F.: „Engineering Shortest Path Algorithms“. In: *Experimental and Efficient Algorithms*. Bd. 1. Springer Berlin Heidelberg, 2004, S. 191–198.

[Dij59] Dijkstra, E. W.: „A note on two problems in connexion with graphs“. In: *Numerische Mathematik* 1 (1959), S. 269–271.

[HNR68] Hart, P. E. ; Nilsson, N. J. und Raphael, B.: „A Formal Basis for the Heuristic Determination of Minimum Cost Paths“. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968). Conference Name: IEEE Transactions on Systems Science and Cybernetics, S. 100–107.

[Sch05] Schultes, D.: „Fast and Exact Shortest Path Queries Using Highway Hierarchies“. In: *Master's thesis, Universität des Saarlandes* (2005).

[Sch08] Schultes, D.: „Route Planning in Road Networks“. In: *Dissertation, Universität Fridericiana zu Karlsruhe (TH)* (2008), S. 1–235.

[SS05] Sanders, P. und Schultes, D.: „Highway Hierarchies Hasten Exact Shortest Path Queries“. In: *Algorithms – ESA 2005*. Hrsg. von Brodal, G. S. und Leonardi, S. Bearb. von Hutchison, D. u. a. Bd. 3669. Series Title: Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, S. 568–579.

[SS06] Sanders, P. und Schultes, D.: „Engineering Highway Hierarchies“. In: *European Symposium on Algorithms (ESA)*. Bd. 4168. Springer, 2006, S. 804–816.

[SS07] Schultes, D. und Sanders, P.: „Dynamic Highway-Node Routing“. In: *Experimental Algorithms*. Hrsg. von Demetrescu, C. Bd. 4525. Series Title: Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, S. 66–79.

[SS12] Sanders, P. und Schultes, D.: „Engineering highway hierarchies“. In: *ACM J. Exp. Algorithmics* 17 (2012), 1.6:1.1–1.6:1.40.